From the vault of my engineering newsletter
"Systems That Scale"

Saurav Prateek's

Data Structures behind our Databases

Explaining the **Data-Structures** and **Datastores** that powers the Databases.

Table of Contents

Chapter 1

Data Structure 1: Hash Indexes

We will discuss the Hash Indexes data structures that power the Databases under the hood. This chapter will mainly focus on Hash Maps or Hash Tables and how they can be combined with the file-based data stores and leveraged by the Databases to store the data.

Introduction

A database needs to do two things fundamentally:

- When we give it some data, it should store that data.
- When we ask for the same data, it should return it.

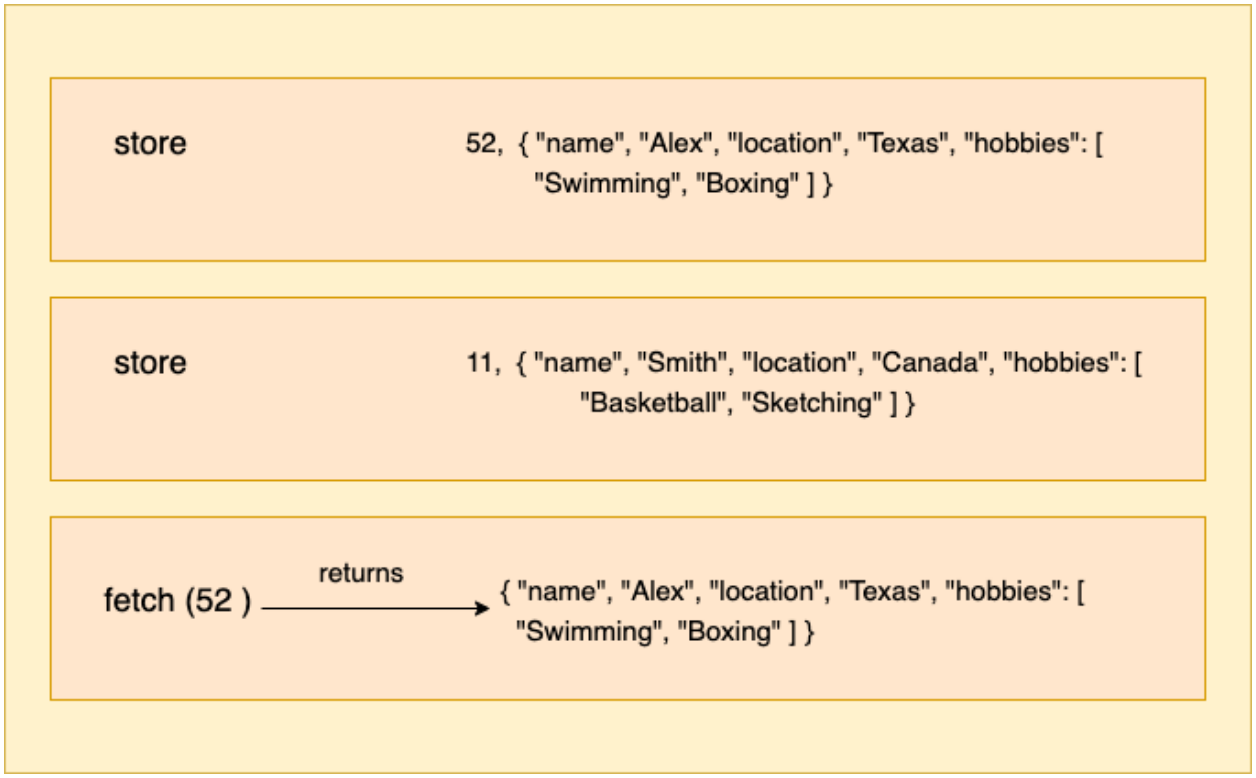
In this edition we will majorly discuss some Data Structures that are widely used by the Databases to store the data. Being an engineer or an architect there can be situations where we are expected to pick a Database for our System. In those situations, having a rough idea about how these storage systems function under the hood can help us in picking the right storage system that is appropriate for our application.

A Naive Data Structure for our Database

Suppose there is a database that stores our data in a key-value format. It primarily performs two major operations:

- **store(key, value)**: Stores the given key-value pair in the data store.
- **fetch (key)**: Fetches the most recently updated value from the data store for a given key.

The value can be anything ranging from simple numbers, strings or timestamps to complex JSON documents as well. The database works like this.

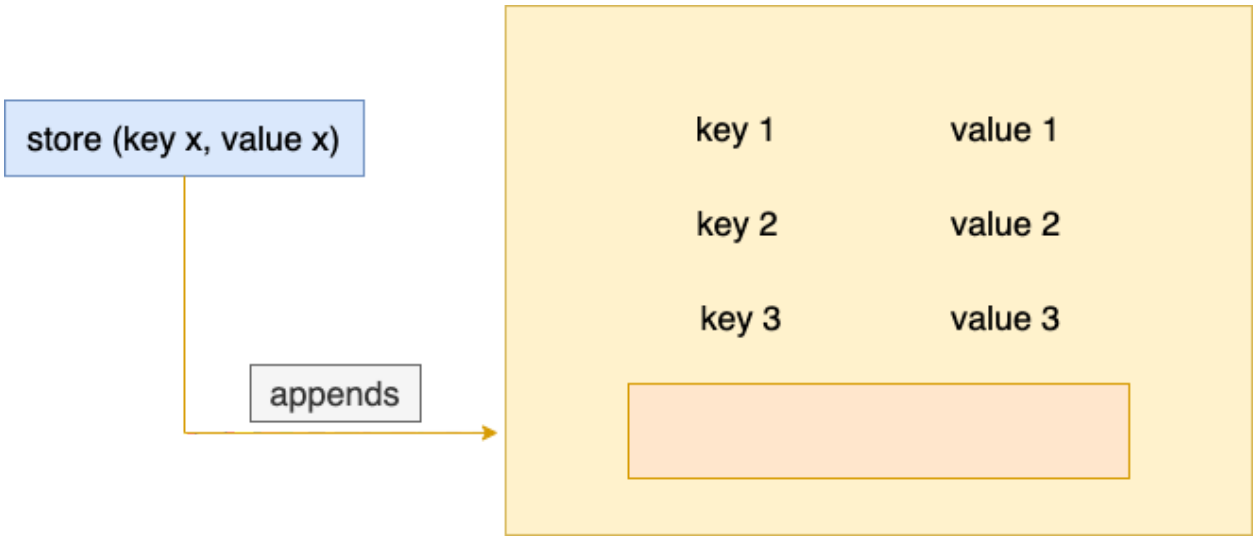


Now let's discuss how this database actually stores the data under the hood. The database uses a fairly simple **text file-based** storage system in order to store the data. The key-value pair is stored in a comma separated format inside the file.

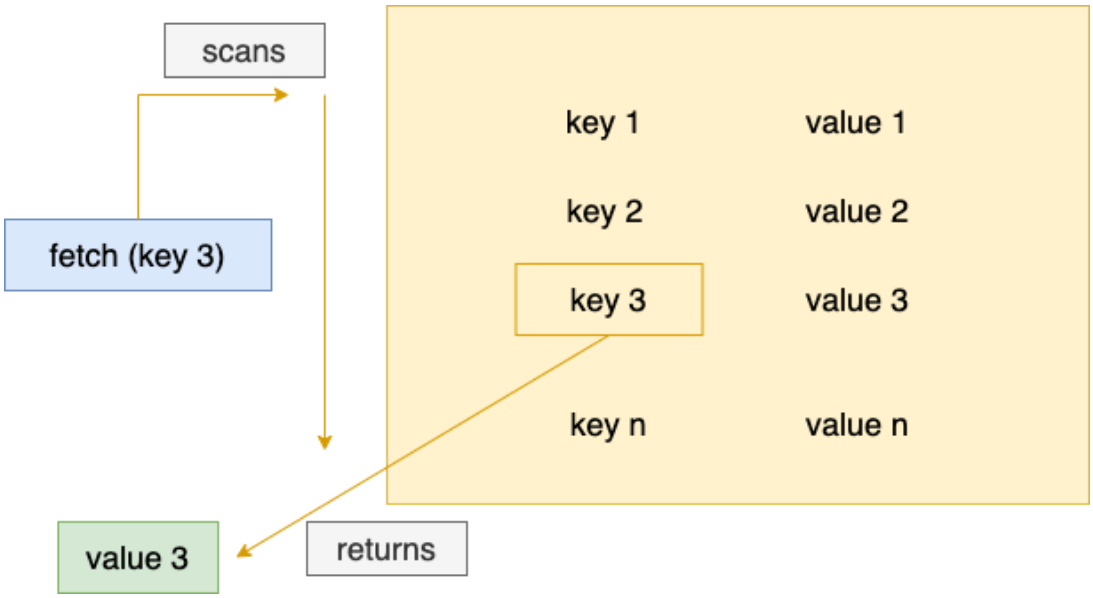
The **text file** structure looks like this.

```
52, { "name", "Alex", "location", "Texas", "hobbies": [
      "Swimming", "Boxing" ] }

11, { "name", "Smith", "location", "Canada",
      "hobbies": [ "Basketball", "Sketching" ] }

...
```

When we perform a **store(key, value)** operation to store the key-value pair in the data store then the database simply appends that key-value pair entry in the text file.



When we perform the **fetch(key)** operation then the database looks for that key by scanning the text file and returns the corresponding value back to the user.



Discussing Complexities of the Operations

The **store(key, value)** operation simply appends the key-value entry to the data store and hence it has a pretty good performance. We can consider the complexity of this operation to be constant or **O(1)**.

