Chapter 1

Data Storage

"*Proper storage is about creating a home for something so that minimal effort is required to find it and put it away.*" - Geralin Thomas

In the last chapter, we discussed Database Management System and its components on a higher level. In this chapter, let us discuss briefly about various mediums via which data is stored in databases internally and how efficiently we can access it. We will be first discussing about physical implementation of database systems.

1.1. Physical Storage Mediums

Computerized Databases are stores as files of records on a storage medium, which can further facilitate the retrieval, update, and processing of data as and when required. Data storage in computer systems comes in various forms, and each type of storage medium is classified based on several key factors, including the speed of data access, the cost per unit of data, and the reliability of the medium. These factors are crucial for determining which type of storage is best suited for a particular use case or application.

Faster access times are essential for applications that require quick retrieval and processing of data. Whereas the cost per unit of data is an important consideration, especially for organizations that need to manage large volumes of data. The reliability of a storage medium is crucial to ensure data integrity and availability. Data volatility is important consideration as we need to safeguard data in times of system failures. Based upon these factors, computerized storage mediums are divided into three main categories as show below:

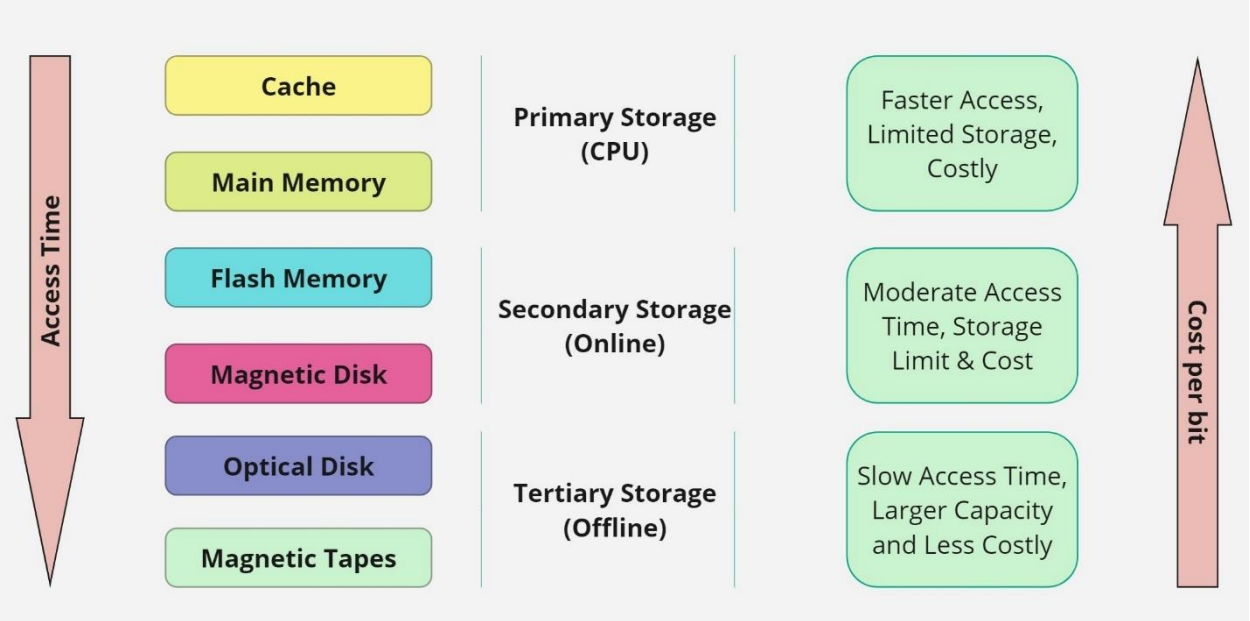

Figure 8-0-1 Physical Storage Mediums

1.1.1. Primary Storage

Imagine primary storage as the fast and small desk right in front of us while we work. This desk (RAM/Cache) allows us to quickly access the things we are actively using, like the current page of a

book. **Cache** memory is relatively small and is manged by CPU to speed up execution of program instructions using techniques such as prefetching and pipelining.

The data to be operated on is stored in **main memory or DRAM (dynamic RAM)**, which provides the main work area for the CPU for keeping program instructions and data. It may contain tens of gigabytes of data, even enterprise databases can be entirely stored here, however, it is volatile, meaning in case of power failure or system crash, data will be gone!

1.1.2. Secondary Storage

We can think of secondary storage as a bookshelf or filing cabinet. It is not as fast as primary storage, yet it can store a lot more items, like all the books or files. This is where our important documents, pictures, and software are kept. Even if the computer turns off, the data on secondary storage stays safe. Magnetic disks (hard drives) are like bookshelves, and flash memory (like USB drives) is like a digital filing cabinet. Magnetic disks provide the bulk of secondary storage for modern computer systems.

Flash memory have been widely used a medium of data storage in phones, camera, etc. Even in personal computers, flash memory is replacing magnetic disks and is referred as solid-state disk (SSD). It uses flash memory internally to store data but provides an interface similar to a magnetic disk, allowing data to be stored or retrieved in units of a block.

1.1.3. Tertiary Storage

Tertiary storage is like an attic or basement. This is where we keep things we don't need frequently yet need to store for a really long time. Imagine it using as a store to keep old family photo albums, holiday decorations, and other stuff we don't access regularly. In the computer world, this is analogues to things like CDs, DVDs, and magnetic tapes. They can store a lot of data and don't cost too much, however they come with the trade-off of access time.

To access data stored on magnetic disk, the system must first move the data from disk to main memory (personal computer), from where they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

Optical storage options like CD/DVDs are capable of storing any type of digital data, including backups of database contents, however, they are not suitable for storing active database data since the time required to access a given piece of data can be quite long compared to the time taken by a magnetic disk.

So, in a computer, primary storage is like one's working desk, secondary storage is like a bookshelf or filing cabinet, and tertiary storage is like your attic or basement for things you don't use often but still want to keep. Each has its own role in managing and accessing data with their own trade-offs. In general, Magnetic tapes are used as a storage medium for backing up databases because storage on tape costs much less than storage on disk. Their capacities have been growing steadily in recent years; however, the storage requirements of large enterprise applications have outgrown rate of disk capacities.

Alternatively, in recent years, SSD storage sizes have been growing rapidly, and their cost has come down significantly, which has made them a main competitor to magnetic disks due to their superior performance. It has become the preferred choice for enterprise data and are being used as an intermediate layer between main memory and secondary rotating storage in the form of magnetic disks.

1.1.4. RAID

In fact, the data-storage requirements of some applications such as multimedia applications have grown so fast that large number of disks are required to store the data. And this is where a relatively newer technology called **RAID** is being leveraged by organizations for improving the rate at which data can be read or written.

RAID originally stood for **redundant arrays of inexpensive disks**, however, in recent years, the **I** in **RAID** is said to stand for **independent**. The idea is to improve performance and reliability and even out the widely different rates of performance improvement of disks against those in memory and microprocessors. An argument can be made that RAM capacities have quadrupled every two to three years but in contrast disk access time and transfer rate have not made significant improvement.

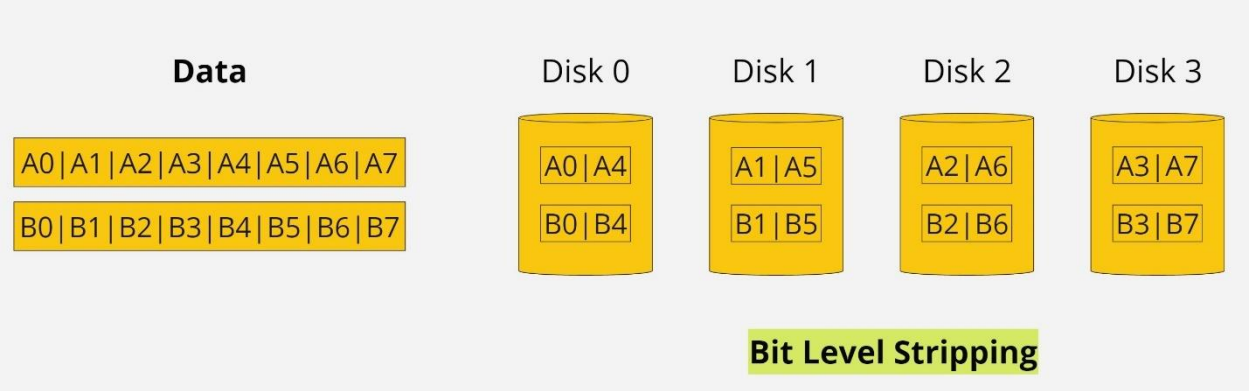

Figure 8-0-2 Bit Level Stripping

RAID relies heavily of the technique of parallelization where a large array of small independent disks, acting as a single higher performance logical disk, are operated in parallel. Concept of **data-stripping** is used to distribute data transparently over multiple disks to make them appear as a single large, fast disk. Each disk in the array holds a portion of each data block. This allows for parallel read and write operations, improving overall data transfer rates and I/O performance. This also provides a window to improve the reliability parameter as redundant information can be stored on multiple disks and failure of one disk does not account for loss of data or distribute the data uniformly across all disks resulting in better load balancing. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired.

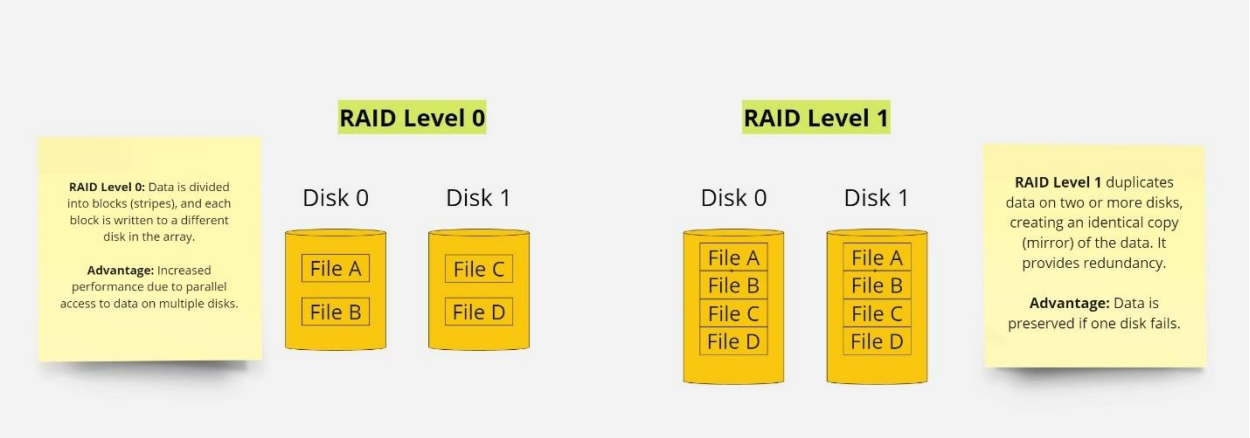

Figure 8-0-3 RAID Level 0 & 1

There are different RAID levels that use data striping, and each has its own approach to striping data across disks. Diagram above shows RAID Level 0 and 1. RAID Level 5 stripes data at the block level across multiple disks and includes distributed parity information for fault tolerance across all the disks. It gives a good balance between performance and redundancy. Each disk in the array stores both data and parity information.

