Document In-Memory Persistent Database

a project documentation

submitted by

Mahdy Mohammad Mousa / Hamad

mhd0187795@ju.edu.jo

supervised by

Dr. Bilal Abu-Salih
Department of Computer Science

Dr. Heba Saadeh
Department of Computer Science

University of Jordan

King Abdullah II School of Information Technology

Department of Computer Science

Jan 2023

Abstract

This documentation shows the implementation of an **In-Memory Document Database**. Using efficient data structures and search algorithms to achieve the optimum performance. ***The testing conducted on this implementation shows that an average lookup time of an indexed data can be around 10 milliseconds in a database with millions of records.***

Database systems are complex and vary in implementations depending on the requirements. As this is an in-memory database, the primary focus is to develop fast and efficient search in memory. And will discuss enhancements to the proposed implementation. ***This project will not focus on indexing and data management on the disk***. We will focus more on building indexing in memory.

The implementation of the database is done using the Java Programming Language. No frameworks are used, pure java programming including: network and socket programming.

This database is named ***RaziDB*** after the Islamic philosopher and alchemist who lived during the Islamic Golden Age. This project is Open-Sourced on GitHub (*https://github.com/razidb*). And it is the first project of many to come focusing on building a scalable database.This represents the Core Database.
The code is available on this github repository: *https://github.com/razidb/razi-coredb*

Acknowledgement

As a bachelor student, working and building this project was a blast, I would like to thank my family for their continuous and unprecedented support.

And I would like to thank the supervisors of this project. As this project is not common between students, Dr. Bilal Abu-Salih and Dr. Heba Saadeh supported me to proceed with this challenging project.

Content

Chapter 1. Architecture

When building complex software, it is a good idea to break things into smaller components, so we have a clear image of the responsibility of each part. We will try to understand the responsibility of each component and try to connect them together. Remember, this is a general overview of the database, and we will talk about each layer in detail.

An important note that we need to address, that every connection open with the database is handled by a thread, so if we have multiple clients connected with the database, we will have multiple threads running and each of them serves a specific client.

Application Layer

This is the top layer of the database server. Its main responsibility is to receive a stream of bytes from a client and organize these bytes (based on a specific protocol) so that it is easy to extract useful information like collection, query, and operation. This layer is used when a connection is established over the network. But sometimes, the database could be accessed on the device itself, so there is no need for the application layer. So, this layer is optional, we may not see it if the owner decided to access the database from the shell.

Access Control Layer

At this point, we have the database to be accessed and what operation will be applied on which resources. For security reasons, we need to give access to authorized users only. Here, we assume that a session was created for a specific user, and we will talk about that in detail in this section. So, we know the user is trying to access a specific resource. The objective here is to check somehow if that user has access to perform a specific operation on specific resources.

Parsing Layer

We received the operation; we need to parse it and generate a set of operations that can be executed and applied on the data. Every operation has a parser, because read operation is parsed differently from delete operation.

Data Layer

The set of operations generated by the parser will be applied to the data. Our data is stored on disk and on RAM. So, we need to appropriately manipulate these data using specific data structures and algorithms for efficient execution.

Chapter 2. Networking – Application Layer

As the database is a system that can be accessed over the network most of the time. It makes sense to have an overview of the way we will expose the database. For any system to communicate with the outer world, as our database is technically a server, we need to listen to coming requests. We can use socket communication to receive connections to the database and transfer data through the socket connection to the client. We will use TCP connection to make sure the socket connection is reliable, and we do not lose data if anything goes wrong.

Ok, let us assume that we have established a socket connection between our database and some client. We must have a set of rules that specify how data is transferred and in what form. We also need to control access to the database resources and data. We can do this by implementing authentication techniques which we will take about in detail in the authentication and authorization section.

Also, we need to specify what is the accepted form for receiving queries from clients. Because our server will be processing data that comes from clients through the socket connection. The client and the server need to agree on the format sent from the client, and because the server will send back responses, the client also needs to be aware of the response format.

2.1. Data Flow Between Client and Database Server

2.1.1 Request to connect

This is the first step to be able to establish a connection with the database. In this step, the client sends authentication credentials to only one. If authentication fails, the socket connection will terminate.

```
Database                                                                    Client

   |                                                                          |
   |        <------------------------------ username, password -------------------------------        |
   |                                                                          |
   |         ----------------------------------      ok      ---------------------------------->      |
   |                                          OR                              |
   |         ---------------------------------- unauthorized ------------------------------->         |
```

If authentication succeeds, the database will create a session and will start accepting queries from the client.

2.1.2 Perform Queries

Queries are sent to the database in JSON format. Because JSON is fixable and it has nested fields, we can construct complex queries without the need to write our own query language like SQL – Structured Query Language. After the query is resolved, the database will send the result of the query back to the client.

```
Database                                                                    Client

   |                                                                           |
   |              <----------------------------- Query ------------------------------               |
   |                                                                           |
   |          ---------------------------    Query Result    --------------------------- >          |
   |                                                                           |
   |                                                                           |
```

At this point, the client can perform as many queries as possible if the session is not closed due to time-out. A time out exceeding 1000 milliseconds will result in connection termination and will automatically close the session. This is done to prevent the waste of resources. Because each connection is handled by a thread on the machine running the database. If a timeout limit was not added, we can end up with many connections open but not necessarily utilized.

2.1.3 End Connection Request

This type of request is used to inform the database that the client is requesting termination of the socket connection open with the server. This can be used when the client has no more queries to perform and to inform the database that it does not need the resources and connection allocated for the current open connection.

```
Database                                                                    Client

   |                                                                           |
   |        <----------------------------- End Connection ------------------------------        |
   |                                                                           |
   |                  * Deallocate Resources and end connection                |
   |                                                                           |
   |                                                                           |
```

2.2 Data Format

For two parties to communicate with each other, they should agree on certain things. One of these things is how data is formatted. This will enable both parties to encode and decode data as it is transmitted through the socket.

Data is sent and received as a sequence of bytes. When either server or client sends data to the other end of the connection, the data should be encoded to bytes. For the otherside to decode these bytes back to the original data, both sides should share the way data is decoded and encoded for data to be transmitted correctly.

In this implementation, Type-Length-Value (TLV) Encoding Protocol is used. The first part is a 1 byte used to set the type of data sent. The type can be string, binary, etc. The second part is a 1 byte character that specifies the operation of this request. Operation byte can be: [s] for selection operation, [i] for insert operation, [u] for update, [d] for delete operation, and [e] for end connection. The third part is a 4 byte integer specifying the length of the data in bytes.