On the other hand, the **fetch(key)** operation scans the entire data store looking for the most recent value for the given key. Hence, we can say that the fetch operation has a poor performance in case we have a large number of records in our database. We can consider the complexity of this operation to be O(N) where N is the number of records present in our database. Suppose in future the number of records in our data store doubles, then the fetch operation will take twice as long to fetch a single record from the database.

Indexing

In the previous architecture we observed that the fetch(key) operation had a terrible performance, since it required to scan the entire data store. But we can optimize the performance by introducing Indexing in our database. We can add indexes on the keys and can reduce the look-up time to some extent. Many databases allow us to add or remove indexes and this doesn't affect the contents of the database, it only affects the performance of the queries.

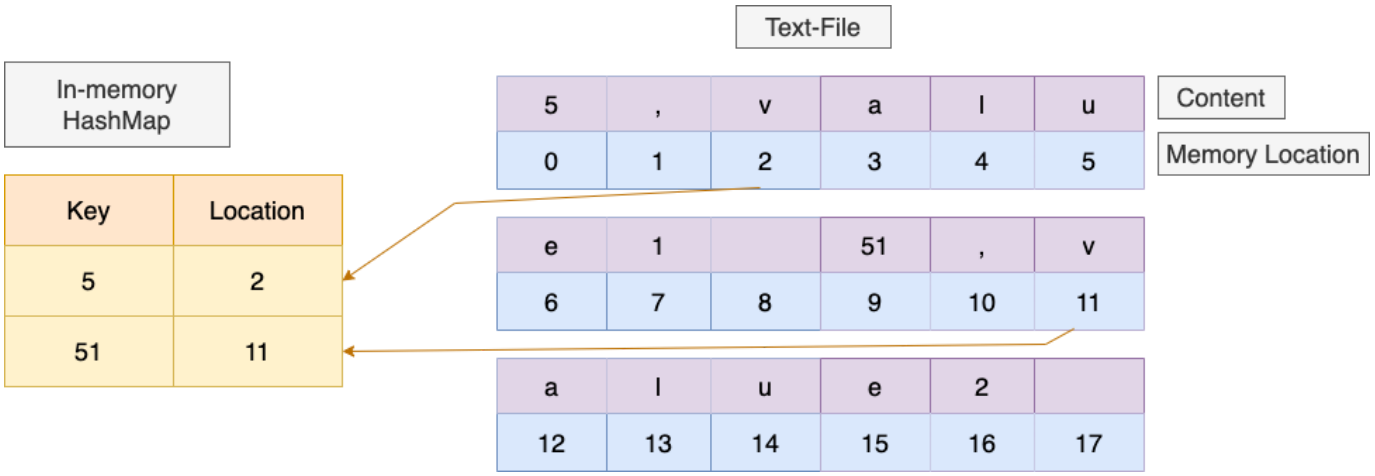
Indexing helps in optimizing the look-up time for the keys, but it also increases the time for the store operation. Initially the store/write operation simply appends the key-value entry to the file. After indexing is applied the indexes also need to be updated every time the data is written to the datastore. Hence, in a nutshell Indexing optimizes the read queries but slows down the write operations.

Due to this reason, databases do not usually index everything by default. It requires us to add indexes on the basis of our system's query patterns. Let's discuss one data structure that uses the concept of Indexing for fast look-ups in the next section.

File Storage with Hash Tables

Let's consider a data store that stores data in a key-value format. We will use a **HashMap** (Hash Table) in order to enhance the lookup performance. It is similar to the **Dictionary** type.

Earlier we appended our key-value pair entry in a text file under the **store(key, value)** operation. But this required us to scan the entire data store in order to fetch a key-value pair under the **fetch(key)** operation. We can leverage our HashMap data structure to index our data. We can use an in-memory HashMap in which every key is mapped to the location in the test file where the corresponding value can be found. Hence, whenever we append a key-value pair in our data store the location of the new value must be updated in our HashMap. When we need to fetch the value for a particular key, then we can simply need to get the location of the value from the HashMap and further seek to that location in the Text File and read the value.



One advantage here is that we can have very large **values** that can not fit in the memory for a particular **key** and still our architecture performs well. Since, we won't be storing the entire value in our in-memory HashMap but just the location of the value present in the Text File data-store.

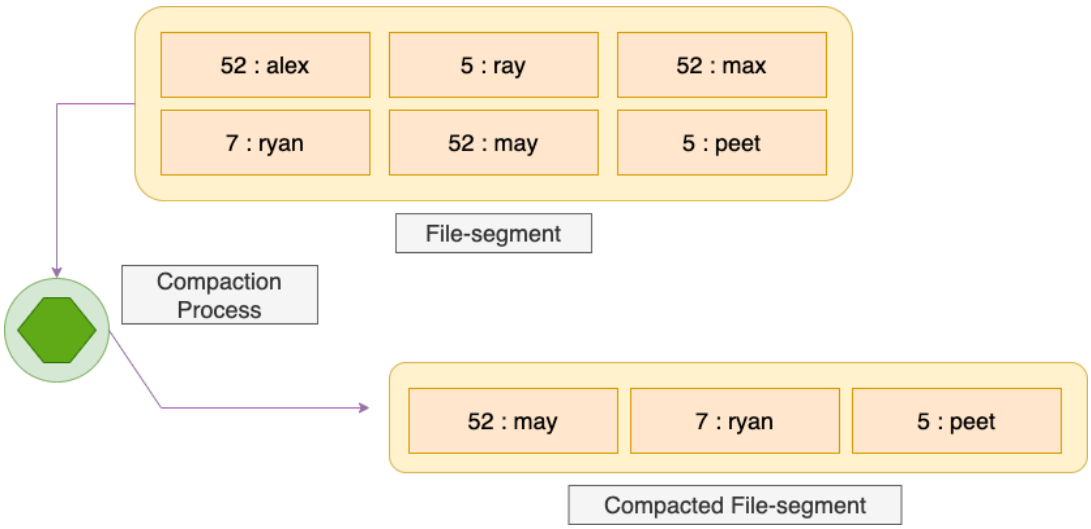
The read operation is also optimized since we are no longer required to scan the entire data store. We just need to fetch the location of the value for that key from the in-memory HashMap and then fetch the value from that location in the Text File. This architecture also works well in the systems that are **write-heavy**.

Optimizing File Storage through Compaction and Merging

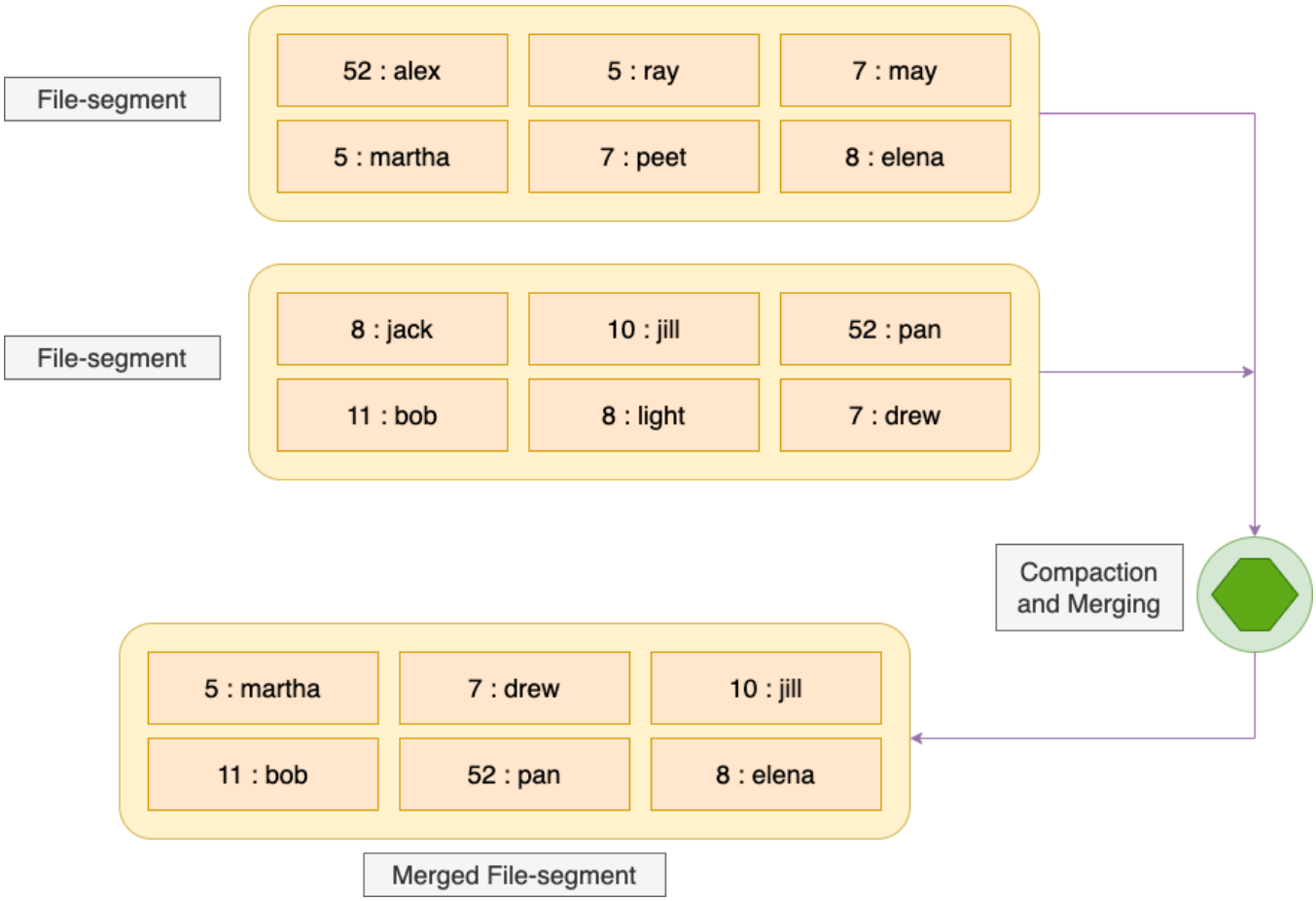
Since we always append to the text file whenever a key-value entry comes up to be stored, there can be a situation at a later point of time where our text file can run out of the disk space. To avoid this we can break our Text-File into small fixed size **segments**, Once the File-segment exceeds its size, we can start writing our key-value pair to a new segment. Then we can perform **Compaction** on these segments.

