NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC3020 / CZ4031 : Database System Principles

Project 1 Report

Github Link For Source Code: https://github.com/karthikstar/SC3020_Project1

Group 17

Team Members:

Amabel Lim Hui Xin (U2121078D)

Elangovan Karthikeyan (U2120949D)

Tan Puay Jun, Klaus (U2122365E)

Tay Jia Ying, Denise (U2122458K)

All members contributed equally to this project.

1 Introduction

1.1 Project Overview

This report focuses on the design and implementation of a database management system's storage and indexing component using the Java programming language. The key feature of this component is the integration of a B+ tree index structure for efficient data storage and retrieval.

1.2 Project Structure

The project is organised into three main packages:

Utils
This package includes a `DataInitialiser` class which initializes data from a txt file, parses it, and prepares it for storage within the database.

Database
This package manages the storage infrastructure of the database management system. It handles data blocks, records, addresses, and disk management to ensure optimal storage and retrieval of data.

BplusTree
This package contains the core components related to the B+ tree implementation. It includes classes and methods for B+ tree construction, node management, and support for search, insertion, and deletion operations.

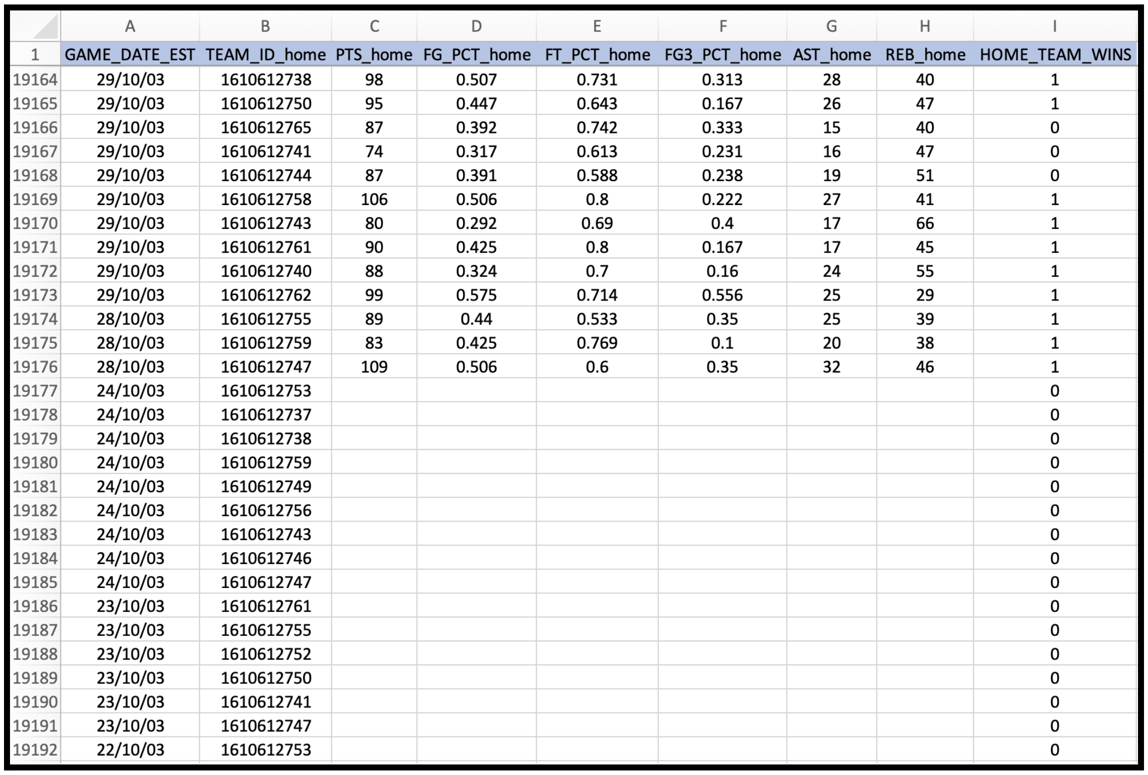
2 Storage Design and Considerations

2.1 Assumptions

According to the project description, the following assumptions are made for the storage component:

- a fraction of main memory is allocated to be used as disk storage for simplicity
- the disk capacity could be 100 - 500 MB
- the disk storage is organized and accessed with a block as a unit
- the block size is set to be 400 B

2.2 Database File



	A	B	C	D	E	F	G	H	I
1	GAME_DATE_EST	TEAM_ID_home	PTS_home	FG_PCT_home	FT_PCT_home	FG3_PCT_home	AST_home	REB_home	HOME_TEAM_WINS
19164	29/10/03	1610612738	98	0.507	0.731	0.313	28	40	1
19165	29/10/03	1610612750	95	0.447	0.643	0.167	26	47	1
19166	29/10/03	1610612765	87	0.392	0.742	0.333	15	40	0
19167	29/10/03	1610612741	74	0.317	0.613	0.231	16	47	0
19168	29/10/03	1610612744	87	0.391	0.588	0.238	19	51	0
19169	29/10/03	1610612758	106	0.506	0.8	0.222	27	41	1
19170	29/10/03	1610612743	80	0.292	0.69	0.4	17	66	1
19171	29/10/03	1610612761	90	0.425	0.8	0.167	17	45	1
19172	29/10/03	1610612740	88	0.324	0.7	0.16	24	55	1
19173	29/10/03	1610612762	99	0.575	0.714	0.556	25	29	1
19174	28/10/03	1610612755	89	0.44	0.533	0.35	25	39	1
19175	28/10/03	1610612759	83	0.425	0.769	0.1	20	38	1
19176	28/10/03	1610612747	109	0.506	0.6	0.35	32	46	1
19177	24/10/03	1610612753							0
19178	24/10/03	1610612737							0
19179	24/10/03	1610612738							0
19180	24/10/03	1610612759							0
19181	24/10/03	1610612749							0
19182	24/10/03	1610612756							0
19183	24/10/03	1610612743							0
19184	24/10/03	1610612746							0
19185	24/10/03	1610612747							0
19186	23/10/03	1610612761							0
19187	23/10/03	1610612755							0
19188	23/10/03	1610612752							0
19189	23/10/03	1610612750							0
19190	23/10/03	1610612741							0
19191	23/10/03	1610612747							0
19192	22/10/03	1610612753							0

Figure 1: Examples of missing values

As shown above, within the dataset, there exists missing values for some of the attributes in the records. In such cases, we have chosen to **ignore** these rows completely.

2.3 Record

For this dataset, there are 9 attributes to be considered: GAME_DATE_EST, TEAM_ID_home, PTS_HOME, FG_PCT_home, FT_PCT_home, FG3_PCT_home, AST_home, REB_home and HOME_TEAM_WINS.

GAME_DATE_EST

For the GAME_DATE_EST attribute, we chose to store it as the long data type. In terms of storage efficiency in Bytes, storing a date as a long is generally more efficient than using the java.util.Date class. Long only uses 8 Bytes, while java.util.Date uses approximately 40 Bytes or more in total for object overhead and the java.util.Date object itself.

Attribute	Data Type	Size
GAME_DATE_EST	long	8 Bytes
TEAM_ID_home	int	4 Bytes
PTS_HOME	int	4 Bytes
FG_PCT_home	float	4 Bytes
FT_PCT_home	float	4 Bytes
FG3_PCT_home	float	4 Bytes
AST_home	int	4 Bytes
REB_home	int	4 Bytes
HOME_TEAM_WINS	int	4 Bytes

Figure 2: Characteristics of each attribute

Based on this allocation, the size of each record would be 40 bytes.

2.4 Block

As a block size is set to be 400 bytes, and the size of one record is 40 bytes, each block would be able to hold exactly 10 records. No space will be wasted in this block based on the current arrangement of records as well. This can be visualized with the diagram below:

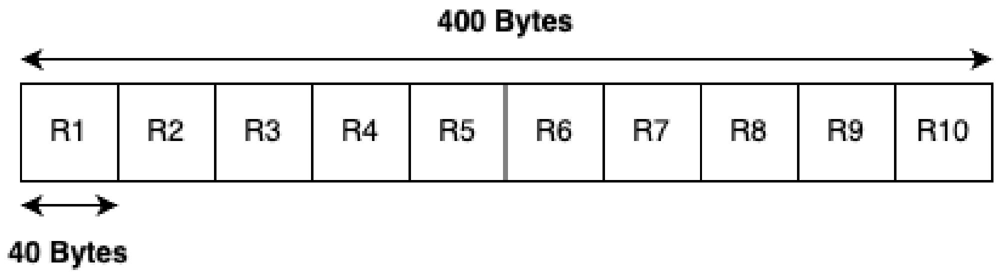


Figure 3: Block structure

2.5 Disk

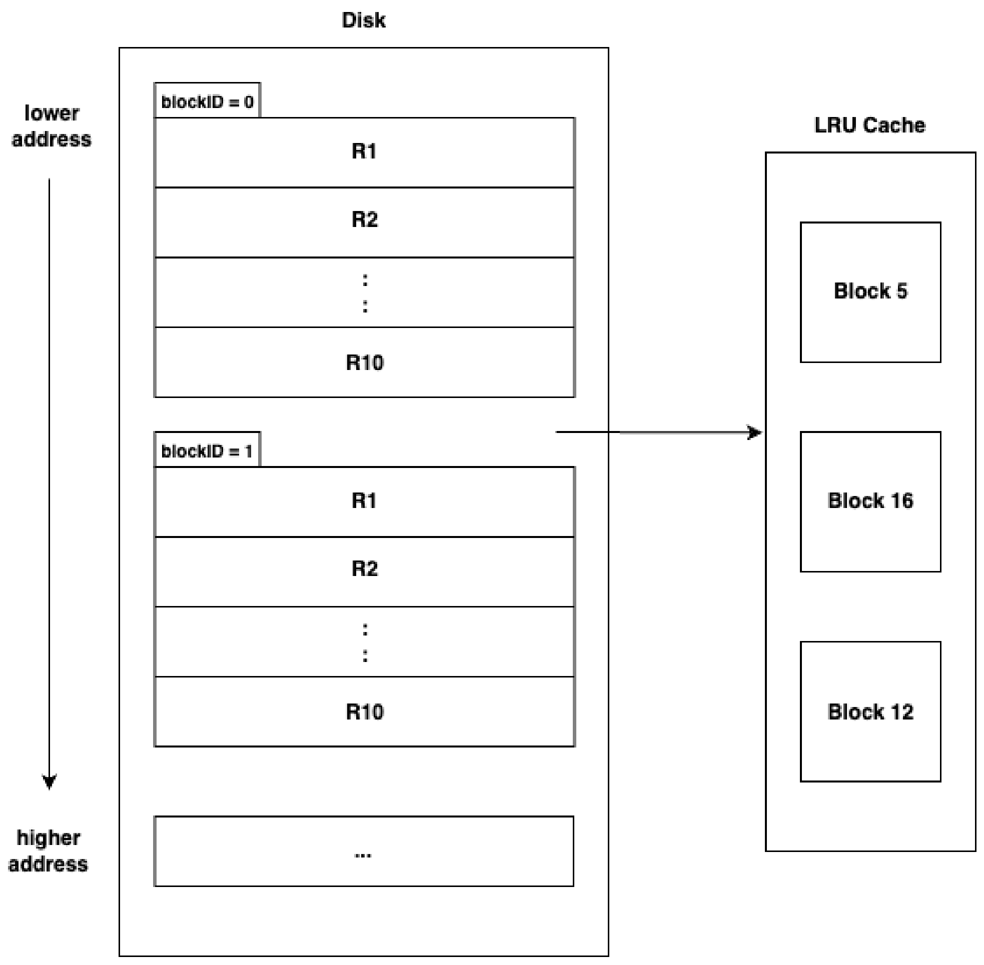


Figure 4: Disk Structure

The disk storage is organised and accessed within a block as a unit. When retrieving a record for a query, the system will first check if the block containing the specific record is available in the Least Recently Used (LRU) cache. If it is found in the cache, the system will directly retrieve the required block from the cache. Otherwise, the system will have to retrieve the block from the disk, and update the block in the cache accordingly.

We have chosen a disk capacity of **500MB** to run our experiments.

2.6 Additional Design Considerations

2.6.1 Spanned vs Non-spanned

Spanned records can span multiple disk blocks to utilise the unused space in each block. On the other hand, unspanned records do not cross block boundaries and each record is contained within a single block.