Four bytes can have positive integers constructed of 4 bytes and can have a maximum value of 4294967294. Last part is the data or payload of the request. As we said, it is encoded as a series of bytes of length specified on the content-length part.

```
------------------------------------------------------------------------------
| content type | operation type | content length |             data          |
^--- 1 Byte ---^---- 1 Byte ----^--- 4 Bytes ----^----------------------------
```

The last part of the request contains the payload that contains most of the information we need. For the server to understand what the client needs, we have to agree on how the request should look like. And for the client to understand what the server has sent, we should agree on how the response is structured.

2.2.1 Request Format

We have two types of requests: authentication and query. Authentication request is used to establish a connection with the server, by providing user credentials.

```
{ "username": "...", "password": "..."}
```

For query requests, we have 3 main things: database name, collection name, and payload. The database and collection names are important information to know what resources the client is trying to access. And the payload contains different things depending on the type of operation the client is trying to perform.

```
{
  "type": "document",
  "database": "db-name",
  "collection": "collection-name",
  "payload": {...},
}
```

2.2.2 Response Format

If you do not know already, a response is usually generated from the server (in this case is the database). As query is processed and applied on database resources, normal data response can be expected depending on the type of operation.

Yet, some issues can be raised to the client with a clear description of what went wrong. Status code is common in network communications. Status code is an integer value, every code has a meaning. For example 200 means that the query was successful. This is helpful for the client side. It can be used for debugging and for data validation.

```
{
   "statusCode": 200,
   "queryResult": {...}
}
```

Chapter 3. Access Control

In this section, we will look at the way we manage access control on the database. Access control is a security layer which restricts access to database resources. In this implementation, a system file (called acl) is responsible for storing access control entries.

The database engine stores data related to internal database data like users collection and access control list collection under the /system/ directory. To keep this data separated from user-defined databases and collections. **Security on the database is done on two levels:**

3.1 Authentication

Any connection to the database requires authentication. And this is the first step to establish a socket connection with the database server. By providing the username and the password. Password protection is very necessary when it comes to storing user data.

The password is not stored as plain text in the database, its SHA-256 hash is stored. To check that the password is correct, the password is hashed using a hash function that implements SHA-256 to give the digest. And then it is compared with the hashed password. If they are equal, the user is successfully authenticated and a socket connection is established. Otherwise, the connection is terminated and resources are deallocated for the connection.

Enhancement - A security enhancement can be done before starting the authentication process. To make sure data is transmitted securely through the socket, the server should provide a certificate to take advantage of Public Key Cryptography to authenticate securely and also to generate a set of secret keys that can be used later for data exchange in a secure manner.

3.2 Authorization and Access Control

Every access to database resources is examined by an access control layer. Before resolving any query, the access control layer knows who the user is trying to access which database and which collection of documents.

Taking this information into consideration, the access control layer checks if the user is authorized to access the requested resources using the access control list manager that implements methods to access the acl file and checks the access control list associated with a specific resource. Usually these resources are databases and collections.

Chapter 4. Query Parsing - Operation Builder

When the query is received by the server in its json format, we will not be able to construct efficient queries by looking at the json query. We should think about a way to process the query. To be able to execute a query on the database, we need to parse it. The result of parsing is to build a group of operations that can be applied on a set of data.

Let us see an example of a JSON query.

```json
{
  "faculty": "KASIT",
  "isActive": true,
  "$OR": [
    {"department": "CS"},
    {"department": "CIS"}
  ],
  "credit": {
    "$lte": 90,
    "$gte": 40
  }
}
```

With JSON, we can construct data hierarchy, and we can deal with each level as nested query. In each level, we have attributes. These attributes can indicate a direct field lookup, complex field lookup, or a nested query with multiple levels.

We now have an idea of how a query looks like. But we need to understand how data is fetched in order to parse the query. We have two ways to fetch data, by using a pre-indexed set of fields for a specific collection, or via disk. If there exists a set of indexed fields, we can use index search, which is the preferred way. Because the data is stored in memory and can be accessed quickly. But if the query does not contain indexed fields, we are forced to perform a full collection search on disk. And because disk access is expensive, we need to make sure that we access the disk once and get all the data we need.

Ideally, we expect the fields to be queried to be indexed. But the database should deal with any number of fields, whether indexed or not. We can have n fields; m fields share one or more indexes and k fields with no index. A mix of index lookups and disk lookups should be expected.

Another thing that should be taken into consideration is the order in which queries are performed in case of complex queries. Queries results should be ORed and ANDed in the correct order.

We can think of the query as a mathematical equation. There are a couple of ways we can represent a mathematical equation in a way that every operation is applied to a set of numbers. And there are priorities. **Prefix Notation** is known as: a mathematical notation in which operators precede their operands. It does not need any parentheses if each operator has a fixed number of operands.

Prefix notation does not fully align with what we are dealing with. Because we must use some kind of parentheses to know when to stop applying an operation. Because some operations have variable length of operands. We can use the same concept to resolve the query.

We can represent the example query #xx as an operation stack as follows:

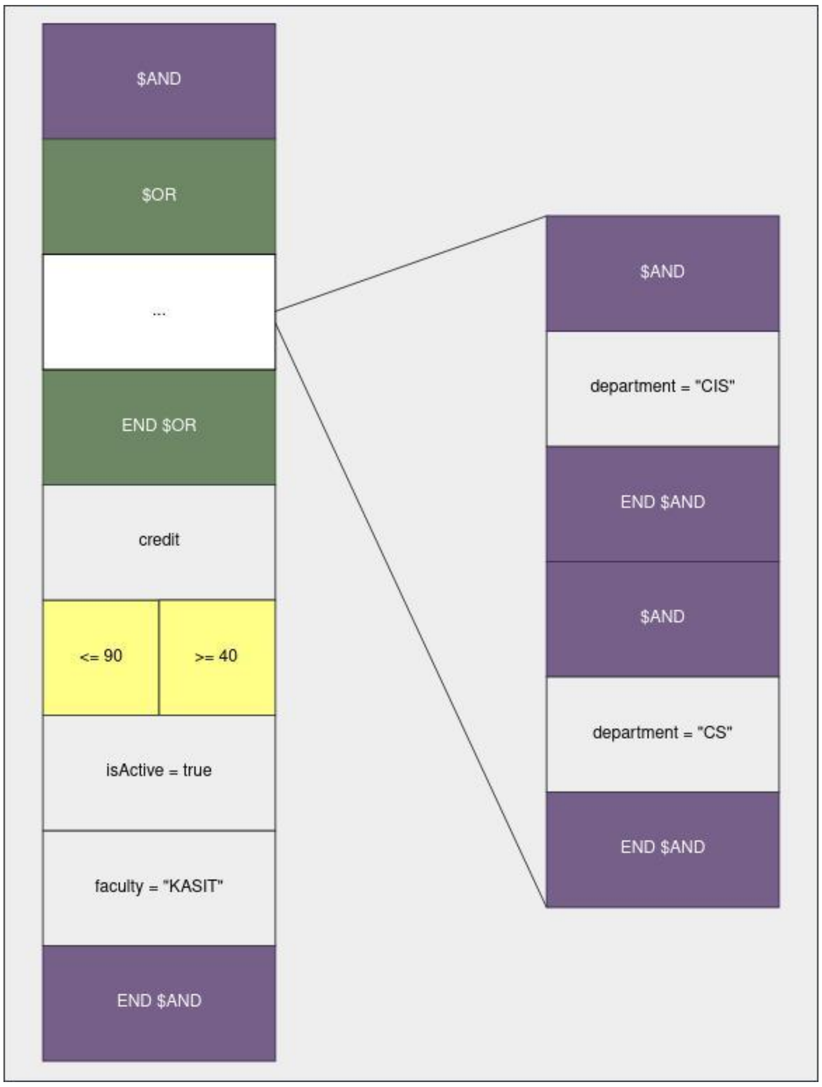

Figure 4-1. Operations Stack

At the end of parsing the query, we end up with an operations-stack which contains operations and operands in the correct order for execution. Operations-stack can be parsed recursively, to resolve the query.

Operation Stack Resolver

To resolve the operation stack, similar to postfix and prefix notations, we pop the first item in the stack, if it is an operation (Like $OR) we keep popping items until we reach the END OPERATION item. After we have collected all lookup fields for a specific level and operation, we look through these lookups to see if we can find an associated index. If an index is found, we resolve these lookups by implementing a binary search which is the responsibility of the index. Otherwise, we perform a full collection search to resolve these lookups. Doing that recursively, when we reach a nested query. The return of each recursive call is a queryset, which can be merged or intersected with the current operation result. After resolving the operations-stack, we end up with the queried data.

Chapter 5. In-Memory Data Storage

In this section, we will discuss how data is indexed in memory and what data structures will be used to efficiently store and retrieve data from memory. With the evolution of on-disk storage hardware, accessing disk is very costly compared to memory. That is why some databases are designed to store most of the data in memory.

First, what do we mean by indexing? Indexing is a data structure technique which allows you to quickly retrieve records from a database file. Indexing can be implemented either on disk or in memory. The data structures used to build an index in memory differ from data structures used to build an index on a desk.

What do we expect from an in-memory index?

- An index should be built on a pre-specified key.
- Fast search compared to a large data set (log(n))
- Fast insert and delete or records
- Ability to search for a range of values
- Ability to retrieve data in a specific order

A viable choice would be a tree where we can organize nodes in a specific way. Binary Search Trees can achieve all these requirements.

5.1 Binary Search Trees

A *binary search tree* (BST) is a sorted in-memory data structure, used for efficient key-value lookups. BSTs consist of multiple nodes. Each tree node is represented by a key, a value associated with this key, and two child pointers (hence the name binary). BSTs start from a single node, called a *root node*. There can be only one root in the tree. Figure 5-1 shows an example of a binary search tree.

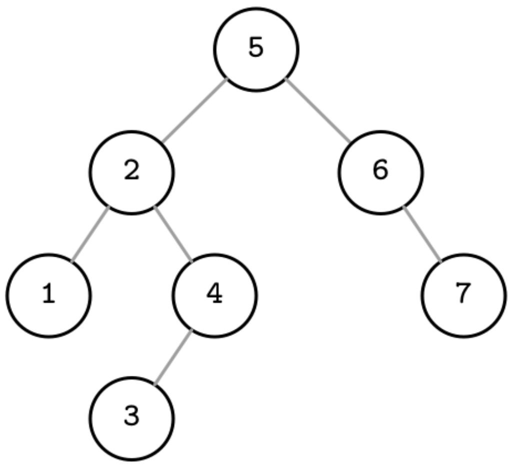

Figure 5-1. Binary search tree

Each node splits the search space into left and right *subtrees,* as Figure 5-2 shows: a node key is *greater than* any key stored in its left subtree and *less than* any key stored in its right subtree.