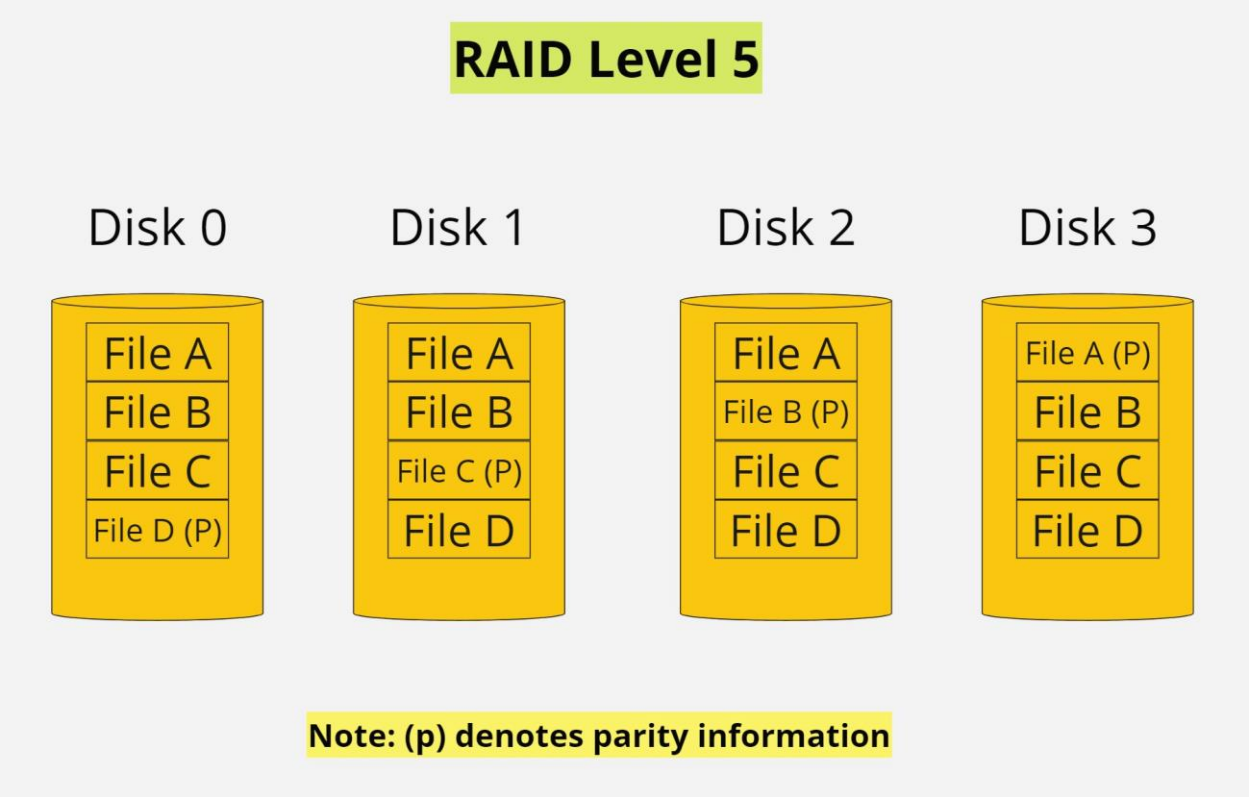

Figure 8-0-4 RAID Level 5

In the context of RAID, parity is a calculated value that represents the XOR (exclusive OR) result of a specific set of data bits. The purpose of parity is to detect errors and, in some RAID levels, to reconstruct data in case of disk failure. The result is stored as the parity information on a designated drive or distributed across all drives, depending on the RAID level.

RAID 5 can tolerate the failure of a single drive without losing data. If one drive fails, the missing data can be reconstructed using the parity information from the remaining drives. It offers good read performance because data can be read in parallel from multiple drives. However, write performance can be slower due to the need to calculate and update parity information for each write operation. RAID 5 is more space-efficient than mirroring-based RAID levels (e.g., RAID 1) because it does not require a complete duplication of data. It is commonly used in scenarios where a balance of performance and data protection is required, such as file servers, application servers, and database servers. However, it is not recommended for high-write environments or with very large capacity drives, as the time to rebuild the array after a drive failure increases with drive size.

Originally, people used RAID because it was cheaper to buy lots of small drives than one big one. But today, technology has changed, and larger drives are cheaper per unit of data storage. So now, we use RAID for speed and safety, not just because it is cheaper. Managing a bunch of smaller drives working together is often easier than dealing with a single large drive. It is like managing a team of people who can help each other out. In simple terms, RAID is like a team of smaller hard drives working together to store your data faster, keep it safe, and make it easier to manage.

1.1.5. Storage Area Networks

With the rapid growth of online businesses, streaming platforms, and complex software systems that manage all sorts of information, the need for storing data has shot up. It is like needing more and more shelves to store all our stuff because we have got so much data stacked. Storing and managing all this data is getting really expensive. In some cases, it is even more expensive to manage the data storage part than the actual computers that use the data. Many organizations use RAID systems to store data

efficiently, but they face a problem: these systems are tied to specific servers and can't easily be shared. It is like having a bookshelf that is permanently attached to one room; we can't move it around to where we need it.

To solve these issues, organizations have adopted a concept called a **Storage Area Network (SAN)**. Think of it like a super-fast network where entire data storage is connected. It is like having books on wheel trolley and connected to a high-speed track so we can easily move them to the room where we need them.

In today's world, where the internet plays a massive role in how organizations operate, there is a need to shift away from rigid and fixed data centres to a more adaptable and flexible infrastructure. This change is necessary because the cost of handling all the data is increasing so quickly that, in many cases, it is actually more expensive to manage the storage of data connected to servers than it is to buy the servers themselves.

Many companies have become providers of Storage Area Networks (SANs) and they have their own unique ways of setting up these networks. In fact, companies offer their own versions of Storage Area Networks (SANs) with each offering having its unique way of setting up these networks. These SANs are designed to put data storage systems far away from the servers. This flexibility means we can set up our systems in more diverse ways. If we already have applications that manage our data storage, we can use these in the SAN setup. They can work with the SAN using a special kind of network called **Fibre Channel**. This network makes the old way of connecting storage devices (using SCSI) compatible with the new SAN setup. So, our SAN-connected devices can act like the familiar SCSI devices.

Alternatively, we can also use a Fibre Channel switch to connect multiple storage systems (like RAID setups and tape libraries) to your servers. It is similar to having a central hub that connects various stores to our house. Another option is using Fibre Channel hubs and switches to connect servers and storage systems in different arrangements. It is like having a more complex road system with multiple intersections and pathways. Idea is to start with simpler setups and gradually make them more complex by adding more servers and storage devices as needed.

The benefits of SAN set up is centred around flexibility. We can connect many servers and storage devices in various ways using hubs and switches. It is like having a versatile road network that connects different places. We can place a server up to 10 kilometres away from a storage system using the right kind of fibre optic cables. New devices and servers can be added without causing disruptions. Data can be quickly copied to multiple storage systems. This is helpful for making sure our data is always available and for keeping backups in case of disasters. It is like making instant copies of our important documents both at home and in a safe deposit box at the bank.

Although, Storage Area Networks (SANs) are becoming more and more popular, they come with their fair share of challenges cantered around compatibility. If we have storage equipment from various companies, making them work together in a single SAN can be tricky. The standards for both the software and hardware used in storage are constantly evolving. So, keeping our SAN up to date and compatible with the latest technology can be a bit of a headache.

Despite these challenges, many big companies are looking at SANs as a viable option for storing their important databases. They see the potential benefits, even though they have to deal with these issues.

1.1.6. Network-Attached Storage

Network Attached Storage (NAS) is a way to share and manage data over a network, but it appears more like a networked file system. The NAS devices are like super-sized hard drives that you can connect to a network. They are helpful because they can provide a lot of storage space to multiple computers without making them stop working for maintenance or upgrades. This means NAS provides a way to access files and folders, much like how we would use our computer's local storage. NAS uses

networked file system protocols, such as NFS (Network File System) or CIFS (Common Internet File System), to make files and folders available over the network.

We can place NAS devices anywhere on your local network (like in different rooms of your house) and use them to store all sorts of data. We can even group them together in various ways. Each NAS system has a central box (like a small computer) that connects to the network. This box acts as the go-between for the NAS system and the computers on the network. We don't need a monitor, keyboard, or mouse to use it. We can even add one or more hard drives or tape drives to many NAS systems to increase the total storage space. These protocols allow us to access and manage our data as if it were on a regular computer's hard drive, rather than treating it like a big shared disk.

NASs act as an alternative to SANs, as the latter focus more on block-level data access and often appear as large disks that store data but don't necessarily provide the same kind of file management and sharing features that NAS does. Network Attached Storage (NAS) systems aim to be reliable and easy to manage. They come with features like secure login and the ability to send email alerts if something goes wrong. These devices are designed to be scalable, dependable, flexible, and fast. NAS systems are designed to work with a wide range of operating systems (like Windows, UNIX, or NetWare) without needing specific changes on the client side. SANs may require more specific configurations on the client's end. SANs usually create their own private network, often called a LAN, where all the storage devices and servers connect. In contrast, NAS devices are directly connected to the existing public network. In simple terms, NAS is like a user-friendly and versatile file storage system that is easy to manage and works with different devices and operating systems (more compatibility than SANs, reference to drawback of SANs).

1.1.7. Object-Based Storage

In recent years, there have been significant changes in how data storage works, driven by the rapid growth of cloud computing, distributed database and analytics systems, and data-intensive web applications. These changes have led to a transformation in the way enterprise storage infrastructure is designed.