> "Compaction means throwing away the duplicate keys in the segment and keeping only the most recent one"

The compaction process on a segment can look something like this.



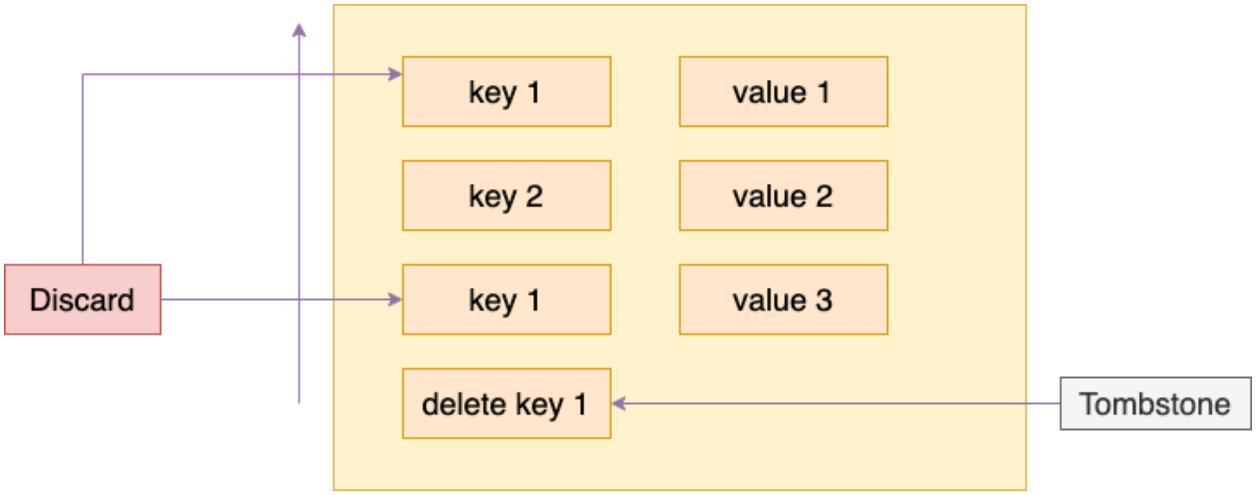
Now the **Compaction** process reduces the size of a File-segment. Hence we can **Merge** multiple segments at the same time while the **Compaction** process is taking place. The entire process looks like this.



Once the **Merging** and **Compaction** process is completed and we have our new merged segment, we can simply discard all the old file segments and start writing the key-value pair to the newly merged segment.

Deleting the Key from the Data Store

The deletion of a Key from the data-store takes place very differently under this scheme. A new entry for deleting a Key is appended in the text file. The entry is known as Tombstone. When the file segments are merged then the tombstone tells the system to discard all the previous values for the deleted key.



Cache Recovery

If the server is restarted then the entire in-memory **HashMap** will be lost. But we can build the HashMap by scanning the file segments and storing the keys with the location of its most recently updated value.

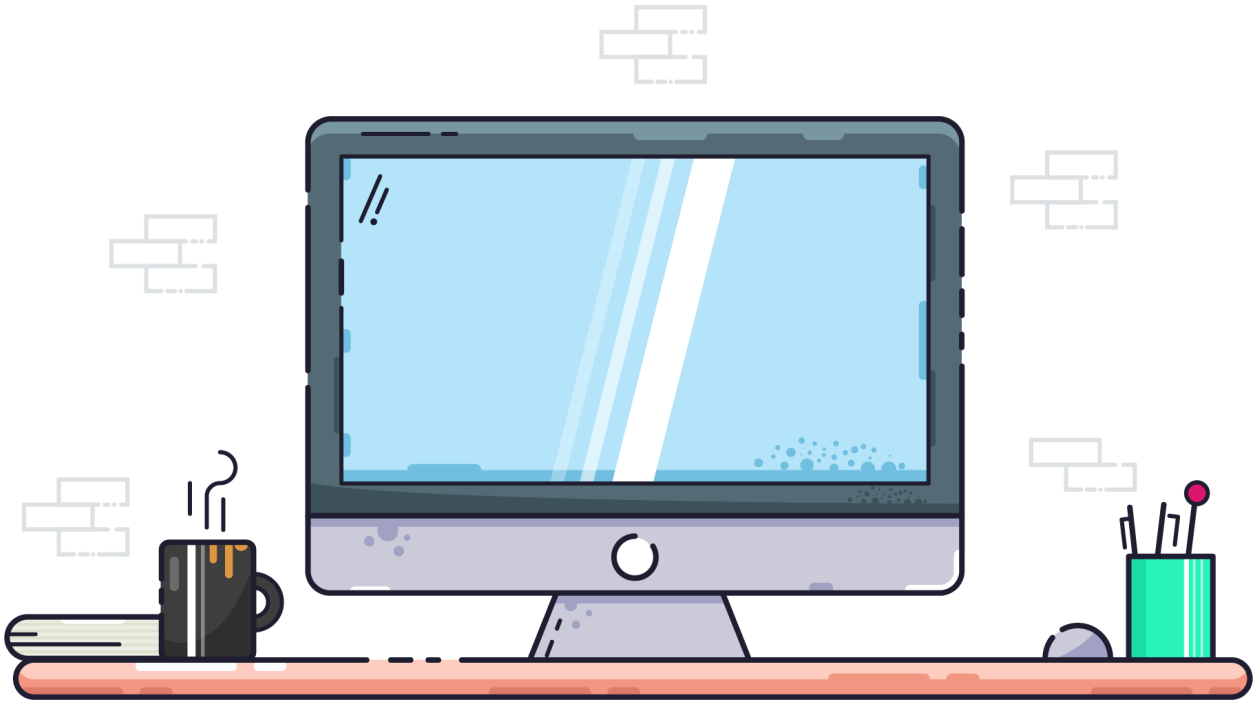
This can be a lengthy process, hence we can also store the snapshot of the in-memory HashMap on the disk and later retrieve it in case of failures.

Drawbacks

If the data-store has a **large number of keys** then it might be difficult to fit the HashMap in a single memory. As an alternative, we can store our HashMap on the disk but then constantly querying the map for the keys and location can be costly since it's no longer present in the memory.

Summary

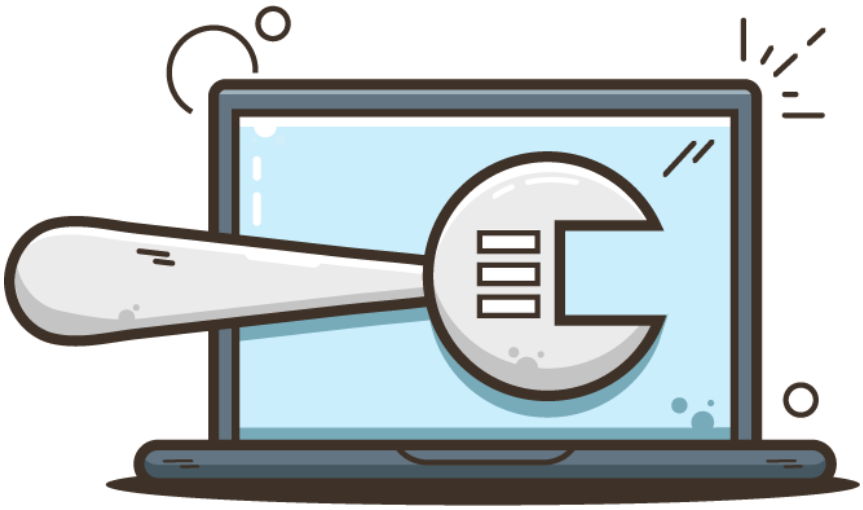
We discussed the combination of **File Based** data-store and **Hash Based** data structures used by the databases to store the data. We also looked around various operations performed by these data structures under the hood, such as **Compaction** and **Merging** and the complexities and performance enhancement of the read/write queries for the database.



Chapter 2

Data Structure 2: Log Structured Merge (LSM) trees

We will discuss **LSM Trees**, **SS Tables** and **Memtables** and how they power the Databases to perform optimized Reads and Writes.



Introduction

In our previous chapter we discussed **Hashing** and **Indexed** based Data Structures that can be used by the Databases under the hood. In this edition we will majorly focus on **Log-Structured Merge-Trees (LSM)** and how they are used by multiple Databases to store the data under the hood. They also allow an efficient read and write operation over the Database.

Log-Structured Merge-Trees

Log-Structured Merge-Trees is a data structure widely used by the databases to implement their datastore. This data structure provides indexed access to files with high insert volume, such as Transactional Log Data. LSM Trees also store data in a key-value pair format.