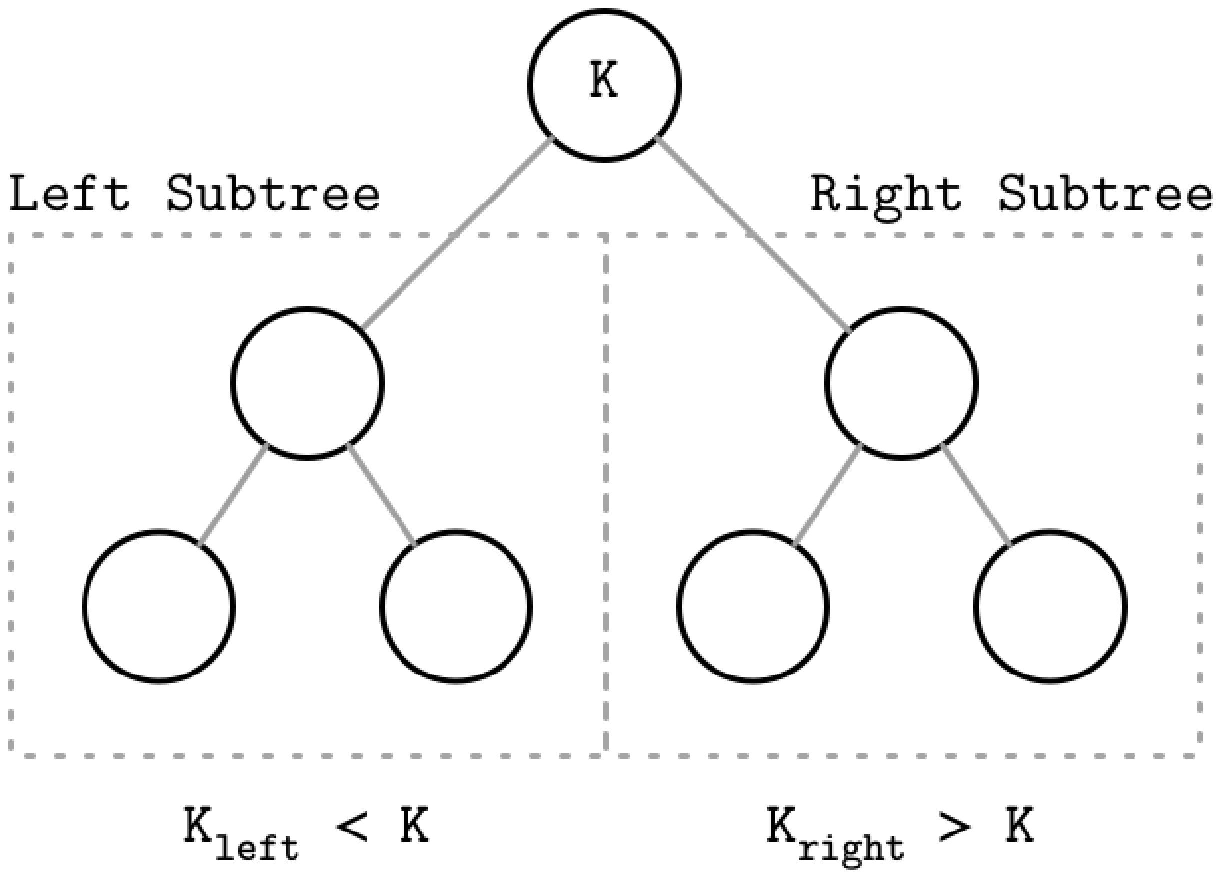

Figure 5-2. Binary tree node invariants

Following left pointers from the root of the tree down to the leaf level (the level where nodes have no children) locates the node holding the smallest key within the tree and a value associated with it. Similarly, following right pointers locates the node holding the largest key within the tree and a value associated with it. Values are allowed to be stored in all nodes in the tree. Searches start from the root node, and may terminate before reaching the bottom level of the tree if the searched key was found on a higher level.

A Binary Search Tree is a binary tree but with some restrictions that are applied on insertions and deletions of nodes. Every node has at most two children, hence the name binary. Any parent node should be greater than its left child and less than its right child.

Let us take an example. If you look closely at the following tree, you will find that it in fact applies characteristics of a Binary Search Tree. So, if we want to look for the number 13. What should we do? First, we go to the root node. The number we are looking for (13) is greater than 8, so we go to the right side of the tree, which is 10. We keep looking until we reach an empty node, that way we know that the number we are looking for does not exist. Or we find the number we are looking for.

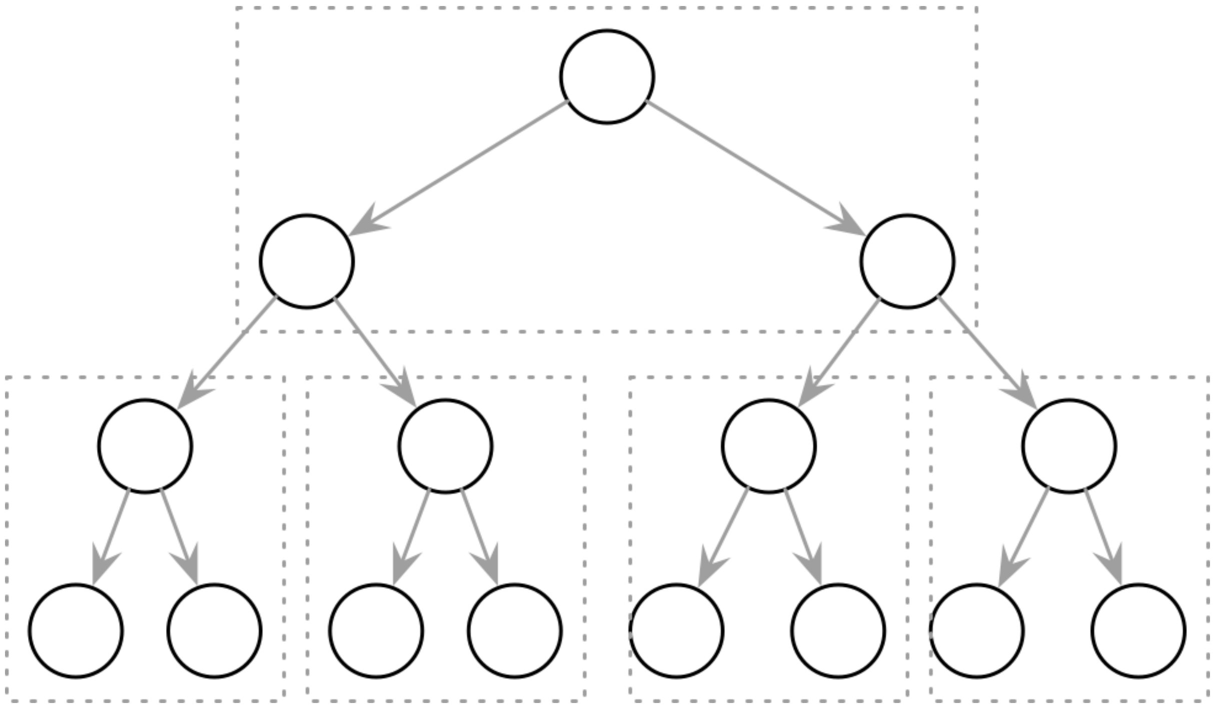

Figure 5-3. Binary search tree hierarchy

But you can see that the tree is not balanced. In our example, you can see that the right subtree is not balanced. When we previously were looking for number 13, we did an extra comparison to find the number 13. So, we should maintain a tree with a minimum depth to minimize the number of comparisons which decreases lookup time. There are multiple types of Self-Balancing Binary search trees that can maintain a tree with minimum depth. Examples of self-balancing binary search trees are: AVL and Red-Black Tree.

The main difference between AVL and Red-Black Binary Search Trees, BST for short, is that AVL is strictly balanced while Red-Black Tree is not strictly balanced. This results in AVL tree having efficient search but complex insertion and deletion due to the required rotations to achieve balance after every insert or delete operations. While Red-Black Trees have less complex insertion and deletion, it does not provide efficient querying due to its not strict balance.

To have efficient search, we should keep the tree in balance. This can be done by performing rotations for every insertion or deletion. In AVL, we have 4 types of rotations: Left Rotation, Right Rotation, Left-Right Rotation, and Right-Left Rotation.

5.2 Tree Balancing

Insert operations do not follow any specific pattern, and element insertion might lead to the situation where the tree is unbalanced (i.e., one of its branches is longer than the other one). The worst-case scenario is shown in Figure 5-4 (b), where we end up with a *pathological* tree, which looks more like a linked list, and instead of desired logarithmic complexity, we get linear, as illustrated in Figure 5-4 (a).