Traditionally, storage systems used to be hardware-centric and focused on managing files stored in blocks. However, the latest trend in storage architecture is known as **object-based storage**. In object-based storage, data is organized as objects, not files made up of blocks. These objects come with metadata, which contains important information for managing them. Each object is uniquely identified with a global ID, making it easy to find and access.

The idea of object storage started with research projects at institutions like Carnegie Mellon University (CMU) and the Oceanstore system at the University of California, Berkeley. These projects aimed to create a global infrastructure that allows continuous access to data from various trusted and untrusted servers.

With object storage, we don't need to worry about lower-level storage tasks like managing storage capacity or deciding which RAID architecture to use for data protection. Object storage simplifies data management by treating data as self-contained objects, making it easier to organize and access large volumes of information.

Object storage provides a high level of flexibility by allowing applications to directly control objects and making objects easily accessible across a wide network. It supports replication and distribution of these objects. Object storage is particularly well-suited for handling large amounts of unstructured data, such as web pages, images, audio/video files, and other types of data that don't neatly fit into traditional file structures.

Object-based storage was introduced as part of the SCSI protocol but didn't become a commercial product until Seagate adopted it in its Kinetic Open Storage Platform. Today, it is widely used by

technology giants like Google, Facebook, Spotify, and Dropbox for storing vast amounts of data. Many cloud services, including Google Cloud, Amazon AWS, and Microsoft Azure, rely on object storage to store files, relations, and messages as objects.

Some popular object storage products include Hitachi's HCP, EMC's Atmos, and Scality's RING. OpenStack Swift is an open-source project that simplifies object storage by allowing users to retrieve and store objects using simple HTTP GET and PUT requests. It is cost-effective, fault-resistant, takes advantage of geographic redundancy, and scales well for managing large numbers of objects.

However, object storage may not be the best choice for transaction-intensive systems that require concurrent processing. So, it might not be considered suitable for mainstream enterprise-level database applications where high-throughput transaction processing is crucial.

1.2. Storage Structures

1.2.1. File Storage

Persistent data, which are data that remain even when the computer is turned off, are typically stored on non-volatile storage devices such as magnetic disks or solid-state drives (SSDs). These storage devices are organized into blocks, meaning data is read from or written to them in fixed-size units called blocks.

Databases, on the other hand, deal with records. Records are usually much smaller than these blocks, although they may contain attributes that are quite large in some cases. To manage this difference in size, most databases use the operating system's files as an intermediate layer for storing records. These files abstract away some of the underlying details related to blocks. Each file is structured in a way that it is like a sequence of records. These records are then mapped onto blocks of data on the disk.

However, even though databases use these files to store records, they still need to be aware of the block structure for efficient data access and to support recovery from system failures. So, it is crucial for databases to understand how individual records are stored in these files while taking the block structure into account. This ensures that data can be efficiently managed, accessed, and safeguarded, even when the underlying storage operates in blocks.

Now, files are a fundamental concept in operating systems. They serve as containers to store data. So, we assume that there is a file system in place that takes care of managing these files on the storage devices. It is important to figure out how to represent the logical data models (the way data is organized conceptually) in terms of these files. Essentially, it is about translating how the database is supposed to work into how the data is actually stored on the disk through files. This translation is a critical aspect of database management to ensure data is organized efficiently and can be accessed as needed.

Each file is further divided into fixed-size storage units called "blocks". These blocks are the fundamental units for allocating storage space and transferring data. Most databases default to block sizes of 4 to 8 kilobytes, although some databases allow you to specify the block size when creating a database instance. In certain cases, larger block sizes can be advantageous for specific database applications.

Now, each block can contain multiple records, but which records are stored within a block depends on the specific way the data is physically organized in the database. The choice of how data is organized, and which records are placed in a block is a fundamental aspect of database design and directly impacts how efficiently data can be accessed and managed.

Let us look at the structure of a magnetic disk:

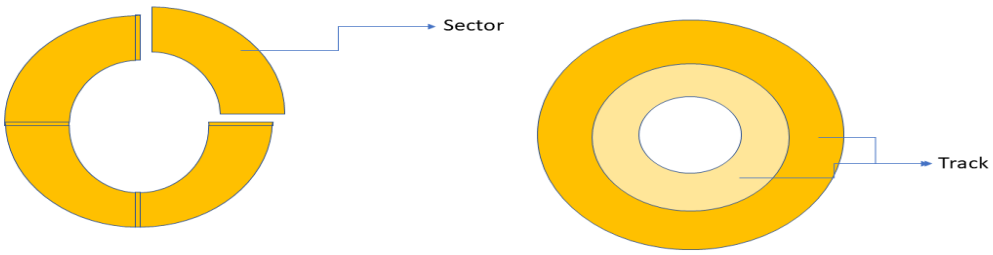

Figure 8-0-5 Magnetic Disk Structure

A disk surface is logically divided into concentric circles called **tracks**, and each track is further subdivided into **sectors**. When data is read from or written to the disk, it's done at the sector level. The division of a track into sectors is hard-coded on the disk surface and cannot be changed. Sectors are typically fixed in size, commonly 512 bytes or 4 KB.

When a disk is formatted or initialized, the operating system sets up the structure of the disk, including the division of the track into equal-sized portions called **disk blocks or pages**. These blocks are predetermined in size and remain fixed once the initialization process is completed. The size of these blocks cannot be changed dynamically. Typically, a disk with hard-coded sectors (the smallest addressable unit on a disk) organizes these sectors into larger units called blocks during initialization. The organization involves either combining multiple sectors into a single block or subdividing larger sectors into smaller blocks based on the predefined block size. Data is transferred between disk and main memory in units of blocks.

Each block is separated by interblock gaps, which are spaces reserved between blocks on the track. These gaps contain specialized control information that is written during the disk initialization process. This control information includes specific codes or markers that identify the beginning and end of each block.

This block size is significant for various file systems and disk management operations, as it determines how data is organized, stored, and retrieved on the disk. Larger block sizes can be more efficient for handling larger files and reducing overhead, while smaller block sizes might be advantageous for storing smaller files or optimizing space usage. However, there's a trade-off between efficiency and wasted space, as smaller block sizes can lead to increased internal fragmentation (wasted space within blocks). Changing this size later would involve reformatting the disk, which is a process that erases all existing data, making it a critical decision during the initial setup of the disk.

Now, let us consider a sample table to understand how data is organized on the disk in the form of database:

Employee ID	**Name**	**Department**	**Section**	**Address**
E101	Rahul	D-201	A	<……>
E102	Shardul	D-301	B	<……>
…..	…..	…..	…..	……

E200	Naresh	D-202	D	<…….>

Now, let us consider the column wise size break up:

Employee Id	10 bytes
Name	50 bytes
Department	10 bytes
Section	8 bytes
Address	50 bytes
Total	**128 bytes**

This means that each row of the table is of 128 bytes. Say, we have a block size of 512 bytes (default), so we can calculate, how many records a block of disk can store:

$$Records\ per\ block\ =\ (512/128)\ =\ 4;$$

Hence, we can deduce that to store 100 such records, we will utilize 25 blocks of the disk. In other words, we can say, to access entire table, we will have to access 25 blocks.

Now in a relational database, we have tables, and each table consists of rows, which are called tuples, and these tuples often have different sizes because they represent different sets of attributes or fields.

When it comes to storing this relational data on a physical storage device like a disk, there are following two options:

Fixed-Length Records in Separate Files: In fixed-length records, each record (or row) in a database table is allocated a specific, fixed amount of storage space, regardless of the actual amount of data it contains. This ensures that each record takes up the same amount of space in the storage file.

Consider a database of customer information. In a fixed-length record format, each customer's record is, let's say, exactly 200 bytes long. Even if one customer's data only requires 100 bytes (e.g., name and address), their record would still occupy 200 bytes in the file. This approach makes it easy to locate and retrieve records because we always know that, for example, the first customer record starts at byte 0, the second one at byte 200, and so on.

This approach is relatively simple to implement because we always know where a specific record starts and ends in the file. However, it can lead to wasted space if some records are much smaller than the allocated space.

Variable-Length Records in the Same File: An alternative is to structure your files in a way that can accommodate records of different lengths within the same file. This way, you don't allocate a fixed amount of space for each record. Each record contains its own metadata or markers that indicate its start and end, so the system can distinguish one record from another.

Let's stick with the customer database. In a variable-length record format, one customer's data might occupy 100 bytes, and the next customer's data could take up 150 bytes. The records themselves include markers indicating where each record starts and ends. This approach makes better use of storage space as there's no wasted, empty space between records.

This is more efficient in terms of storage utilization because it doesn't waste space and can accommodate varying record sizes, making it more adaptable, but it can be more challenging to implement because we need to manage variable-length records and keep track of where each record starts and ends in the file. Retrieving data can be slower compared to fixed-length records because the system must locate the start and end of each record before processing it.

Storing Large Objects: Databases often need to store data that can be much larger than the typical storage block on a disk. For example, multimedia data like images, audio recordings, or video files can be quite large, ranging from megabytes to gigabytes in size. In SQL databases, you have data types like

BLOB (Binary Large Object) and CLOB (Character Large Object) specifically designed to handle such large data.

To manage these large objects, many databases have an internal restriction where the size of a record is limited to be no larger than the size of a block on the storage device. In other words, a record in the database is usually limited to a certain size, and this size restriction can be smaller than the size of large objects. In such cases, databases logically include large objects within records but store these large objects separately from the other attributes of the record.

To make this work, the database stores a reference or pointer to the large object within the record. This pointer essentially tells the database where to find the actual large object's data. The large objects can be managed in two main ways:

- **File System Storage:** In this approach, large objects are stored as separate files in a file system area that is managed by the database. The database record contains a reference to the location of the large object in the file system.
- **Database-Managed Storage:** In this method, the database itself manages the storage of large objects. Instead of storing them as external files, the database stores large objects as internal structures. The database can use storage mechanisms like B+ tree file organizations to efficiently access different parts of the large object. This means you can efficiently read the entire large object or specific portions of it, as well as insert or delete parts of the object. For example, consider a video file stored as a large object in a database. The record might contain information about the video, while the video data itself is stored separately. The record would include a pointer to the location of the video data. With the use of B+ tree file organization, the database can efficiently access and manage this video data. This approach helps databases handle very large data objects without overwhelming the size of each individual record in the database.