The LSM Trees use two separate structures to store data and perform optimized **Reads** and **Writes** over them. The first structure is smaller and stored in the **memory** in the form of a **Red Black Tree** or an **AVL Tree**. This structure is also known as Memtable. The second structure is comparatively larger and is present on the **Disk**, It is stored in the form of **SS Tables** also known as **Sorted String Tables**.

Lets understand how these two structures work together to provide optimized reads and writes.

Memtables

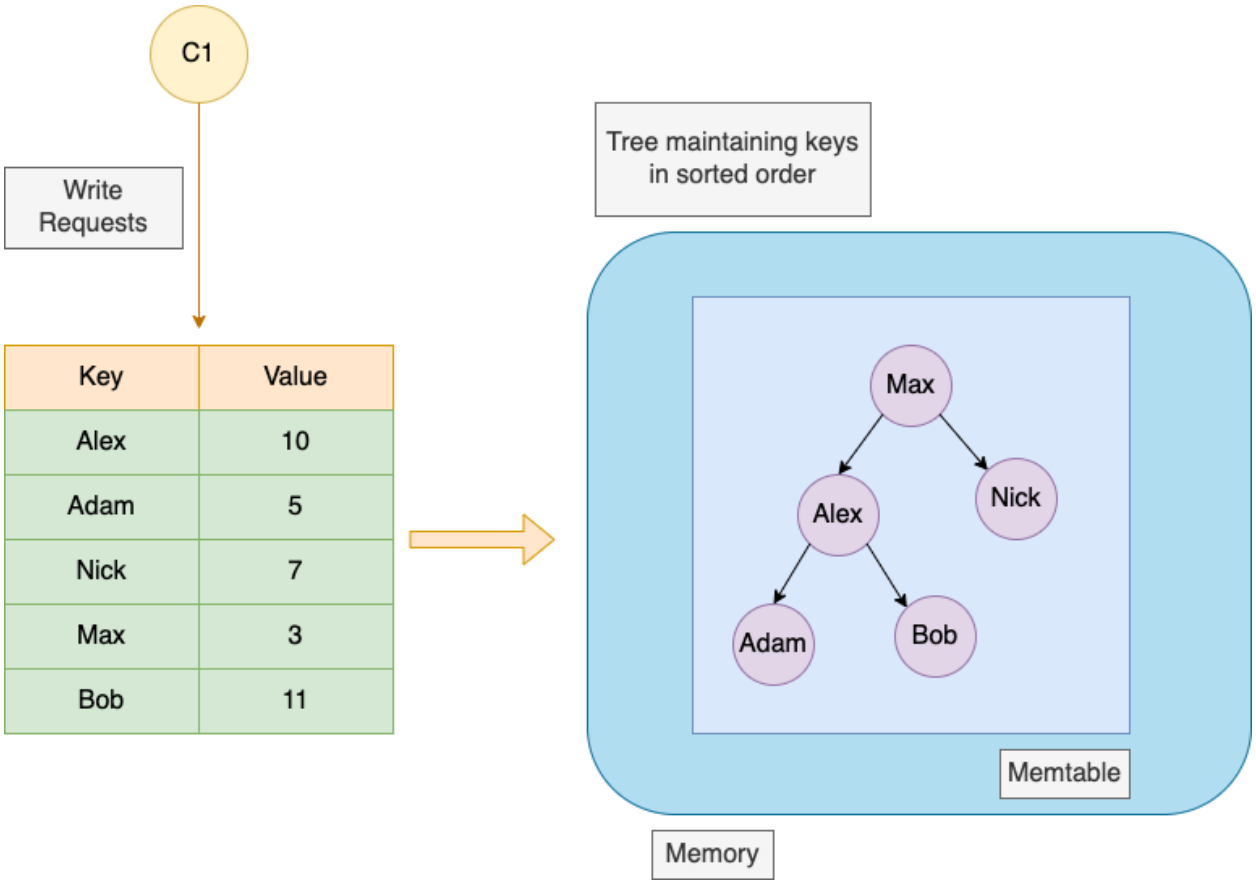
The **Memtables** are maintained in the memory and every Database write is directed to this Memtable. These Memtables are generally an implementation of Tree data structure in order to maintain the keys in a sorted format. It can be a **Red-Black Tree** or an **AVL Tree**.

Writing to the Memtable

When **Write** is issued to the Database it is simply written to the Memtable in the memory. Since the writes take place in the memory, the process is very fast.

Under the hood we have used a Tree Structure in order to store keys in a sorted format. Since the Tree structure used here is a **Self Balancing** tree, it guarantees the Writes and Reads to be performed within **Logarithmic** time.

We can use **AVL Tree** for this implementation which supports addition of nodes in logarithmic time and maintains them in a sorted order. They also allow reading all the nodes in a sorted format in linear time. This advantage of the AVL Tree will help us in future when we flush the data to the disk from the memory.

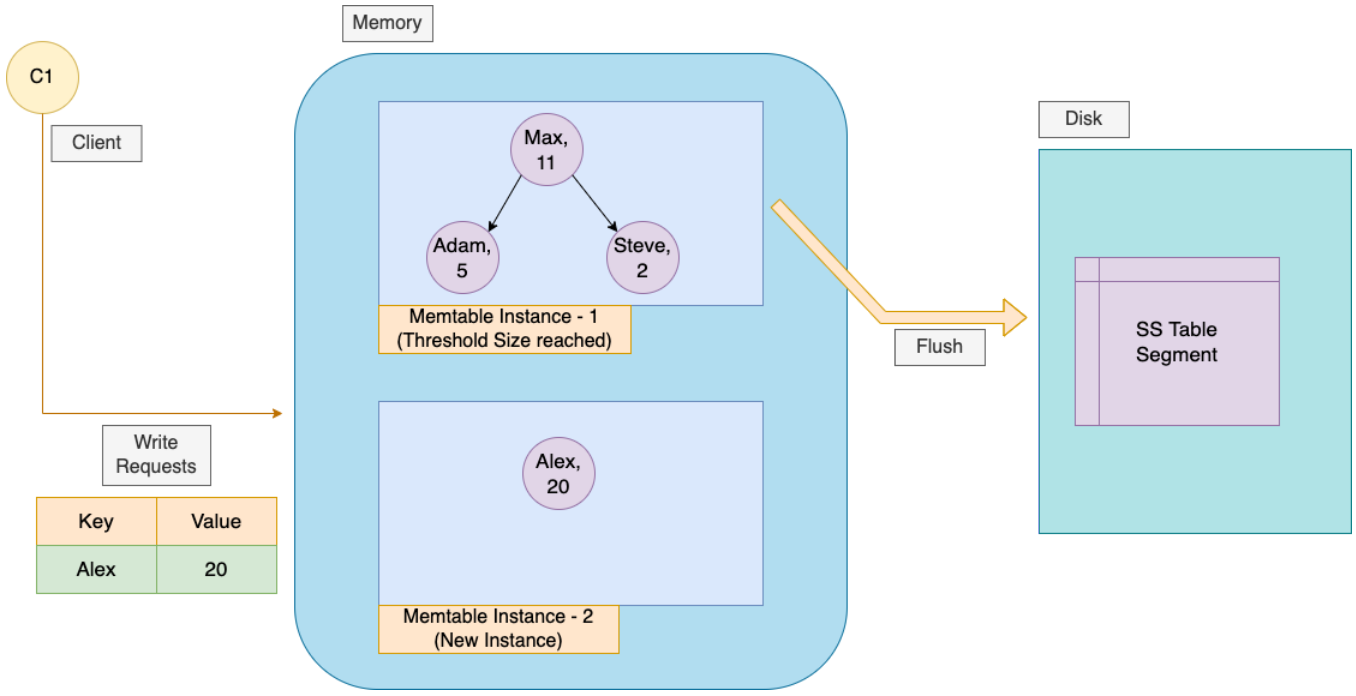


Writing the Memtable into the Disk

We keep a **threshold** value for the Memtable. If the size of our memtable exceeds the threshold value then we stop writing the key-value pairs to that instance of the Memtable.

Once the new instance is created and we start accepting the future writes to this memtable, we can flush the content of the previous memtable into the Disk. The memtables are nothing but a self balancing tree data structure that maintains the keys in a sorted order. Hence it's very easy to iterate over the keys in a sorted order. The keys are read in a sorted order from the memtable and are further stored on the Disk.

On the Disk, the data is stored in the form of the **SS Tables** also known as **Sorted String Tables**. We will discuss this in our next section. Our Database keeps flushing the key-value pairs (in sorted order) to the disk once the size of the memtable reaches the threshold limit.