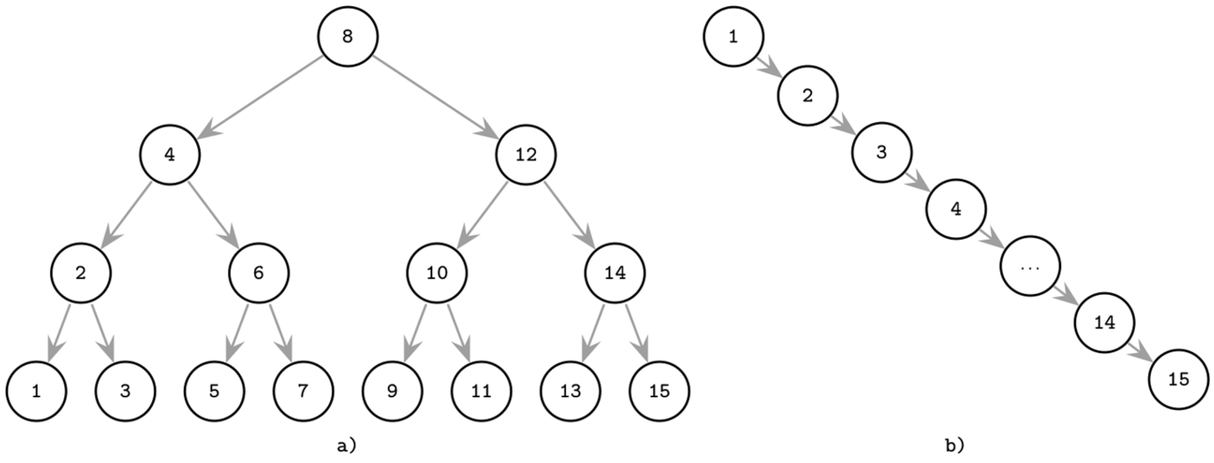

Figure 5-4. Balanced (a) and unbalanced or pathological (b) tree examples

This example might slightly exaggerate the problem, but it illustrates why the tree needs to be balanced: even though it's somewhat unlikely that all the items end up on one side of the tree, at least some of them certainly will, which will significantly slow down searches.

The *balanced* tree is defined as one that has a height of $\log_2 N$, where N is the total number of items in the tree, and the difference in height between the two subtrees is not greater than one. Without balancing, we lose performance benefits of the binary search tree structure, and allow insertions and deletions in order to determine tree shape.

In the balanced tree, following the left or right node pointer reduces the search space in half on average, so lookup complexity is logarithmic: $O(\log_2 N)$. If the tree is not balanced, the worst-case complexity goes up to O(N), since we might end up in the situation where all elements end up on one side of the tree.

Instead of adding new elements to one of the tree branches and making it longer, while the other one remains empty (as shown in Figure 5-4 (b)), the tree is *balanced* after each operation. Balancing is done by reorganizing nodes in a way that minimizes tree height and keeps the number of nodes on each side within bounds.

One of the ways to keep the tree balanced is to perform a rotation step after nodes are added or removed. If the insert operation leaves a branch unbalanced (two consecutive nodes in the branch have only one child), we can *rotate* nodes around the middle one. In the example shown in Figure 5-5, during rotation the middle node (3), known as a rotation *pivot*, is promoted one level higher, and its parent becomes its right child.

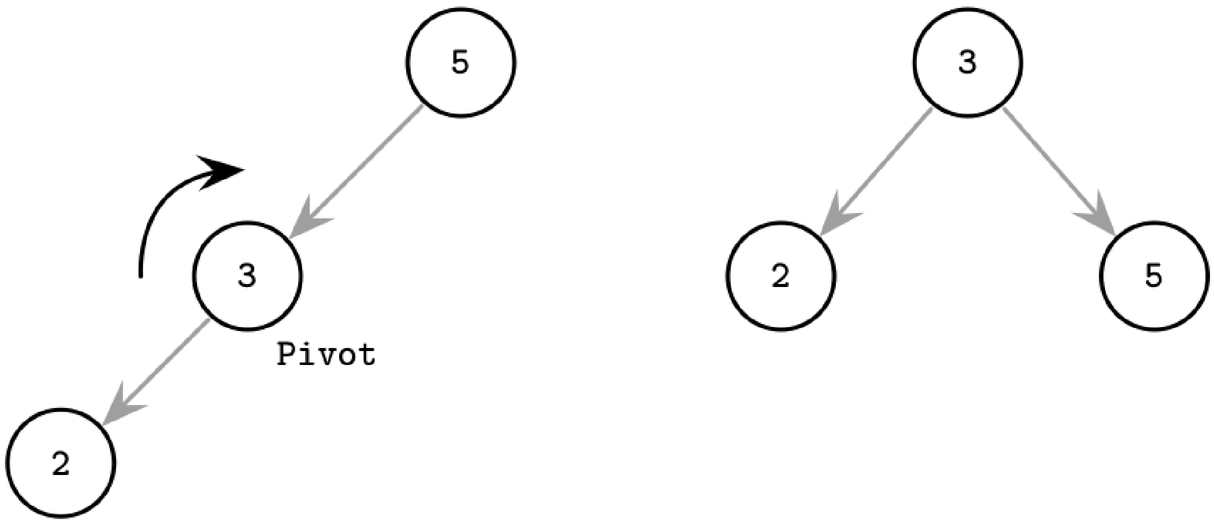

Figure 5-5. Rotation step example

5.3 Data Indexing

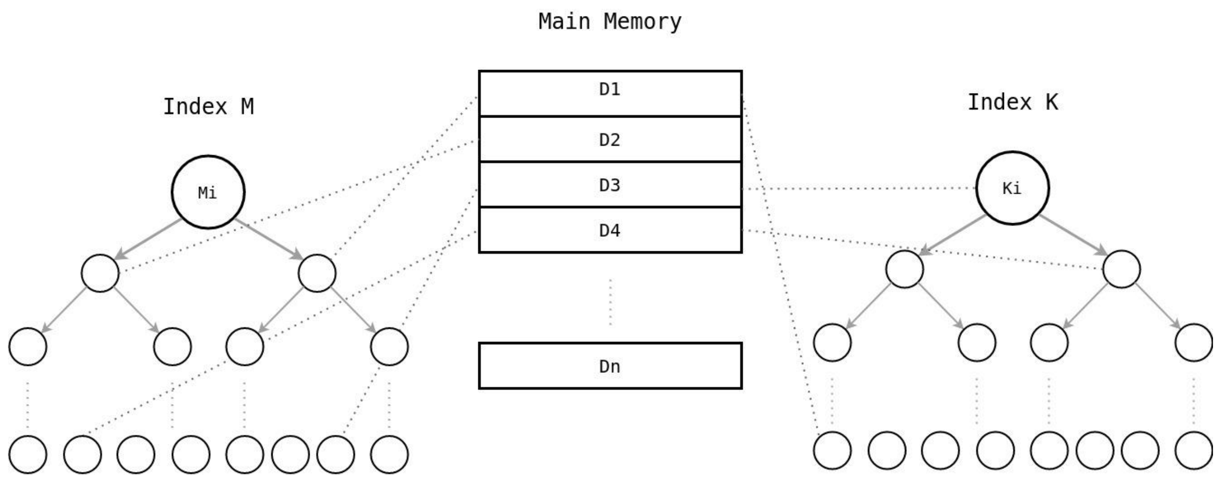

Figure 5-6. Main memory and indexes relationship

In this implementation of index, all data should fit to main memory to be able to construct indexes. Every index can be represented as a Binary Search Tree. To build an index using a binary search tree, we will be working with comparable keys.