However, accessing large objects directly through database interfaces might not be as efficient as accessing them from a file system. Database systems are optimized for structured data and complex queries, while handling large, unstructured data like multimedia can be less efficient. Databases are often backed up periodically. These backups are known as **database dumps**. When we store large objects in the database, it can significantly increase the size of these database dumps, making them larger and potentially slower to create and restore.

Because of these concerns, many applications opt to store very large objects, such as video data, outside of the database, typically in a file system. In this approach, the application stores a reference to the file, usually in the form of a file path, as an attribute of a record in the database. This way, the database record contains a reference to the location of the file in the file system.

Storing data outside of the database in a file system can lead to potential issues with data integrity. For example, if a file is deleted or moved in the file system, the reference stored in the database might become invalid, resulting in a form of foreign-key constraint violation. This can lead to data inconsistencies. Database systems typically have robust security and authorization controls in place to manage access to data. When data is stored in the file system, these controls are not directly applicable, and access to the files might not be as secure or easily managed.

We need to be aware about the issues when storing data in this fashion as it requires careful handling to ensure data integrity and security.

1.2.1.1. Record Organization in Files

So far, we have seen how records are represented in file structure in databases. Each record represents an entity or an item, and a collection of records forms a relation, which is essentially a table in a

database. In this section, we will discuss about how to organize these records within a file in the database system. This involves deciding on the physical structure and layout of the data on the storage device.

The organization of records in a file is a critical decision in database design, and it impacts the efficiency and performance of data access operations. Different file organization methods can be employed based on the specific needs and characteristics of the data, as well as the anticipated access patterns. Understanding how the data will be accessed is crucial. Are we going to retrieve records in a specific order or based on certain criteria, or will access be random? Size and nature of the data needs to be considered as well. Whether record size is fixed, or does it vary? Are we dealing with large objects like multimedia files? We also need to think about query patterns to be run on the dataset. While we do need to ensure that data organization ensures efficient query run and at the same time adheres to data integrity and consistency while maintaining optimal space utilization.

There are certain file organization methods which are used across industry, but they come with their advantages and trade-offs. The choice of file organization method should align with the specific requirements and characteristics of the database and the expected usage patterns. Let us discuss common file organization methods:

1.2.1.1.1. Heap file organization

It is the simplest file management technique where records are placed in the file without any particular order. There is no inherent logic to how they are stored. Imagine tossing items into a drawer randomly. There is no specific order or arrangement. Each item can be anywhere in the drawer. Similarly, in Heap, records are put at the end of the file as they are added. The records are not sorted or ordered in any way. The next record is saved in the new block after the data block is filled. This new block does not have to be the next one.

Heap organization works well for smaller databases or in scenarios where significant amount of data needs to be inserted in the database at once. However, it does not suit a larger database as record access time will take a toll.

1.2.1.1.2. Sequential File organization

In this technique, ordering of records is maintained at the time of insertion. The record will be entered in the same order as it is inserted into the tables. For deletion and update operations, memory blocks are identified for the records in question and are marked for deletion. A new record is added in their place.

It is a simple technique data is stored with comparatively little effort and avoids usage of expensive storage mechanisms. However, accessing a specific record at once is not possible in this organization; instead, we must proceed in a sequential manner, which is a costly operation.

1.2.1.1.3. Multitable Clustering Organization

In multitable clustering, records from different tables are stored together in the same file or block. This is done to make certain operations, like joining data from different tables, more efficient. Imagine a big bookshelf where books are sorted not only by title but also by genre. All science fiction books are in one section, all mysteries in another, and so on. In a typical relational database, data is organized into tables, with each table representing a specific entity or concept (e.g., customers, orders, products). Each table is usually stored in its own file or block. However, in multitable clustering, records from different tables are stored together in the same file or block.

For example, we have a sample employee table and department table below:

EMP-ID	**EMP Name**	**Address**	**Dep-ID**
E-1	John	London	D-2

E-2	Rahul	India	D-3
E-3	Mark	US	D-2

Dep-ID	Dep Name
D1	IT
D2	HR
D3	Management

A resultant clustered table would look something like:

Dep-ID	Dep-Name	EMP-ID	EMP Name	Address
D2	HR	E1	John	London
D3	Management	E2	Rahul	India
D2	HR	E3	Mark	US

In the above table, the department information is clustered with the corresponding employee records. This organization lowers the cost of searching multiple files for various records. This can significantly reduce the number of disk input/output (I/O) operations, which is a performance bottleneck in many database systems, since the relevant data is collocated in the same file or block, potentially reducing the need for extensive disk I/O during join operations. A cluster key is used to link the tables together. The primary goal of multitable clustering is to co-locate data that is often joined together in queries. For example, if we frequently need to retrieve data from both the “Employee” and “Department” tables, storing them together can make the join operation more efficient.

Note that in a real-world scenario, additional considerations such as indexing, and data distribution would be taken into account for optimal performance. However, this approach has its own limitations. It does not perform well for larger databases, as in scenarios like change in join condition, it will not give appropriate result. In case of update, traversal of files will take time.

1.2.1.1.4. B+ Tree Organization

B+ tree structure is an advanced way of indexed sequential mechanism for storing and managing data in a structured way within the database. We will be discussing about index in detail in next section, and indexes are the place where B+ tree structures are utilized the most. B+-Trees are designed to remain balanced, meaning they have roughly the same number of branches at each level. This balance is maintained during insertions and deletions. Balancing ensures that searches remain efficient, as the system can quickly determine the path to the data you want to access. It is particularly useful when you need efficient and ordered access to records, even when there are frequent insertions, deletions, or updates.

Think of a B+ Tree as a hierarchical structure, somewhat like a family tree but for data records. The tree is organized based on a specific attribute, often called a “search key” or “primary key”. This key is used to determine the position of records in the tree. The index value is generated for each primary key and mapped to the record.

1.2.1.1.4.1. Working of a B+ tree

The order of a B+ tree, represented by the variable ‘d’, determines the maximum number of entries in a node. Specifically, each node (except the root) must have at least ‘d’ entries and at most ‘2d’ entries, making them at least half full. Leaf nodes can end up with fewer than ‘d’ entries after deletions. Entries within each node must be sorted, allowing for efficient search operations. Between each entry in an inner node, there is a pointer to a child node. An inner node can have at most ‘2d + 1’ child pointers, which is also known as the tree’s fanout. After deletions, leaf nodes may end up with fewer than "d" entries, but the overall structure remains consistent with the defined order.

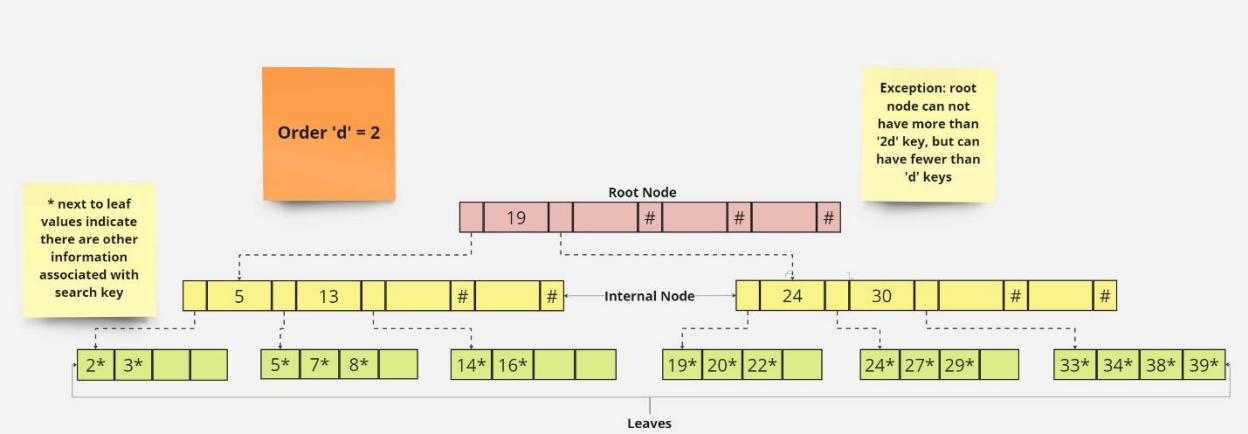

Figure 8-0-6 B+ Tree

The tree in picture has a root node and two internal nodes serve as a pointer to the leaf nodes. Values lesser than the root node are stored in nodes which are left to the root node and greater ones are stored to the right ones. This ordering facilitates the search process, guiding the traversal down the tree based on the keys. Here is how a lookup in B+ tree would look like:

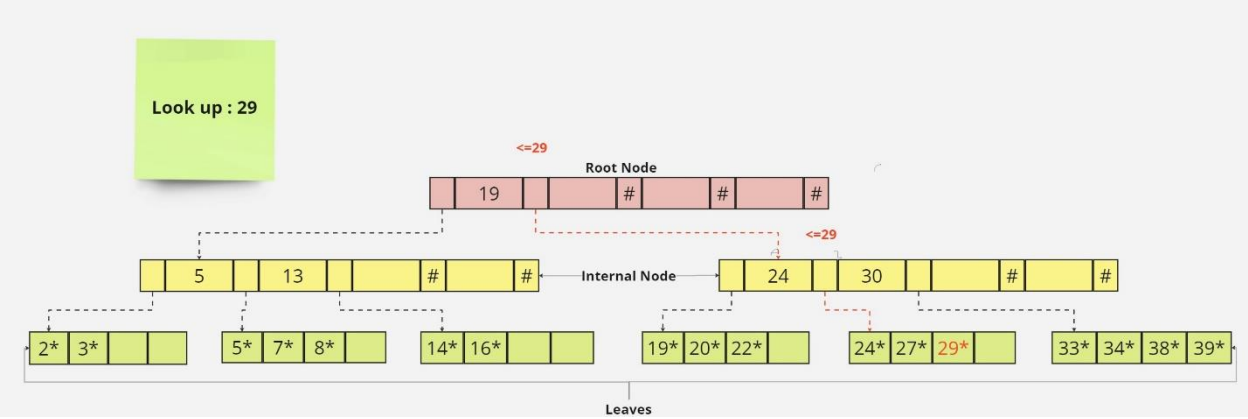

Figure 8-0-7 Look Up in B+ Tree

Now, let us look at a sample insertion in B+ tree and then understand the process and reasoning behind it.