Fault Tolerance in Memtables

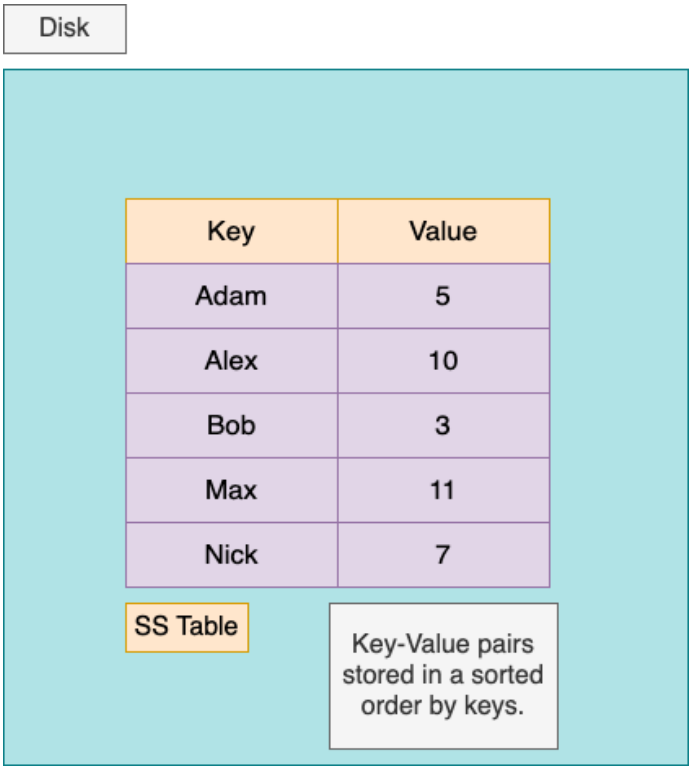
Since we are performing the writes in the memory, there is one problem that comes along with it. What if the Database crashes? We will lose all the Writes which are currently present in memory (**Memtable**) and are not written to the Disk (**SS Tables**) yet.

In order to resolve this we can maintain a separate Log on disk just to store the in-memory writes. In case of failure we can easily regenerate our Memtable from the backup Log. Once the contents of the Memtable is flushed to the Disk, we can simply discard the Log.

Sorted String Tables (SS Tables)

SS Tables also known as **Sorted String Tables** keep the key-value pair sorted by keys. We discussed in our previous section, how the Writes are performed in the memory into the Memtables and later the content of the memtable is flushed into the Disk.

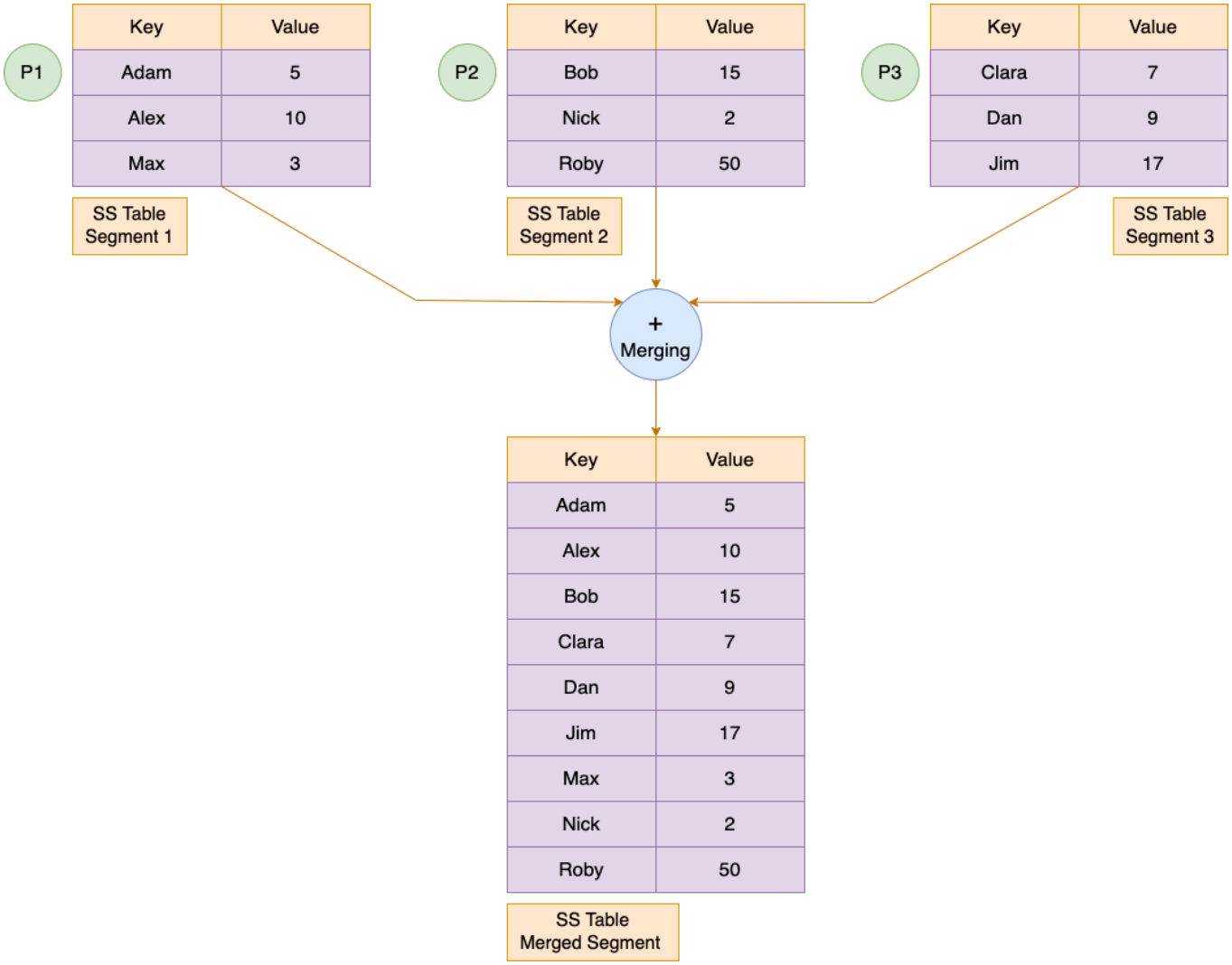
Now on the Disk, we maintain the **key-value** pairs in a different data structure known as **SS Tables**. The SS Tables might look like this in the disk.



Merging and Compaction of SS Tables

Since we were flushing every **Memtable** into an **SS Table** on the disk, there can be multiple segments of SS Tables present on the disk. In order to optimize the **Reads** and free up some space on the disk we can compact and merge these segments of SS Tables into a single segment of SS Table.

Suppose we have three segments of SS Tables with us over the disk. The merging process will look similar to the process we follow in the Merge-Sort Algorithm to merge two sorted arrays. The idea is to keep a pointer at the start of each Table segment and pick the smallest key and add it to the resulting segment and move that pointer one step ahead. We keep iterating over this process until all the key value pairs in all segments are pushed to the resulting segment.



While Merging we also take care of the duplicate keys. If the same key appears in multiple segments then we keep the key from the most recent segment and discard the keys from the older segments. This process is known as **Compaction**.

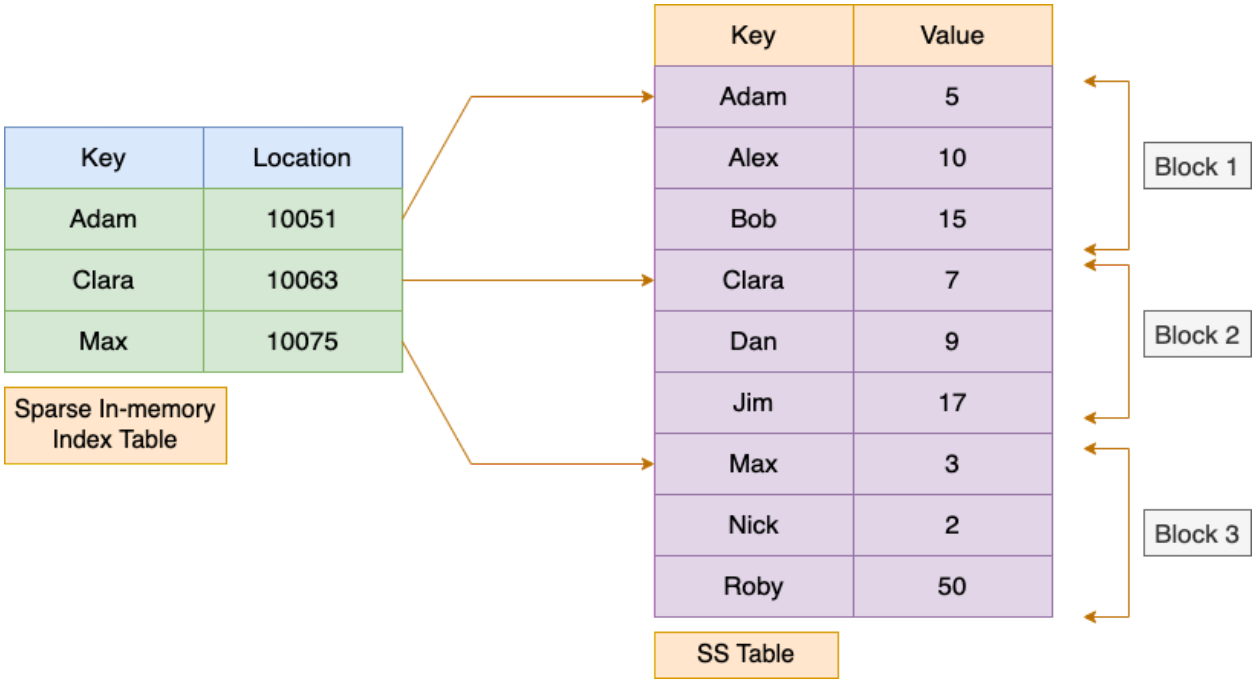
Maintaining all the keys in a sorted order in every segment helps in the merging process. It helps generate a merged sorted-ordered segment in linear time. This is a big advantage over the Hash-based data structures.

Optimizing Reads with SS Tables

We have already optimized the Writes but we didn't talk much about the Reads. When a Read is performed over the database then we look for the **Key** in the **Memtable** (memory) first. Once the Key is not found in the Memtable then it is looked in the **SS Table** (on the disk).