We have 2 types of indexes: single-field index and multi-field index. Single-Field Index can be easily done by taking the field to be indexed as the key and build a BST. Multi-Field Index can be constructed by generating a key by concatenation of index fields.

Index should be stored on disk. Each index needs 2 things: field/s to be indexed and a name. Usually, the name of the index is the concatenation of its fields. The issue with multi-field index is the construction of the key. Each time we want to look for a specific node, we should construct a key for that specific document according to the order of fields.

All documents are stored in memory, and the nodes of an index contain two things: the key, and a pointer to the document in main memory. By doing that, we are saving memory.

Chapter 6. On-Disk Data Storage

While it is not the focus of this project to develop efficient storage on disk, it is a very important aspect for any storage engine. Because disk access is expensive, data should be indexed on disk using special data structures like B-Tree which is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children.

It is important to mention B-Trees, but we will implement a simple indexing method that can work for our use case of an in-memory persistence database. Although this is the final goal, we should be able to perform basic on-disk operations like: read, insert, update, and delete on a specific collection. Dealing with write operation with a non-indexed collection on-disk can easily drop performance to very low levels. Many databases including SQL and NoSQL databases implement indexing on disk using B-Trees that guarantees Log(n) time complexity.

6.1 Simple Indexing

In this database, we do not actually need the features gained from implementing B-Tree indexes on-disk. Most of the queries should be done on indexed fields which are loaded to main memory as explained in the previous chapter. So when we need to delete a record, we should be able to have an identifier for that record. One of the limitations of the identifier, which will be auto generated by the database, is the fact that there will be only one type of ID and it will be of type Integer. Many databases like MongoDB have restrictions on identifiers generated internally by the database.

The idea of having an ID of type Integer, is the ability to be generated sequentially. The first record is assigned ID = 1, and the second will be 2, and so on. As the records will be inserted sequentially, the IDs will be sequential. So it is easy to find a record given its ID by using the Binary Search Algorithm which has Log(n) time complexity.

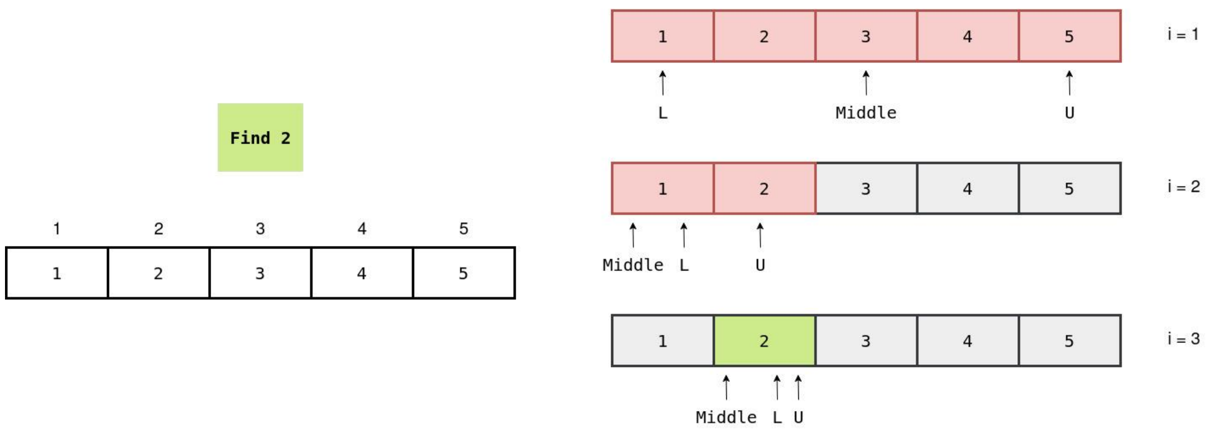

Figure 6-1. Binary search algorithm

Binary Search Algorithm

Binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array.

We can use Binary Search to look for a number in a list of sorted numbers. The algorithm works by setting an upper and lower bounds which are initially the start of the array and the end of the array. By accessing the middle element and comparing it with the number we are looking for. If the number we are looking for is greater than the middle number, we set the middle element + 1 as the new lower bound. But if the middle element is larger, middle element -1 is set as the new upper bound.

If this is done enough times, we can find the number when the middle is the number we are looking for, or lower and upper bounds are intersected. If the lower and upper intersected and we did not find it, the number does not exist.

6.2 Implementing Simple Indexing

To be able to access every document, they have to be separated in some way. New line `\n` can be used to separate every document from the other. To keep track of the sequence of IDs, we need to store the identifier for the last inserted record. So, when a new Item is inserted, we simply increment this value and assign it to the new record.

Data is usually encoded in bytes. Because it is easily parsed and flexible, in addition to other advantages. But for simplicity, we will use one text file for every collection to store its documents. The first line can be used as a header for the collection, but we will only use it to store the last inserted identifier.

There should be a format for every line. There are two options for storing the identifier in the line: either ID can be inside the document data, and we should parse the data from string to JSON to obtain the ID, or we store the ID and the document in comma-separated-values to be able to get the ID without parsing the whole document data.