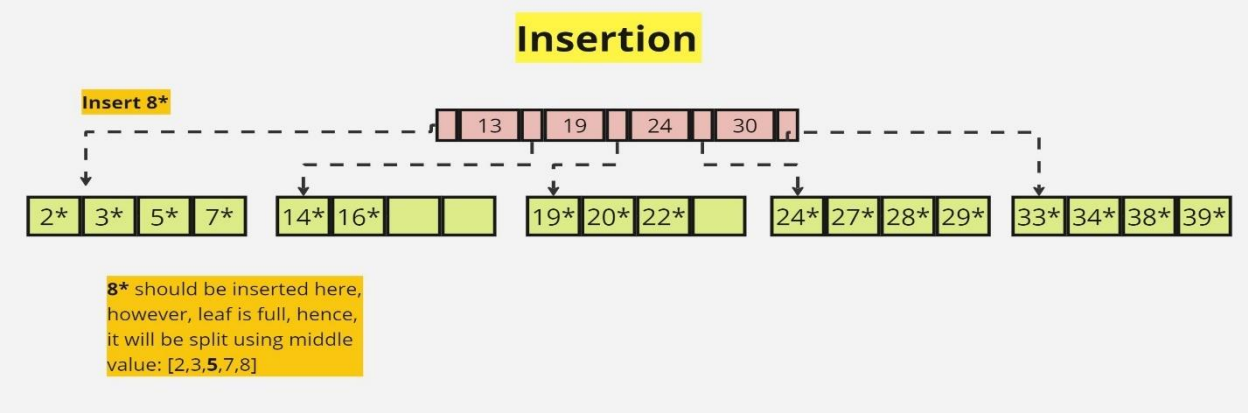

Figure 8-0-8 Insertion in B+ tree: Step 1

When a new record is added to the database, the system determines where it should be placed based on the search key. The system traverses down the tree from the root to the leaf level. At each level, it selects

the appropriate branch to follow based on the search key of the record being inserted. The process continues until the system reaches a leaf node. In the leaf node, the system checks if there is enough space for the new record. If there is sufficient space, the new record is added to the leaf node in the appropriate position to maintain the sorted order.

If the leaf node is full, it may need to be split to accommodate the new record. If the leaf node is full, a split occurs. The existing records are divided into two, and the middle record is promoted to the parent node. The new record is inserted into the appropriate half of the split leaf node.

After inserting the record in the leaf node, the system checks if the parent node needs to be updated. If the parent node is full, a split may occur, and the process of promoting a record to the higher-level repeats. The updates propagate up the tree until the root, potentially causing a split at each level. If the root is split during the insertion process, a new root is created, and the height of the tree increases. Throughout the insertion process, the tree is balanced to ensure that each level has roughly the same number of branches. Balancing helps maintain the efficiency of search operations.

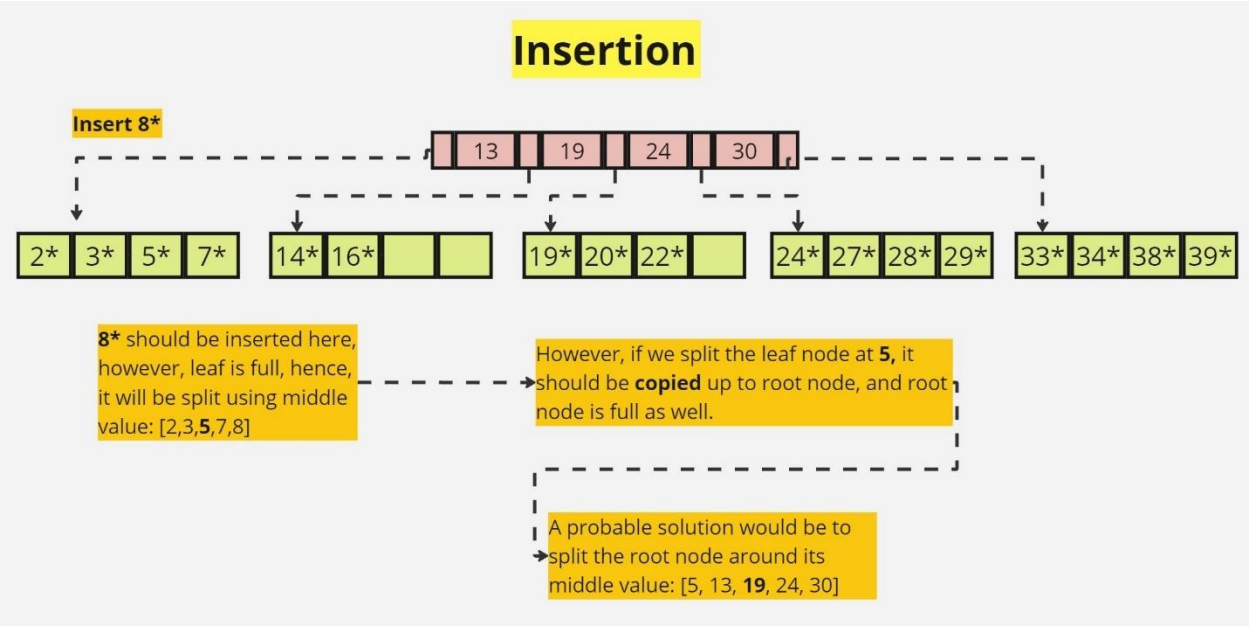

Figure 8-0-9 Insertion in B+ tree: Step 2

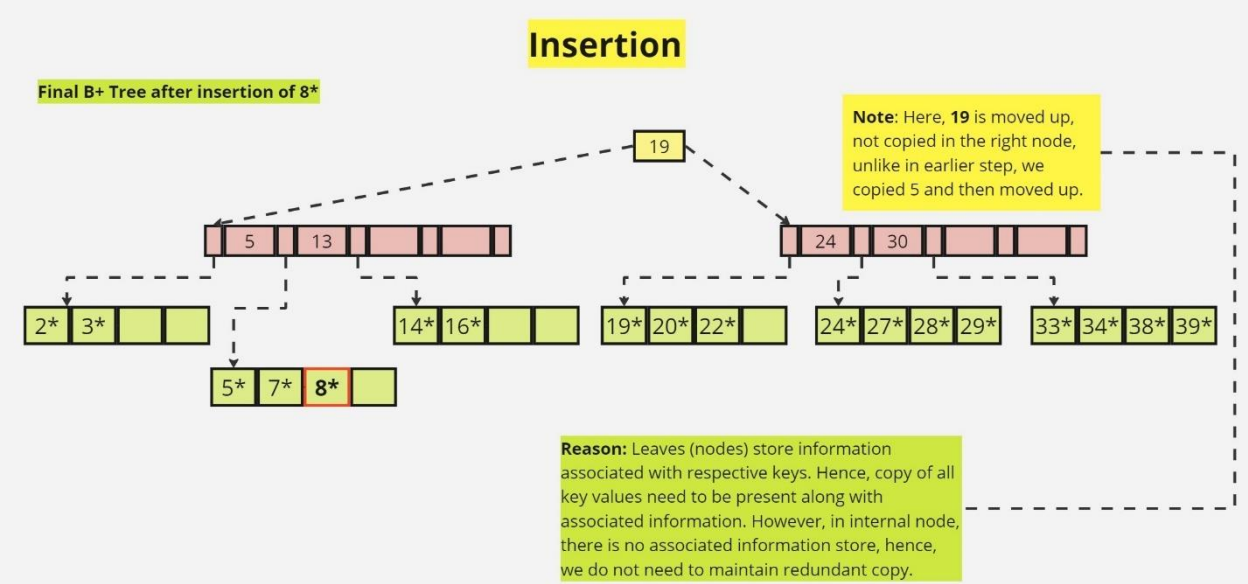

Figure 8-0-10 Insertion in B+ Tree: Step 3

Similarly, to delete a value, just find the appropriate leaf and delete the unwanted value from that leaf.

1.2.1.1.4.2. Storing of records in B+ tree

A record can be stored in B+ tree in three possible ways. We can either contain the records directly on the leaf nodes or maintain a pointer to the records on the leaf nodes. In an advanced alternative, we can store list of pointers to corresponding records in leaf nodes.

B+ Trees are particularly efficient for range queries, where we need to retrieve data within a specific range of key values. It is often used as the data structure behind multilevel index in databases. B+ trees with small nodes fitting within cache line (usually 64 bytes) also seem to provide good performance with in-memory data. It is especially well-suited for scenarios where we need quick and ordered access to data, even in the presence of frequent data changes. The hierarchical structure and balanced design make it a fundamental tool in database systems for optimizing data access.

1.2.1.1.5. Hashing File Organization

Hashing is transformation of a string of characters to a shorted fixed length value that represents original text. A shorter value helps in indexing and faster searches. It is quite popularly used in data structures to verify integrity of data. It is the heart of data structures like Hash Table, and is used to implement search algorithms, bloom filters, fast look up and queries.

In Hashing, a ‘bucket’ is the basic unit of storage that can store one or more records. In memory-based hash indices, a bucket could be a linked list of index entries or records. For disk-based indices, a bucket would be a linked list of disk blocks. In a hash file organization, buckets store either record pointers or actual records.