We are not required to look into every key inside the SS Table to read the value of the searched key. Instead we can maintain a **sparse in-memory index** table to optimize this. We can store some keys with their memory location in this table. Say for every **1000** key-value pair block, we can store the first key of that block along with its memory address in the in-memory sparse index table.

Now when the read request comes up, we can precisely determine the block in which that key can be found by looking into the sparse in-memory index table. After that we can scan all the entries of that block to return the value of the key. Since each block contains a small amount of key-value pairs, it is easy to scan the entire block until we find the key we are searching for.



Since we are keeping only the selected keys in the in-memory index, they all can be fit into the Memory. This helps in optimizing our Database Read requests.

We can still ask why we are not using **Binary Search** to look for a key in the SS Table block, since it already maintains the entry sorted by keys. Generally the values do not have a fixed size and hence it is difficult to tell where one record ends and the next one starts. Hence we naively perform a Linear-Search over the block to look for our **key**.

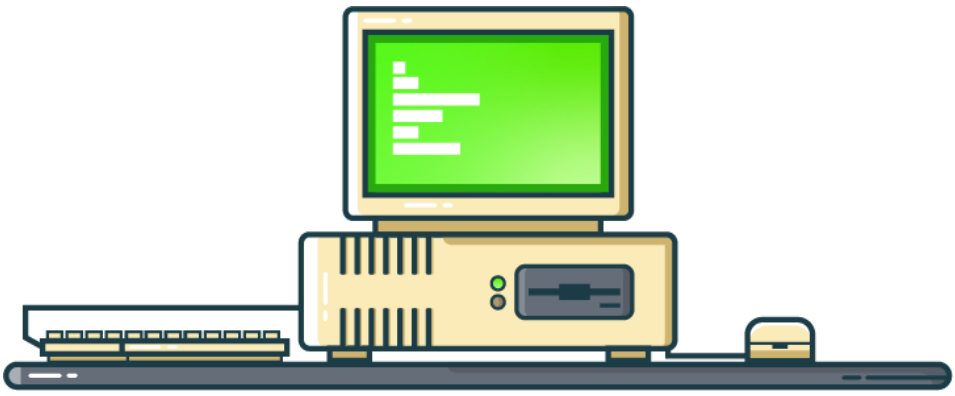
Summary

We discussed **Log-Structured Merge-Trees** (**LSM** Trees) in detail along with **Memtables** and **Sorted String Tables** (**SS Tables**) and how they all power the data storage unit of the Database. We also looked around multiple optimisation techniques performed by the system to achieve efficient Reads and Writes on the database.

Chapter 3

Data Structure 3: B-Trees

we will discuss **B-Trees** which is a data-structure widely used as an indexing structure in several Databases. We will look around the structure, operations and fault tolerance involved in B-Trees. We will also compute the amount of data stored by a B-Tree with a given configuration.



Introduction

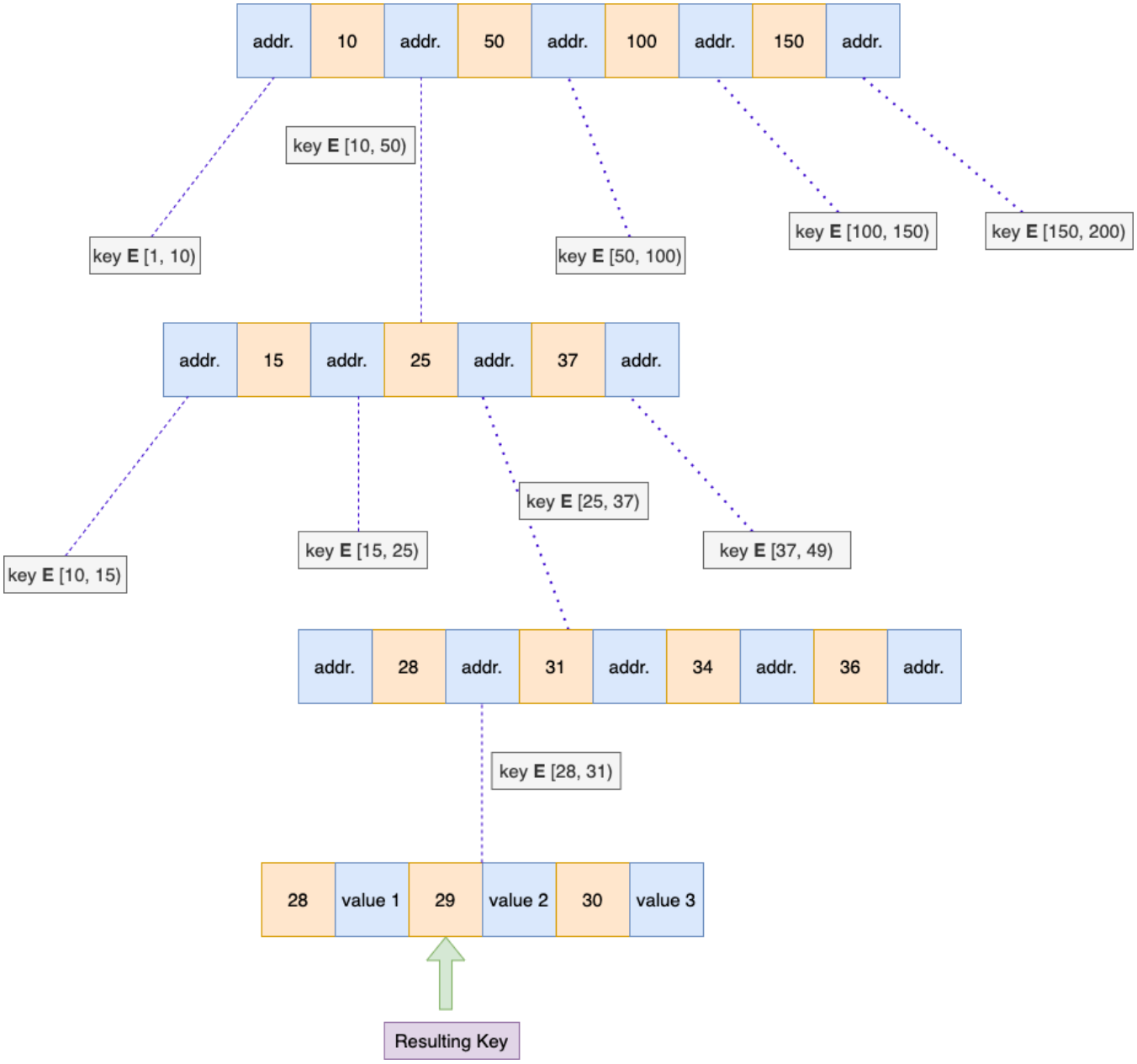
B-Trees are yet another data-structure which is widely used to implement the indexing structure in most of the relational databases and also in many non-relational databases as well.

We looked around **LSM Trees** and **SS Tables** in one of our previous chapters. They are also widely accepted data structures used as an indexing structure in databases. Like SS Tables, B-Trees also keep the key-value pairs sorted by their **Keys** which helps in efficient **lookups** and **range queries**.

B-Trees break the database into fixed size blocks where each block can be of size around **4 Kilobytes**. These blocks are known as **Pages**. Every page can be identified using an address or location similar to the pointers.

Structure of a B-Tree

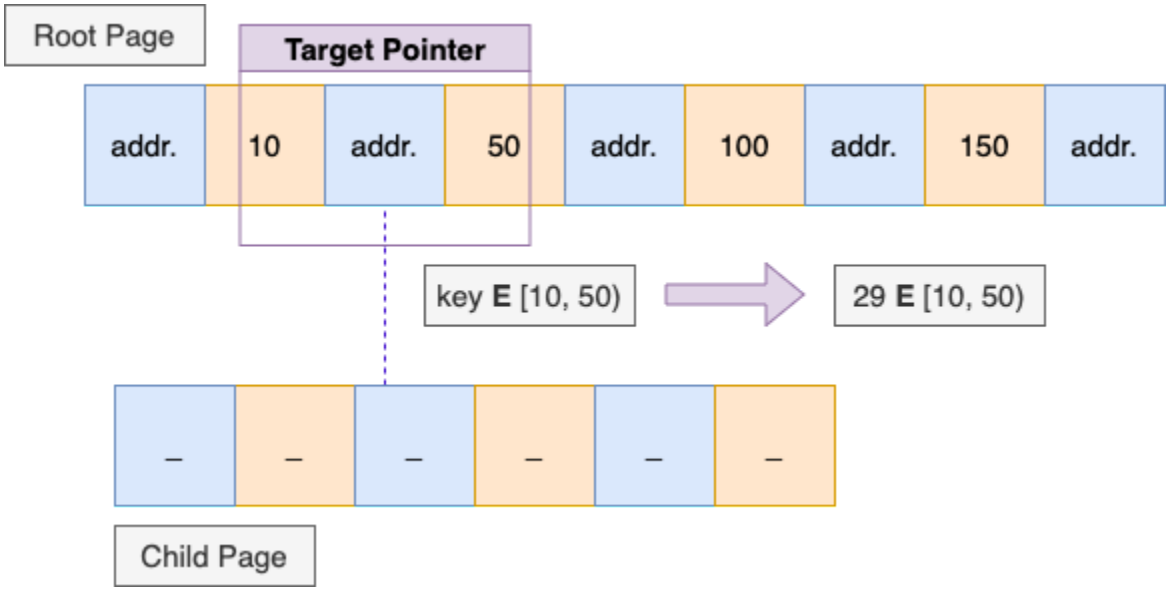
Let's understand the B-Tree by looking at its structure.