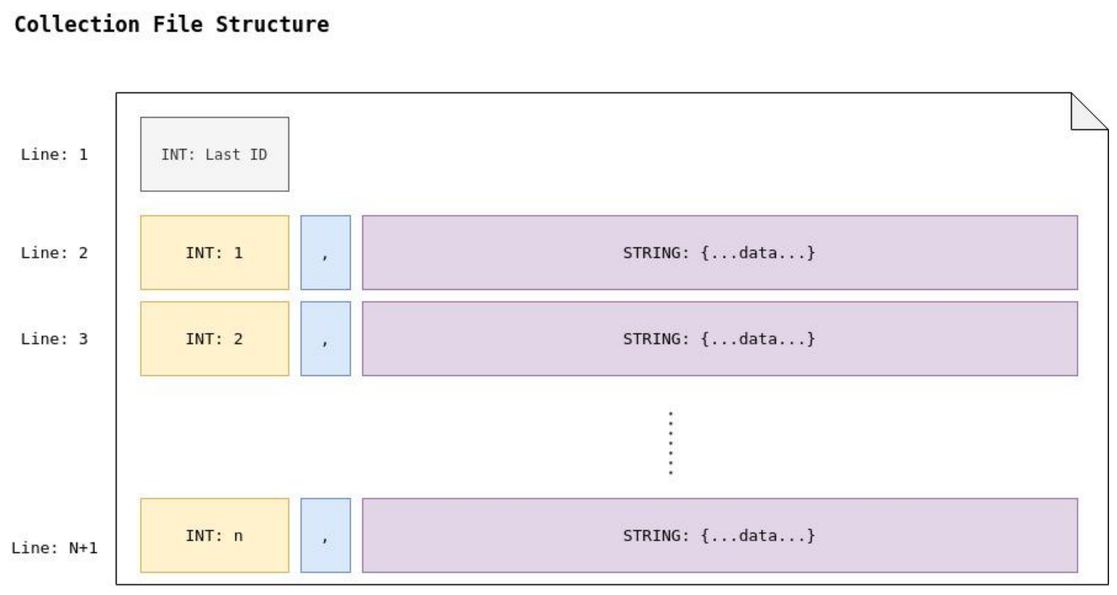

Figure 6-2. Collection's file structure suitable for binary search

6.3 B-Trees

Besides the cost of disk access itself, the main limitation and design condition for building efficient on-disk structures is the fact that the smallest unit of disk operation is a block. To follow a pointer to the specific location within the block, we have to fetch an entire block. Since we already have to do that, we can change the layout of the data structure to take advantage of it.

This word has slightly different semantics for on-disk structures. On disk, most of the time we manage the data layout manually (unless, for example, we're using memory mapped files). This is still similar to regular pointer operations, but we have to compute the target pointer addresses and follow the pointers explicitly.

Most of the time, on-disk offsets are precomputed (in cases when the pointer is written on disk before the part it points to) or cached in memory until they are flushed on the disk. Creating long dependency chains in on-disk structures greatly increases code and structure complexity, so it is preferred to keep the number of pointers and their spans to a minimum.

In summary, on-disk structures are designed with their target storage specifics in mind and generally optimize for fewer disk accesses. We can do this by improving locality, optimizing the internal representation of the structure, and reducing the number of out-of-page pointers.

Ubiquitous B-Trees

B-Trees can be thought of as a vast catalog room in the library: you first have to pick the correct cabinet, then the correct shelf in that cabinet, then the correct drawer on the shelf, and then browse through the cards in the drawer to find the one you're searching for. Similarly, a B-Tree builds a hierarchy that helps to navigate and locate the searched items quickly.

As we discussed in "Binary Search Trees", B-Trees build upon the foundation of balanced search trees and are different in that they have higher fanout (have more child nodes) and smaller height.

In most of the literature, binary tree nodes are drawn as circles. Since each node is responsible just for one key and splits the range into two parts, this level of detail is sufficient and intuitive. At the same time, B-Tree nodes are often drawn as rectangles, and pointer blocks are also shown explicitly to highlight the relationship between child nodes and separator keys. Figure 6-3 shows binary tree, 2-3-Tree, and B-Tree nodes side by side, which helps to understand the similarities and differences between them.

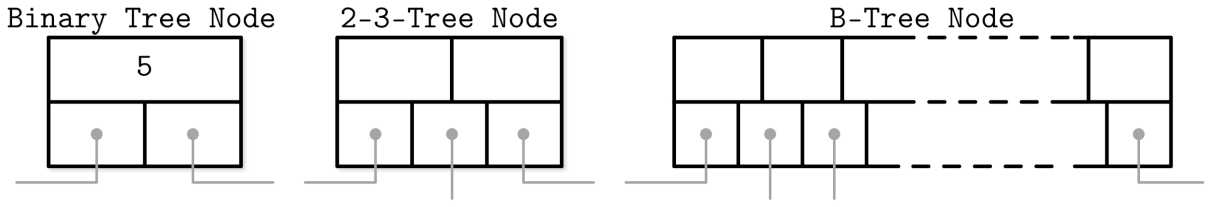

Figure 6-3. Binary tree, 2-3-Tree, and B-Tree nodes side by side

Nothing prevents us from depicting binary trees in the same way. Both structures have similar pointer-following semantics, and differences start showing in how the balance is maintained. Figure 6-4 shows that and hints at similarities between BSTs and B-Trees: in both cases, keys split the tree into subtrees, and are used for navigating the tree and finding searched keys.