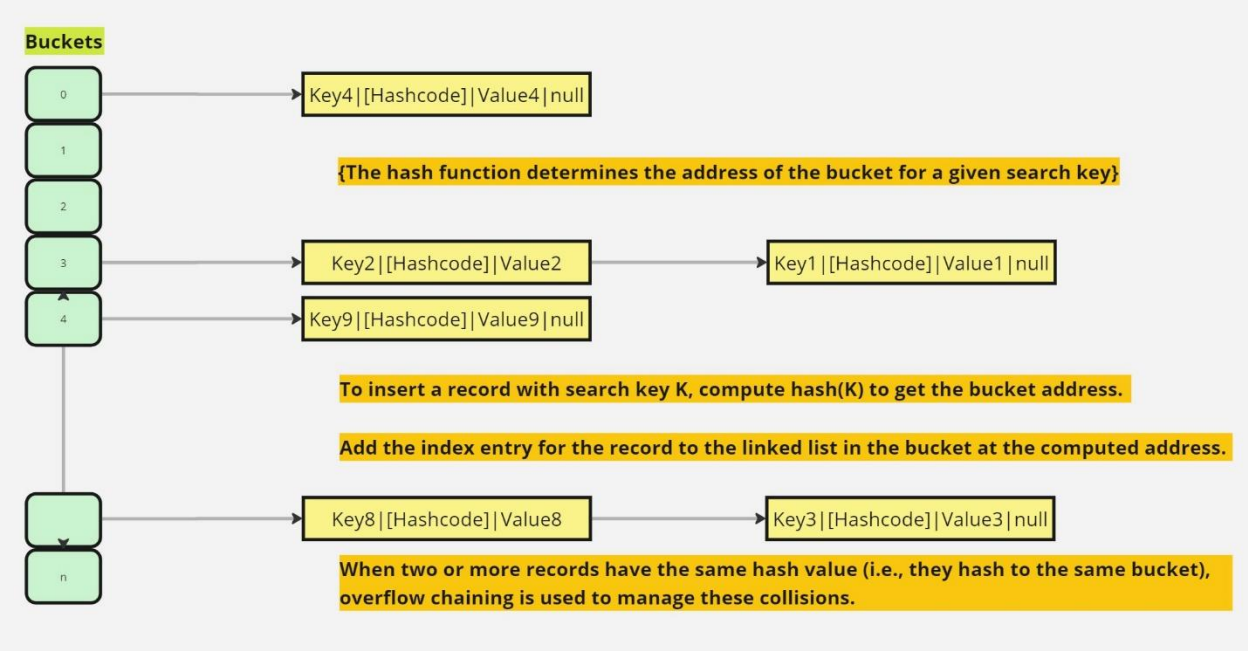

Figure 8-0-11 Data storage representation in HashMap structure

Building on this concept, hashing file organization is a technique used to organize and store records in a file based on the result of a hash function applied to a key. Every time we add or update a new piece of information (like a key-value pair) in a file, we simultaneously update a special map to remember where that information is stored within the file. This process happens whenever we insert new keys or update existing ones. When we need to find a specific value, this map helps us quickly locate where that information is stored in the file. By using this map, we directly know the precise location, so we can go straight to that spot in the file, retrieve the stored data, and read the associated value.

This method is particularly useful for achieving fast retrieval of records when the search key is known. It is often used to build in memory hash indices. However, handling collisions (when two or more records hash to the same bucket or slot in the file) can be a tricky scenario. Hashing file organization is less suitable for range queries or partial key searches since Hash maps don’t store keys in a sequential order; they are scattered across buckets based on hash values. Sequential scanning or iterating through keys within a range isn't straightforward due to non-sequential storage. The hash map doesn’t maintain inherent information about key sequences or ranges. To find all keys within a range, each key needs to be individually looked up using its hash value. Performing individual lookups for each key in a range incurs significant lookup overhead and computational cost. For large ranges, the time and resources required for individual lookups can be impractical and inefficient. Alternatives can be use of B+ Trees, which use data structure like sorted trees which are suitable for maintaining sequential order of keys. Another approach can be to maintain secondary indexes or auxiliary structures alongside the hash map to facilitate range queries efficiently. Or else divide large ranges into smaller, manageable chunks to perform more targeted queries. Data structures like Sorted String Table (SSTable) or LSM trees are quite efficient in overcoming limitations poses by Hashing data structures in maintaining range queries.

1.2.1.2. Data Dictionary Storage

Earlier section, we discussed about how we can represent relation; however, database systems need to maintain information about the relations as well, kind of a knowledge hub, housing essential metadata about its structure and organization. This hub contains comprehensive details about various elements within the database, enabling effective management, querying, and optimization.

A **data dictionary** serves as a repository of metadata that describes the structure, organization, and properties of a database. It holds essential information about the database schema, such as the names of

tables, columns, their data types, constraints, relationships, indexes, and other pertinent details. The storage of this critical metadata varies depending on the database system and its architecture.

In many database management systems (DBMS), the data dictionary is maintained internally within system catalogue tables. These system catalogue tables are structured tables stored within the database itself, containing detailed information about the database schema. Whereas, some databases opt for separate files exclusively dedicated to storing data dictionary information. These files might be indexed and managed by the DBMS to facilitate quick access and efficient management of metadata.

The efficiency and speed of accessing system metadata are crucial for the smooth operation of a database. To ensure rapid access and responsiveness, most databases employ a strategy of preloading system metadata into in-memory data structures during the database start-up phase before any query processing begins. These in-memory structures, like caches or in-memory tables, are optimized for quick access, allowing the system to efficiently retrieve critical metadata during runtime. By having system metadata readily available in memory, the system minimizes latency associated with disk reads, significantly speeding up access to critical information, which means faster query processing and overall system responsiveness, benefiting users and applications.

1.2.1.3. Database Buffer

In recent years, size of main memory has increased significantly, even to the extent that medium sized databases are easily fit in the main memory, however, for enterprise applications, there has a good amount of load on server's main memory and sustaining databases on main memory is not preferred yet. Larger databases are anyways much larger in size than the memory available on the servers. And for the same reasons, disk is still preferred as the storage medium for databases and data must be transferred from disk to main memory for required purposes and updates shall also be written back to the disk.

Now, this process of transfer of data from disk to main memory and back can be resource intensive as well as time consuming, and different approaches are always leveraged to minimize the cost (number of block transfers between the disk and memory). One such approach is the usage of Database Buffer or buffer pool.

The **database buffer** is a fundamental component of a database management system (DBMS), responsible for efficiently managing data between the disk and memory. It is a segment of the available main memory that's specifically reserved for storing copies of disk blocks. These disk blocks contain portions of the database and having them in-memory optimizes data access and speeds up operations. Similar to operating system concepts of virtual memory manager, a **buffer manager** oversees the allocation of memory space within the buffer for storing these disk block copies. It decides which blocks to retain in memory, evicting older or less frequently used blocks to make room for new ones. Although there is always a copy of each block on disk, the version in the buffer might be more recent due to modifications. The buffer manager determines when to write modified blocks back to disk to ensure data persistence and consistency.

The buffer manager handles I/O requests by managing buffer space, fetching data from disk to the buffer or writing modified data back to disk as needed. When a program needs a specific disk block, it sends a request to the buffer manager. The buffer manager checks if the requested block is already present in the buffer (main memory). If the requested block is present in the buffer, the buffer manager provides the requester with the address of that block in main memory. If the block is not in the buffer, the manager proceeds to allocate space in the buffer for the new block. If needed, the manager makes space in the buffer by evicting another block, employing a specific replacement policy such as **Least Recently Used (LRU)**. It selects a block for eviction, ensuring that modified blocks are written back to disk if necessary, to maintain data consistency. The buffer manager writes the evicted block back to disk only if it has been modified since its last write. Subsequently, it also reads the requested block from the disk

into the buffer. And finally, the buffer manager provides the program with the address of the requested block in main memory for data access.

A Database buffer managed by a buffer manager creates a necessary level of abstraction to the programs making I/O requests.

1.2.1.4. Column-Oriented Storage

What we have discussed till now, dealt with storing all attributes of a tuple (relation) in a row-oriented storage. However, with every increasing data accumulation and rise of NoSQL based databases, demands have risen for faster access of data, especially in analytical applications. Column oriented storage is one such approach which has been trending in recent times. Unlike traditional database storage mechanism, column-oriented storage is designed to store each attribute of a relation separately and values of the attributes from successive relations or tuples are stores at successive positions in the file.

Here, a typical employee table has been represented in column-oriented approach in Fig 8-0-12.

Column-oriented storage for databases is designed to reduce the overall disk I/O requirements and reduce the amount of data to be loaded from disk. Such set up can be easily scaled out using distributed clusters of low-cost hardware to increase throughput, which makes them ideal candidate for data warehousing and Big Data processing. Storing data by columns allows for better compression as similar data types are stored together thus reducing storage requirements. And since initial data retrieval is column specific, only columns needed for a particular query are retrieved, which results in reduced I/O operation and faster access time. This arrangement is well-suited for analytical queries, aggregations, data analytics and even parallel processing operations like filtering. In big data environments, columnar databases handle large volumes of structured data efficiently. Paraquet and Apache ORC are two widely used column-based storage formats.

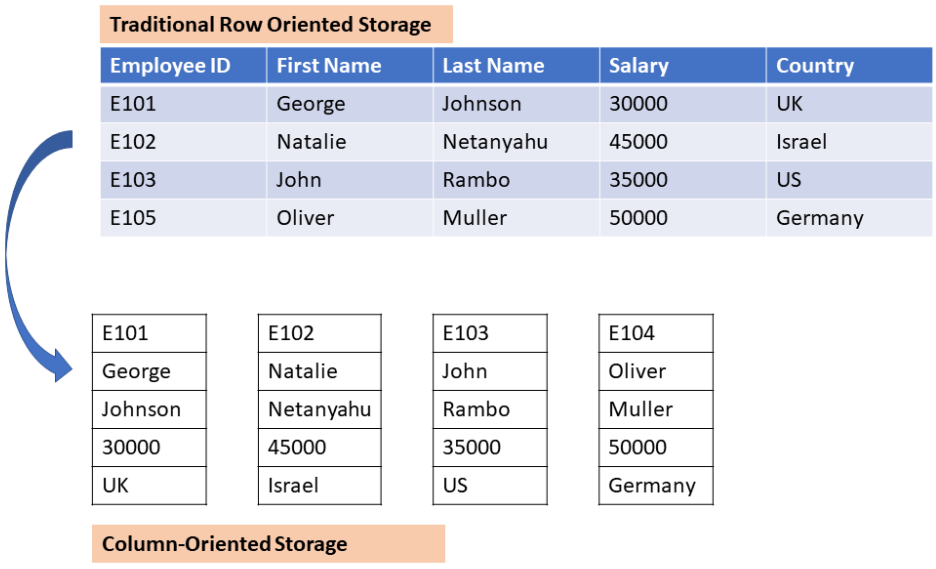

Figure 8-0-0-12 Column-Oriented storage

However, there are certain drawbacks of this approach which we must keep in mind and consider using columnar approach only in selected scenarios like analytics or Big Data processing. One of the main

drawbacks of this approach is that it makes reconstruction of the original tuple very costly, hence not suited for transaction processing applications.

1.3. Indexing

While discussing about structure of magnetic disks earlier, we discussed that for the sample example, to read 100 records, 25 blocks of the disk would need to be traversed. Imagine, if the number of the records are doubled or tripled over time, the look up time would also increase twice or thrice respectively. Suppose we are looking for a particular employee's record, then it is not efficient to look up every tuple in the given relational table, to find the specific record. In ideal situation, system should be able to locate these records directly.