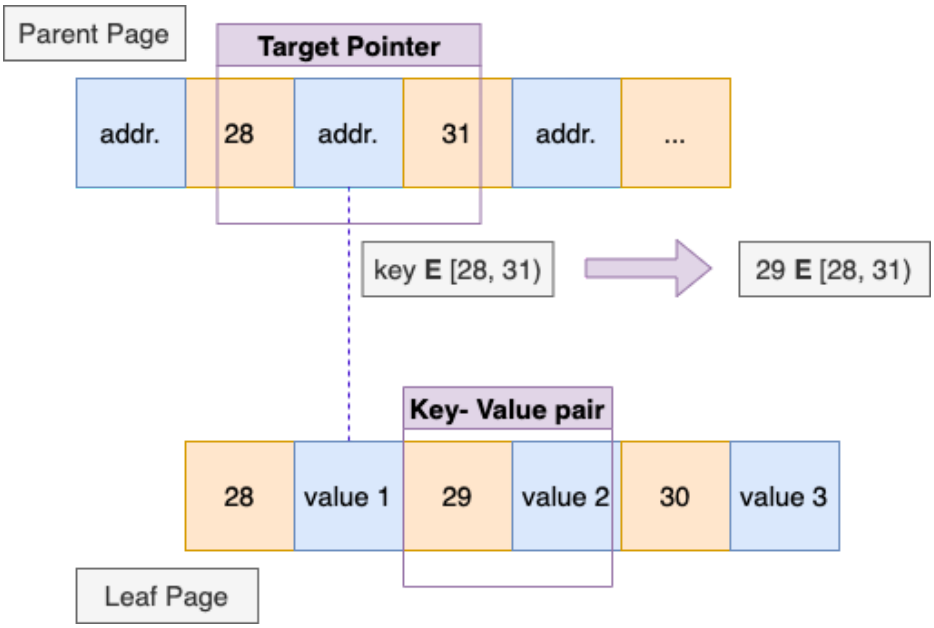
In the above structure of the B-Tree there is a root Page which holds multiple ranges of values along with a reference address pointing to another Page holding the value/range of values addressed by the **Parent** node.

We search for an item with the **key = 29** in the above defined B-Tree structure.

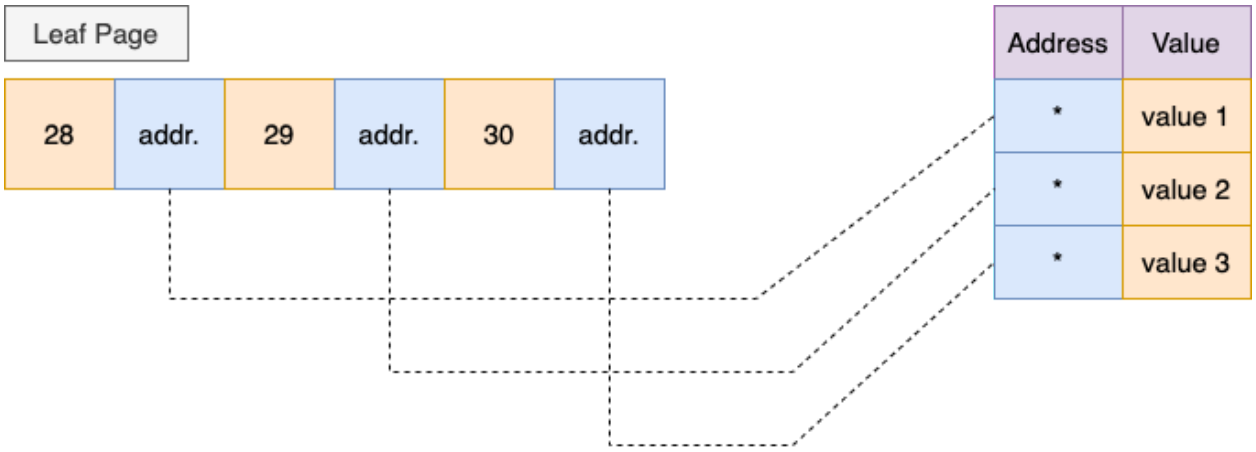
Step 1: We looked into the Page designated as the root node. Since the keys are sorted, we can easily find the node which points to a child page which might have our key.



Step 2: Similarly we kept traversing every page depth wise. Until we came across the leaf page.

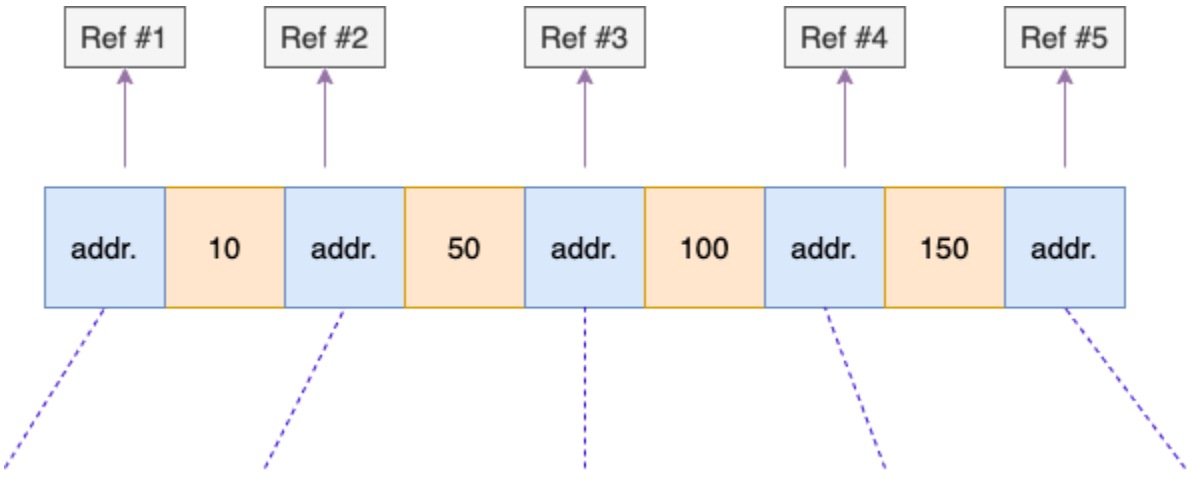


Sometimes the Leaf Page can have reference to their values instead of actually storing the values in place. In that case, we will need to perform one extra look-up to fetch the value for the look-up Key.



Branching Factor: The number of references to the Child Pages present in one Page of the B-Tree is known as its **Branching Factor**.

In our previous example the B-Tree had a branching factor of **5**.

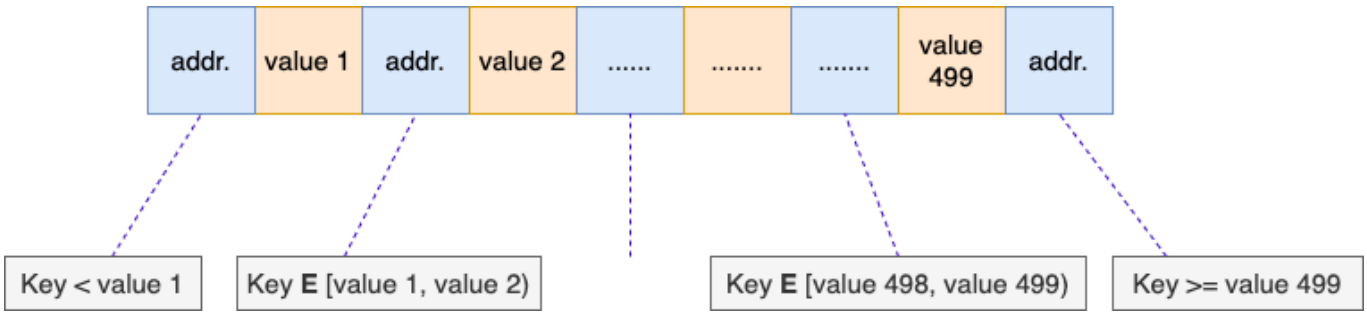


B-Trees are always balanced, hence a B-Tree having **N** nodes will always have a depth of **O(logN)**. Most of the databases can fit into a B-Tree with **three** or **four** levels deep, hence we don't need to hop through so many Page references to find the result.

Estimating the size of data stored by a B-Tree

A B-Tree having a branching factor of **500** with **4 Kilobytes** size pages and **4 levels** deep can store up to **256 Terabytes** of data.

Let's understand the previous statement. Now suppose the root page of the B-Tree with branching factor 500 will look like this.



At every level the parent Page will point to **500** different child Pages. It's safe to say that after **4 levels** the number of Pages in a B-Tree will be equivalent to the following number.

Number of Pages = (500)^4 pages
= 5^4 * 10^8 pages
= 625 * 10^8 pages

Since every Page has a size of 4 kilobytes, hence the total data stored by the B-Tree will be equivalent to the following.