As each of our records uses 40 bytes while the size of each block is 400 bytes, each block holds 10 records perfectly – without leaving any space unused. Hence, our records are naturally unspanned, but without any wastage of space. This allows for easy searching of records within the blocks, as only one block has to be accessed when searching for a record.

2.6.2 Fixed vs Variable Length

We have chosen to store the records using the fixed length method due to the following reasons:

Predictable memory usage and ease of implementation
Fixed-length structures have a consistent size, which makes memory allocation and management more predictable and less complex, therefore easier to implement.

Nature of dataset
Variable length method is commonly used when dealing with a large amount of highly variable-length data such as text descriptions, where substantial space savings can be realized. In our dataset, each field has a predefined meaning and size, which remains constant throughout the dataset. The fields consist mainly of numeric values representing game statistics, and the data's variability is minimal.

2.6.3 Sequencing vs Non-sequencing

We chose to keep the records in the blocks in a non-sequenced manner, as it increases the speed of insertion and deletion of records.

2.6.4 LRU Caching

Least Recently Used (LRU) caching is a technique that improves data access efficiency and reduces disk block accesses.

We have chosen to use LRU caching due to the ease of its implementation with Java's `LinkedHashMap` class. The `LinkedHashMap` combines the advantages of a hash table, which provides rapid access and retrieval of elements, with those of a linked list, which ensures the preservation of the order in which elements were added.

Optimizing Cache Space
In LRU cache, a fixed amount of memory is allocated to store data items. When the limit is reached, the item positioned at the end of the list (least recently used) is removed, to make room for the incoming item.

Efficient Data Access
When a data item within a block is accessed, the entire block is moved to the front of the list. This signifies that the block has been used most recently.

Minimizing Disk Access
Frequently used blocks are consistently positioned near the front of the list and are less likely to be removed from the cache. This reduces the number of disk block accesses required when querying a record.

3 Indexing Component (B+ Tree) Design

A B+ Tree is a self-balancing tree structure that can handle range queries, which enables efficient storage and retrieval of data. We have implemented our B+ Tree in the `Tree` class, which relies upon the `Node`, `NonLeafNode` and `LeafNode` classes.

3.1 Node

The Node class represents a generic node in the B+ tree, and it includes common properties and methods that are essential for both leaf and non-leaf nodes within the tree.

Class Structure

Attributes:

`minLeafNodeSize`

The minimum number of keys a node must have to be a valid leaf node, calculated by $\lfloor \frac{n+1}{2} \rfloor$, where n is the maximum number of keys in a node.

`minNonLeafNodeSize`

The minimum number of keys a node must have to be a valid non-leaf node, calculated by $\lfloor \frac{n}{2} \rfloor$, where n is the maximum number of keys in a node.

`isRoot`

Boolean flag that indicates whether a node is a root node.

`isLeaf`

Boolean flag that indicates whether a node is a leaf node.

`nodeSizeLimit`

Represents the maximum number of keys that a node can accommodate.

`NonLeafNode parent`

A reference that points to the parent node.

`ArrayList<Integer> keys`

An ArrayList which stores the keys within the node, where the number of keys is subjected to the node size limit. ArrayList is chosen over traditional arrays mainly due to the dynamic resizing properties, which help to simplify the implementation and maintenance of a B+ tree data structure due to the nature of frequent insertion and deletion of keys.

3.2 LeafNode

The `LeafNode` class represents a leaf node in the leaf level of the tree. It extends the Node class (inherits its fundamental attributes and behaviors) and includes additional attributes and methods specific to non-leaf nodes. It is responsible for storing records or data associated with keys. Each LeafNode contains a TreeMap for mapping keys to records and an ArrayList for storing the actual records.

Class structure

Attributes:

`TreeMap<Integer, ArrayList<Address>> mapping`

The TreeMap is used to map each key to a list of associated records. The key is an integer, and the value is an ArrayList of Address objects. This structure allows for efficient retrieval of records associated with a given key.

`ArrayList<Address> records`

This ArrayList holds the actual records. Each record is represented by an Address object. Records are stored in the order they were inserted into the leaf node.

`LeafNode rightNode`

This is a reference to the right sibling leaf node, which helps to facilitate navigation to the next leaf node.