To speed up the retrieval of records under certain search conditions, additional auxiliary access structures called **indexes** are used. Indexes are the additional data structures present in the file system, that provide alternate ways to efficiently access a record without affecting the original placement of records in the primary data file.

General idea behind the usage of index is to keep a metadata on the side which act as specialized maps or signposts to help us quickly locate specific information within the bulk of data. Just like an index in a book lists key topics and their page numbers, database indexes list values or keys (like words in an index) along with pointers to where that data is stored in the database (similar to page numbers).

When searching for records, the database uses the index to locate pointers or references that direct it to the specific disk blocks or locations in the data file where the desired records are stored. When we perform a search or query, the database engine refers to these indexes first. It quickly navigates through the index's signposts to find the relevant data rather than scanning the entire dataset, reducing the time it takes to locate specific information. If we want to search for the same data in different ways or based on different criteria, we might need several indexes targeting various parts of the data. Each index allows us to search the data using a specific field or criterion. For instance, one index might help search by customer names, another by product IDs, and so on.

Several databases extend the option to add or remove indexes (without altering the actual data stored in the database) to adjust for certain search condition, in order to improve query performance. Naturally, maintaining an additional structure is an overhead, especially when records are being added in the database (write operation); as the indexes would need to be updated every time there is a write operation related to the table on which index has been applied. This is considered as a necessary trade-off, as read performance of certain important query is enhanced significantly.

1.3.1.1. Types of Index: Ordered & Hashed

Moreover, not everything is indexed by default. It requires understanding of application's typical query pattern to choose indices that yield maximum benefit while limiting the overhead. For example, if we are trying to maintain index on a large table containing employee records, then implementing an index on employee relation by keeping a sorted list of employees ID will not be efficient, as index would be large in size itself and searching and update-delete operation on such index will be expensive. There are various data structure techniques (commonly B+ trees and HashMap structures discussed above) which are leveraged for implementing indexes. On a higher level, indices are divided into two categories: **Ordered indices**, based on sorted ordering of values and **Hash indices**, based on uniform distribution of values corresponding to their keys across a range of buckets (discussed above in section 8.2.1.1.5).

An **index entry or index record** serves as a reference (pointer) within a database that helps locate specific data based on a defined search key value, which is the value used for searching or querying data within the database. It is the specific attribute or field value that's being indexed for quick retrieval.

The pointers indicate where the records with the corresponding search-key value are stored i.e. disk block address and offset within the disk block to locate the record.

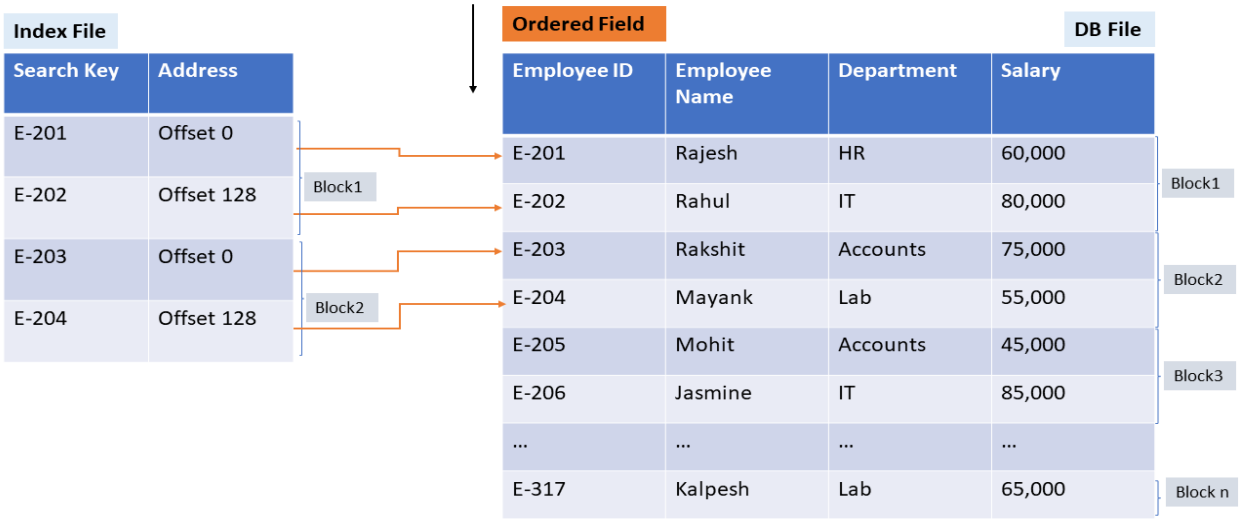

Figure 8-0-12 Dense Index

One approach to implement ordered index is to maintain an index entry for every search-key value in the file (see in Fig 8-0-12). This kind of arrangement is called **dense index**. In case a clustered index is used, a pointer to first data record corresponding to the search-key is maintained in the index file. Other approach can be to maintain an index entry only for selected search-keys (see in Fig 8-0-13). This kind of arrangement is called **sparse index**. To locate a record in this kind of set up, we find index entry with largest search-key value that is less than or equal to the search-key value we are looking for and then starting from the pointed record we start our search sequentially until the desired record is found in the file. Note: block size is considered only 128 bytes in diagrams referred.

Now, this kind of arrangement might not be very efficient if the tuples or records in the table are large in number (say in millions). In this case, a sparse outer index is created on the original index file (now referred as inner index file) and binary search is used first to locate in the inner index the largest search-key value that is less than or equal to the search-key value we are looking for. Then, corresponding offset of the disk block is referred to track the record in database file. This kind of arrangement is called **multi-level indices**.

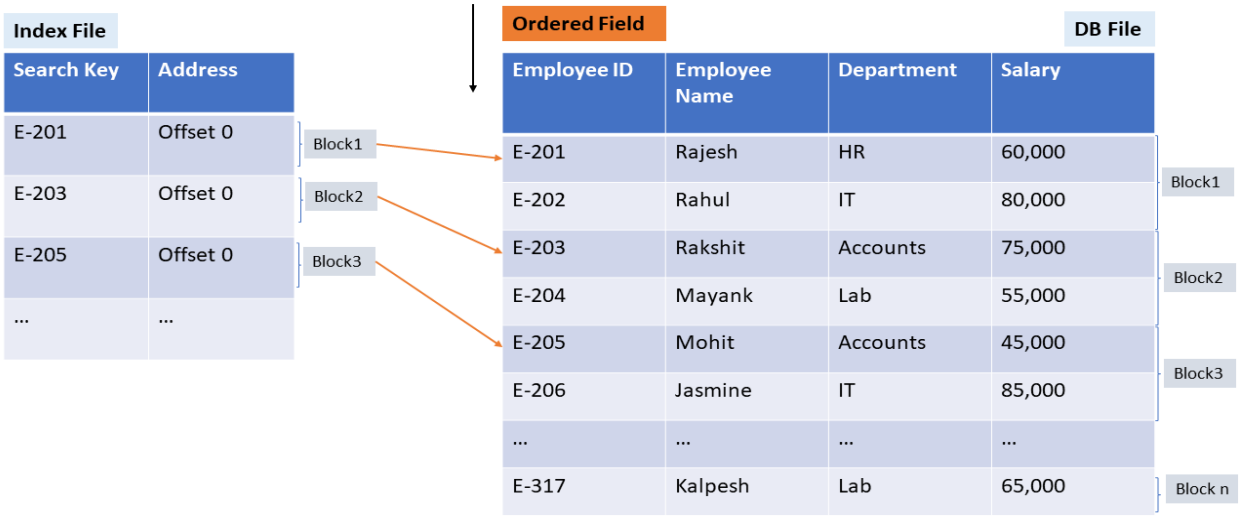

Figure 8-0-13 Sparse Index

It is to be noted that whatever form of index arrangement is implemented, every time an insertion or deletion or update operation affecting search key is performed, corresponding updates must be made in the index file. As evident, the performance of index-sequential file organization degrades as number of records increase. To counter this, indexes based on **B+ Tree** data structure is used to maintain efficiency despite insertion or deletion of data (discussed on section 8.2.1.1.4). Although a B+ tree structure imposes performance and space overhead on insertion and deletion, it is acceptable as file reorganization is avoided.

The B+ tree index structure, while efficient for many operations, faces challenges with random writes, especially when the index is too large to fit entirely in memory. When writes or inserts don't follow the order of the index, each operation often touches a different leaf node in the B+ tree. With a large number of leaf nodes and limited buffer memory, most operations require random reads followed by subsequent writes to update the leaf pages back to disk. For systems using magnetic disks, the time taken for seek operations significantly limits the number of writes/inserts per second per disk, often restricting it to around 100 writes/inserts per second. While flash-based SSDs offer faster random I/O, the cost of a page write, particularly the eventual page erase, remains a significant operation, affecting the overall performance for random writes.

1.3.1.2. Log-Structured Merge (LSM) Tree

In scenarios with high write rates, an alternative to B+ tree can be Log-Structured Merge Tree (LSM Tree), a popular data structure designed to optimize write operations. Since, we are talking about higher write rates, Log based data structures come into foray as they internally use Linked Lists (remember, Linked List have faster write time i.e. O(1); all they have to do is to append at the end of the node). However, this also means that the read operations are going to be considerably slower i.e. O(N). Now, in an enterprise set up, of course we want to speed up writes but not at the cost of reads (look ups or search queries) taking a significant hit. So how do we bring the best of both worlds? As, B+ trees are good at read or to say balanced read/write operations and linked list-based structures like Logs are good at writes but poor at reads.

So, what other data structures we can utilize for efficient reads? Answer would be Sorted Arrays, which have a read time of O(Log(N)). So, the idea is to sort the information or data supposed to be written in

the database and then persist it. Sorted String Tables (SSTables) are used for the purpose of maintaining key-value pairs of data and persisting on disk.