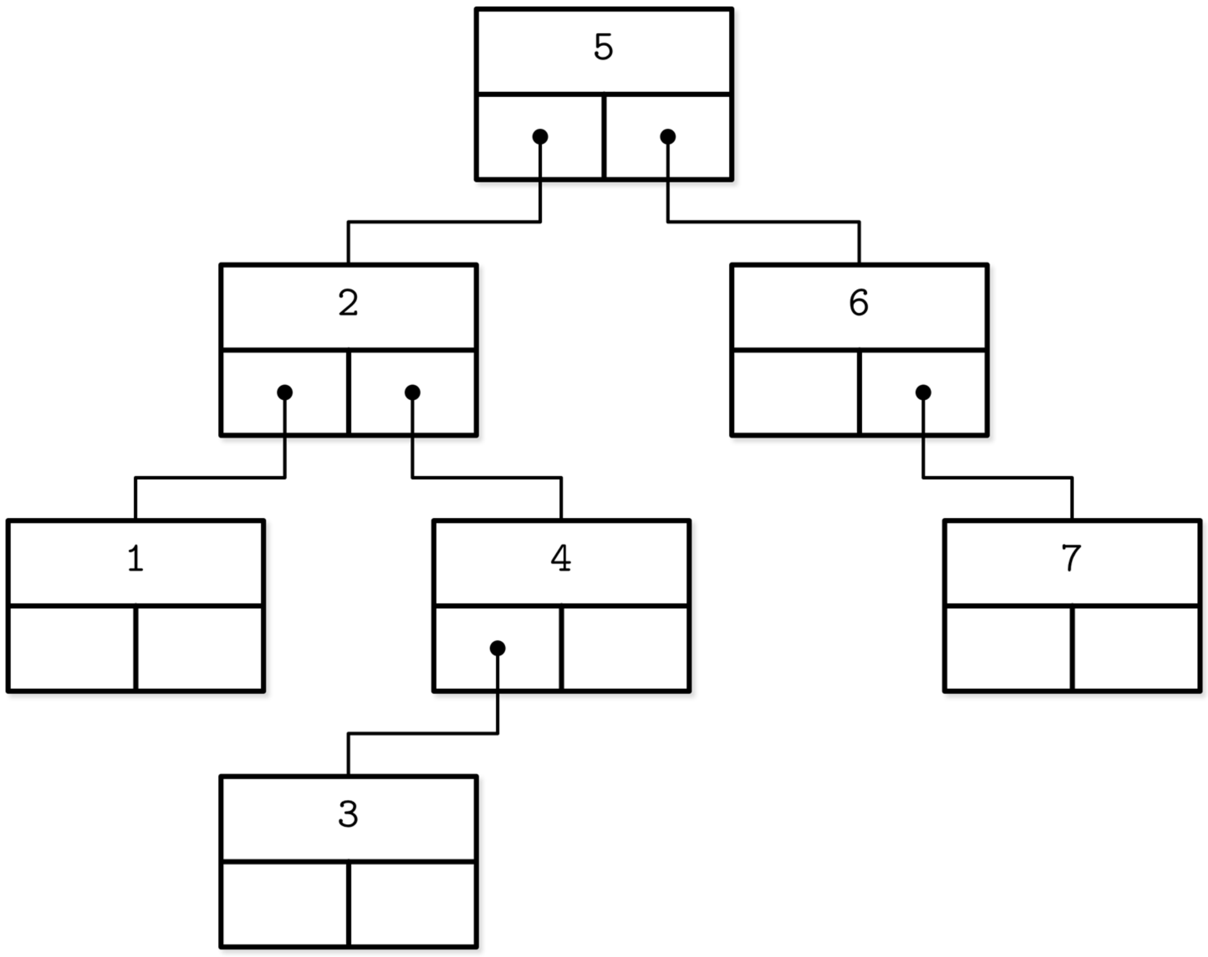

Figure 6-4. Alternative representation of a binary tree

B-Trees are *sorted*: keys inside the B-Tree nodes are stored in order. Because of that, to locate a searched key, we can use an algorithm like binary search. This also implies that lookups in B-Trees have logarithmic complexity. For example, finding a searched key among 4 billion (4×10^9) items takes about 32 comparisons. If we had to make a disk seek for each one of these comparisons, it would significantly slow us down, but since B-Tree nodes store dozens or even hundreds of items, we only have to make one disk seek per level jump. We'll discuss a lookup algorithm in more detail later in this chapter.

Using B-Trees, we can efficiently execute both *point* and *range* queries. Point queries, expressed by the equality (=) predicate in most query languages, locate a single item. On the other hand, range queries, expressed by comparison (<, >, ≤, and ≥) predicates, are used to query multiple data items in order.

B-Tree Hierarchy

B-Trees consist of multiple nodes. Each node holds up to N keys and N + 1 pointers to the child nodes. These nodes are logically grouped into three groups:

Root node

This has no parents and is the top of the tree.

Leaf nodes

These are the bottom layer nodes that have no child nodes.

Internal nodes

These are all other nodes, connecting root with leaves. There is usually more than one level of internal nodes.

This hierarchy is shown in Figure 6-5.

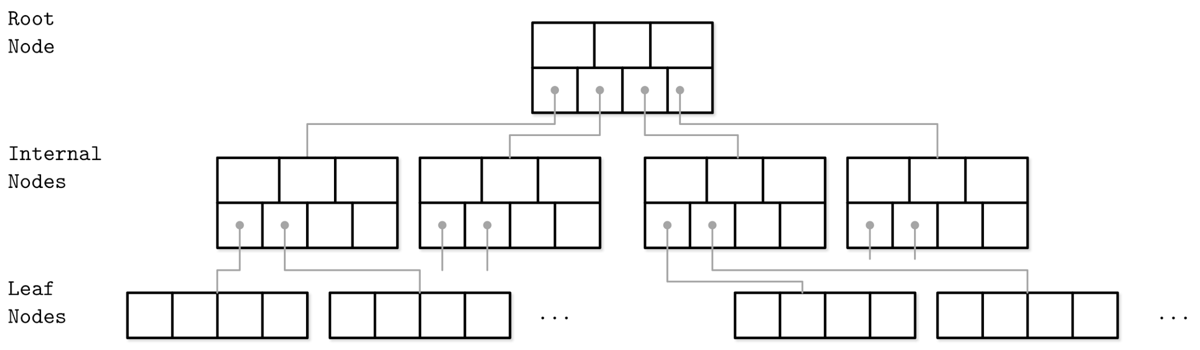

Figure 6-5. B-Tree node hierarchy

Since B-Trees are a *page* organization technique (i.e., they are used to organize and navigate fixed-size pages), we often use terms *node* and *page* interchangeably.

The relation between the node capacity and the number of keys it actually holds is called *occupancy*.

B-Trees are characterized by their *fanout*: the number of keys stored in each node. Higher fanout helps to amortize the cost of structural changes required to keep the tree balanced and to reduce the number of seeks by storing keys and pointers to child nodes in a single block or multiple consecutive blocks. Balancing operations (namely, *splits* and *merges*) are triggered when the nodes are full or nearly empty.

Chapter 7. Limitations

As this paper is done for research purposes, we have to be clear about the limitations of the current implementation. And to make the database ready to be used commercially, there are a lot of things that should be done to make that happen.

- The types of fields implemented are 3: string, number, and boolean fields.

- Reading from disk is very expensive in the current implementation, because there is no index implemented on the disk.

- The number of concurrent connections with the database is limited to the number of threads available on the machine running the database.

- The database was implemented on a Linux-based operating system, and was not tested on other operating systems.

- In the current implementation, the whole index should fit in memory. Which is expected as we are implementing an in-memory database. But other document databases have the option to load part of the index to the memory especially when the data is too big to fit in the memory.

Chapter 8. Future Work

Database systems undergo many enhancements and improvements. And this database is no exception. The current implementation is only used to learn the concepts of Document Databases, but if the database is to be used commercially, it can be enhanced more. And in this chapter, we will discuss some of these enhancements as part of the future work.

Concurrency

When a database resource like a collection or a document is accessed by multiple connections or users at the same time. Read requests are not much of a problem. But what would happen if two users decided to change database resources at relatively the same time (here, we assume access time to resources for both users are a couple of milliseconds difference).

Concurrency introduces multiple issues with data. Concurrency control can be implemented to prevent clients from modifying the same data.

Horizontal Scaling

Horizontally scale the database to handle more load and serve more clients to increase availability.

Data Sharding

Distributed data storage on multiple nodes in the database network.

On-Disk Indexing

For faster data access and flexible storage.

Building Native Drivers for Programming Languages

To enable developers to use the database programmatically like MySQL and other databases.

References

- SQLite. Database File Format. Website: https://www.sqlite.org/fileformat.html

- Couchdb. The Power of B-trees. Website: https://guide.couchdb.org/draft/btree.html

- Kernel. 2019. Ext4 Disk Layout. Website: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

- Pertov, A. 2019, November. Database Internals: A Deep Dive into How Distributed Data Systems Work.