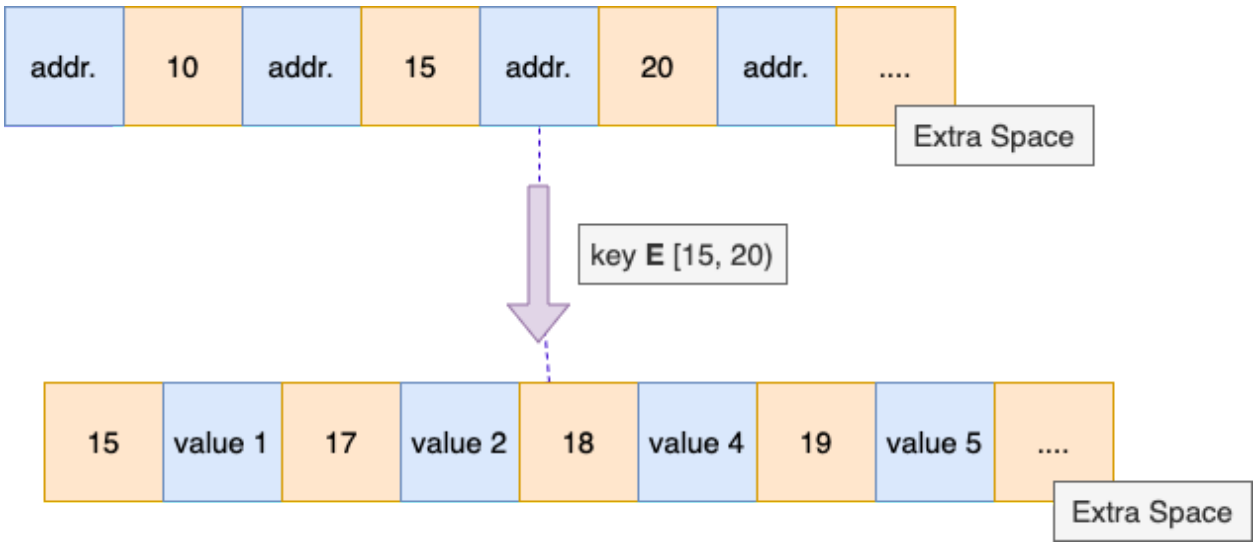
Capacity of the B-Tree = (**Number of Pages**) x (**Size of Page**)
= 625 * 10^8 * 4 Kilobytes
= 2500 * 10^8 Kilobytes
= 250 * 10^9 Kilobytes
= 250 * 10^6 Megabytes
= 250 * 10^3 Gigabytes
= 250 Terabytes ~ **256 Terabytes**

The above computation proves that a B-Tree having a Page size of **4 Kilobytes** with a Branching factor of **500** and **4 Levels** can store approx **256 Terabytes** of data.

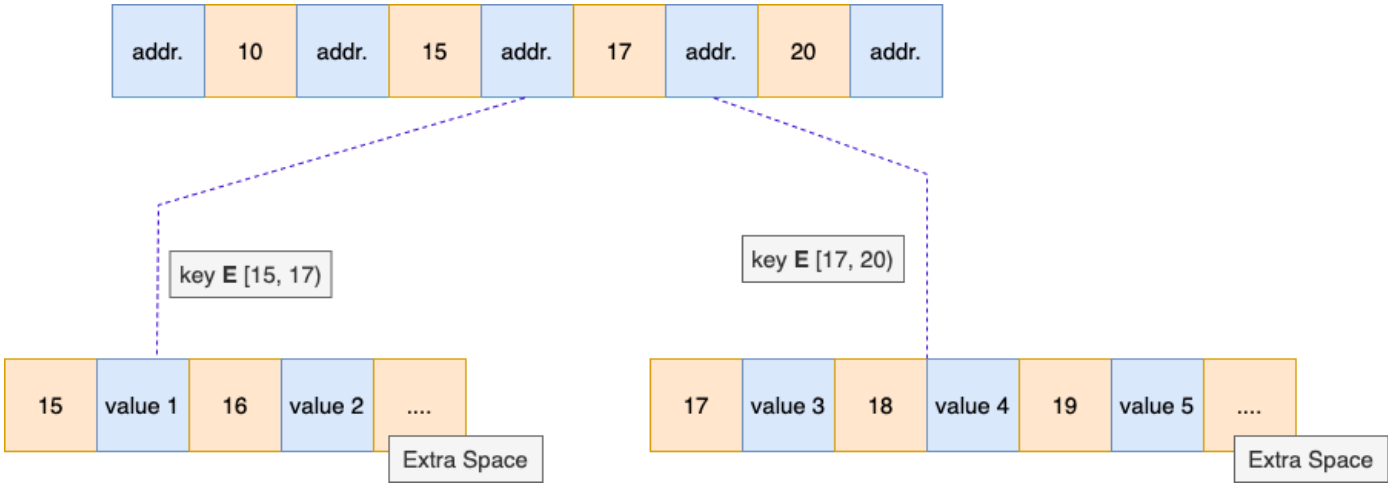
Growing B-Tree by Splitting

If we want to add a new key to the B-Tree, then we need to find the leaf Page responsible for storing the range of that Key. If that page is full then we need to split it into two half-full pages and the parent page is updated to accommodate the new sub-division.

Let's add a **Key = 16** into the existing B-Tree structure.



After adding **Key = 16**, the resulting B-Tree structure looks like:



Fault Tolerance in B-Trees

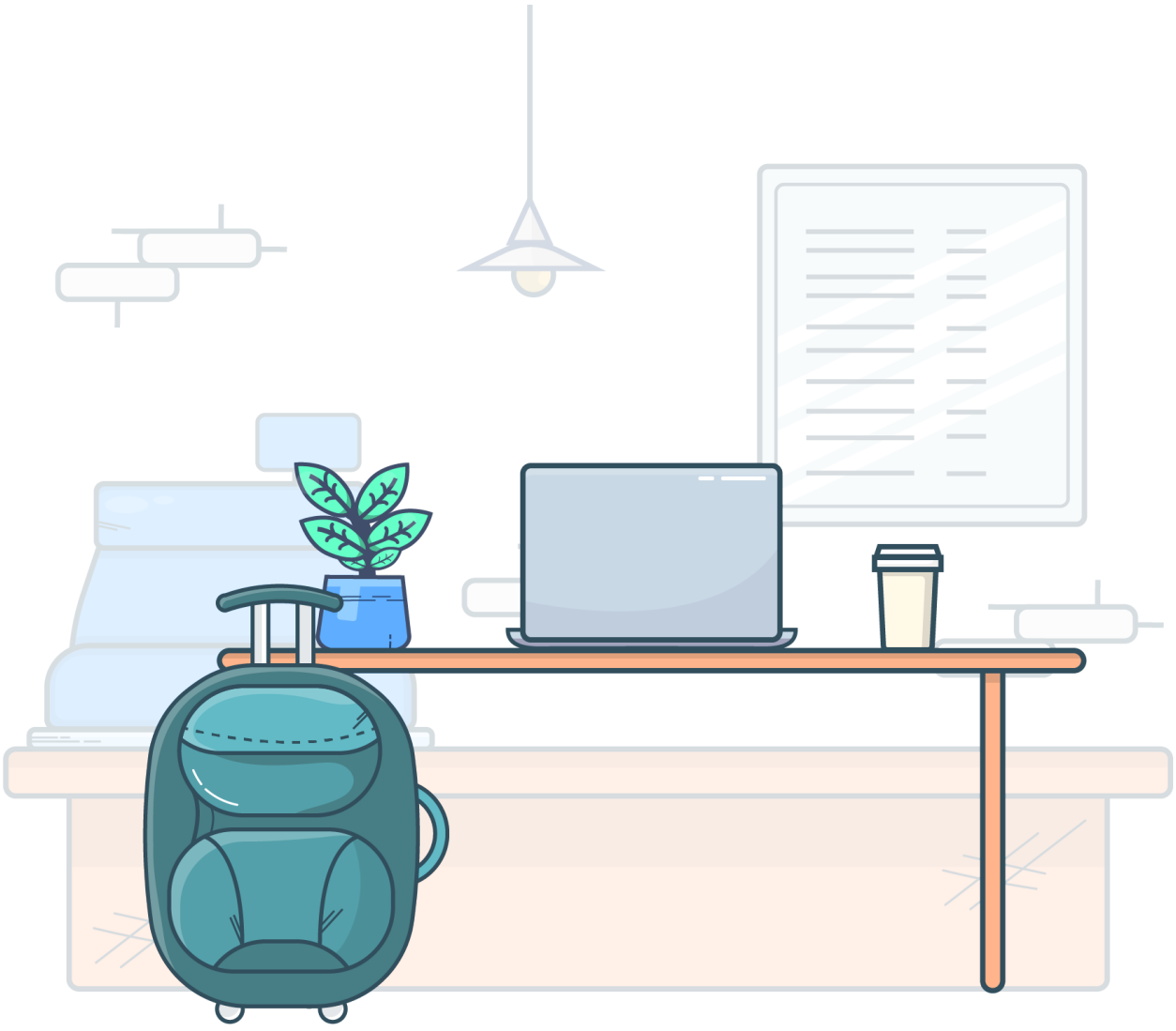
In our previous section, we discussed the case of addition of Key into a Page which is completely filled. In that case we split the leaf Page into two pages and also overwrote the parent page. The series of operations can be risky because if the database crashes in between then our data can be left corrupted.

To recover from the above failure situation, B-Trees uses an additional data-structure on disk which is known as a **Write-Ahead Log (WAL)**. This is an append-only log file which stores all the updates performed on a B-Tree by the order of their timestamp, before these operations can actually be performed on the tree.

Hence, in case of a crash the database can use this log to restore its data and return to a consistent state.

Summary

We looked around one of the most widely used data-structure **B-Trees** as an indexing structure in the databases. We also discussed the structure of a B-Tree and how to estimate the overall size of the B-Tree data structure.



Thank You! ❤️

Hope this handbook helped you in clearing out the concepts of Datastores and Data-structures behind the Databases

Follow me on Linkedin
Do subscribe to my engineering newsletter “Systems That Scale”