1.3.1.2.1. Working of LSM Tree

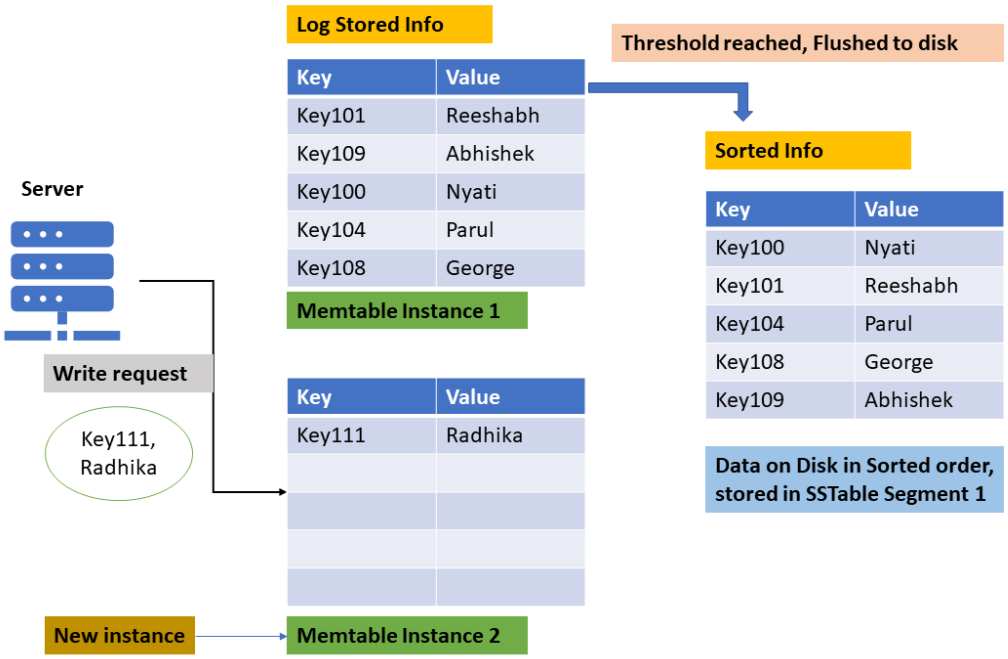

Figure 8-0-14 LSM Tree Implementation

In this arrangement (see fig 8-0-14), incoming writes are buffered in memory (RAM) in a structure known as the **memtable** (typically implemented as self-balancing tree data structure, such as red-black tree, AVL tree, etc.) and the keys within the memtable are kept in a sorted order, facilitating efficient iteration or traversal over the keys. Later, once the memtable reaches a certain size threshold (e.g., a few megabytes); the system stops further writes to that instance of the memtable. A new instance of the memtable is initiated to resume accepting incoming writes and the content of the previous memtable, which has reached its size threshold, is flushed to disk in sorted runs or segments, typically organized as **sorted files or SSTables (Sorted String Tables)**. Writing out the memtable as an SSTable file is efficient since the in-memory tree maintains keys in a sorted order. The newly created SSTable file becomes the most recent segment of the database, holding sorted key-value pairs. While an SSTable is being written to disk, new writes continue to a new instance of the memtable, ensuring uninterrupted write operations.

There is always a possibility of data loss with in-memory data structures as in case of a crash, data not written to disk might get lost completely. To mitigate this, we can use a separate log on disk to maintain a record of in-memory writes provides a mechanism to recover data in the event of a database crash or system failure. This strategy, often referred to as a **write-ahead log (WAL)** or journaling, ensures data durability as recent writes are captured in a persistent medium (disk) and facilitates recovery as the log serves as a backup, containing unflushed writes. The system can always regenerate the memtable from the log on recovery, thus restoring the in-memory writes that weren't yet flushed to disk. Once the contents of the memtable are successfully flushed to disk (stored in SSTables), the log becomes redundant and can be discarded or truncated. However, we must be cognizant of the fact that maintaining a write-ahead log incurs additional disk writes, which might impact performance.

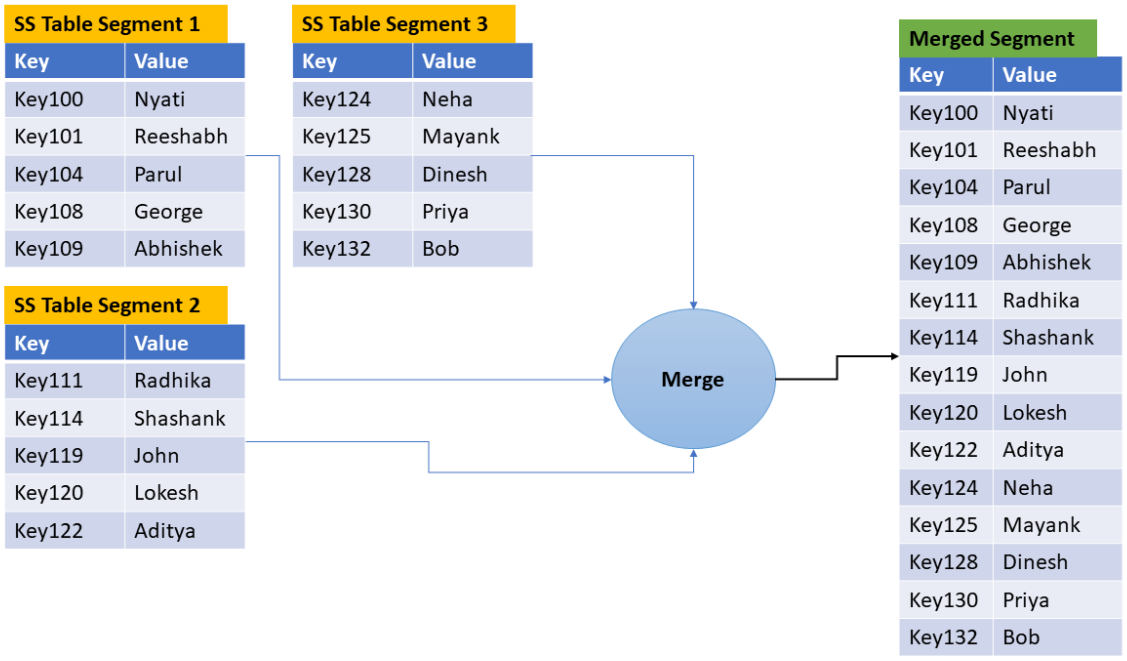

Figure 8-0-15 Merging & Compaction in LSM Tree

SSTables on the disk are present in multiple levels as we are flushing every instance of memtable, and each level containing progressively larger and more consolidated segments of data (see Fig 8-0-15). Periodically, a background process initiates merging and compaction operations on the SSTable files, which is done to optimise the reads and speed up the read process. Merging combines segment files, discards overwritten or deleted values, and optimizes the overall storage structure. By merging these segments while maintaining the sorted order of keys within each segment, compaction enhances read efficiency, minimizing the number of disk reads required to access data. Furthermore, compaction helps reclaim disk space by consolidating and removing redundant or obsolete data. There are various compaction strategies like levelling compaction, which merges smaller SSTables within the same level, and size-tiered compaction that consolidates segments across different levels of the database hierarchy. Since, all keys are stored in sorted order in respective segments, the merged sorted order segment is created in linear time, a big advantage over Hashing based data structures.

Storage engines like Lucene use similar kind of set up for storing its term dictionary, which paves way for implementing full-text search, implemented in ElasticSearch and Solr. For serving a read request, the system first checks the memtable for the key. If not found, it looks in the most recent on-disk segment (latest SSTable), then in older segments successively. To improve efficiency of database read operations, we can utilize a sparse in-memory index table alongside SSTable blocks. This specialized index table acts as a roadmap, holding specific keys paired with their memory locations, typically storing the initial key of each block within the SSTable that comprises multiple key-value pairs (see Fig 8-0-16). When a read request arrives, this index table becomes instrumental as it swiftly pinpoints the exact block storing the sought-after key. The system navigates through this specific block, scanning through a relatively small subset of entries within the block itself to locate the desired key-value pair. By storing only select keys in the in-memory index table, memory consumption is kept minimal, ensuring these indices comfortably fit within available memory resources. This optimization proves pivotal as it eliminates the need to traverse entire SSTable blocks and dramatically speeds up read requests.

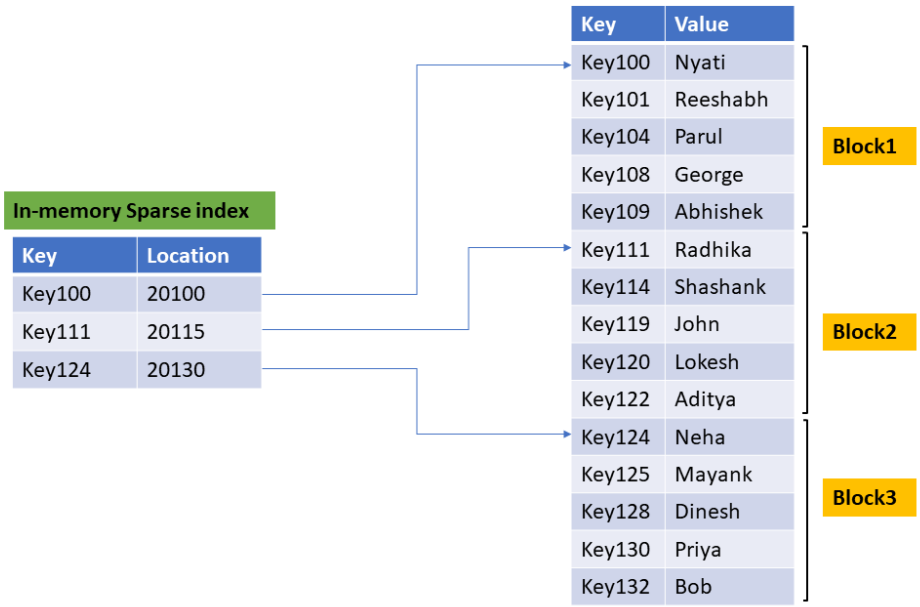

Figure 8-0-16 Use of sparse index to improve read in LSM tree

In some databases, to make the read operations more efficient, storage engines use additional filter like Bloom filter, a memory-efficient data structure for approximating the contents of a set. Benefit of a Bloom filter is that it can tell straight away if a key exists in the disk or not, hence saving unnecessary disk reads for non-existent entity.

In case, we are maintaining in-memory indexes, Hash indices (discussed on section 8.2.1.1.5) are preferred in this scenario. This approach is not commonly used during file organization as its major disadvantage lies in managing collisions. Bitmap indices are designed to query on multiple key and there are a number of techniques for indexing temporal data, including the use of spatial index and the interval B+ tree specialized index.

1.4. Summary

In this section, we discussed how files can be arranged logically as a sequence of records onto disk blocks and how efficiently we can retrieve them using indexes. We discussed several storage structures which work under the hood in database systems and implementation of indexes. We discussed various storage approaches like database buffer and column-oriented approach. In the next section, we shall be discussing some data models which are used in Database systems. Data Models help us understand how different databases structure their data and maintain relationship between entities.