`LeafNode leftNode`

This is a reference to the left sibling leaf node. Similar to the rightNode, it helps navigate to the previous leaf node in the linked list.

3.3 NonLeafNode

The `NonLeafNode` class represents non-leaf nodes in the B+ tree and extends the Node class (inherits its fundamental attributes and behaviors) and includes additional attributes and methods specific to non-leaf nodes.

Class structure

Attributes:

`ArrayList children`

This arraylist contains the child nodes of the current node.

3.4 Tree

The `Tree` class relies on the `Node`, `NonLeafNode`, and `LeafNode` classes to manage the structure, insertion, deletion, and retrieval of data in the B+ tree.

3.4.1 Insertion

The explanation of key functions is as follows:

1) **Initialization**

   `createTreeNode`

   Creates a new root node for the B+ tree

2) **Insertion Request**

   `insertRecord`

   Initiates the insertion process by inserting a record with the specified key and address

3) **Locate Leaf Node**

   `retrieveLeafToInsert`

   Finds the appropriate leaf node for insertion based on the key

4) **Insertion into Leaf Node**

   `insertRecord`

   1. If Leaf Node is empty
      - Create an ArrayList to store records, a TreeMap to map keys to records, and an ArrayList to store keys.
      - Add the new record, map the key to this record, and insert the key into the list of keys

   2. If key already exists in Leaf Node (in both mapping and list of keys)
      - Add the new address to the existing list of addresses associated with that key

3. If Leaf Node is not full

- Create a new ArrayList to store records
- Add the new record into the newly created ArrayList
- Update the mapping to associate the key to the newly created ArrayList
- Insert the key into the list of keys

4. If Leaf Node is full (splitting required)

- Call `splitLeafNode` method which splits a leaf node into two leaf nodes and inserts the new node into the parent.
- If the parent is full, `splitNonLeafNodes` method is recursively called, which splits a non-leaf node and propagates the split up to the root node if necessary.

3.4.2 Deletion

The explanation of key functions is as follows:

1) **Deletion Request**

   `removeRecord`

   Initiates the deletion process for a record with the specified key

2) **Locate Leaf Node**

   `findLowerBoundKey`

   Locates the appropriate leaf node for deletion based on the key

3) **Deletion from Leaf Node**

   `removeNode`

   Removes the key and addresses from a leaf node and triggers tree balancing if needed

4) **Maintenance of B+ tree structure**

   `fixInvalidTree`

   Maintains tree balance after deletion by potentially merging or borrowing nodes

   `fixInvalidRoot`

   Handles root node adjustments after deletion

   `fixInvalidLeaf`

   Manages the balancing of leaf nodes during deletion

   `fixInvalidNonLeaf`

   Manages the balancing of non-leaf nodes during deletion

3.4.3 Search

The explanation of key functions is as follows:

1) **Search Request**

   `searchKey`

   Initiates a key-based search starting from a specified node

2) **Search Values**

   `searchValuesInRange`

   Recursive calls to `searchKey` and `searchValuesInRange` traverses the tree, following keys or checking ranges until the appropriate leaf node(s) is found

3.5 Additional Considerations

3.5.1 Calculation of Maximum Number of Keys a Node Holds, n

In the design and instantiation of our B+ tree, it is essential to compute the n attribute, which represents the maximum number of keys a node can hold. This attribute plays an important role in balancing the trade-off between node size and tree depth, ultimately affecting the performance of our indexing component.

The n attribute can be calculated using the following formula:

$$Max\ number\ of\ keys,\ n\ = \frac{block\ size - overhead}{pointer\ size + key\ size}$$

In our implementation,
Block size = 400 bytes (as given in instructions)
Overhead = 8 bytes
Pointer size = 8 bytes (assume 64-bit JVM system)
Key size = 4 bytes (key is a float)

Calculation of n:

$$Max\ number\ of\ keys,\ n\ = \frac{400 - 8}{8 + 4} = \ 32$$

3.5.2 Handling of Duplicate Keys

In our B+ tree implementation, duplicate keys are handled by allowing multiple records to be associated with the same key in the LeafNode with the usage of **`TreeMap<Integer, ArrayList<Address>> mapping`**.

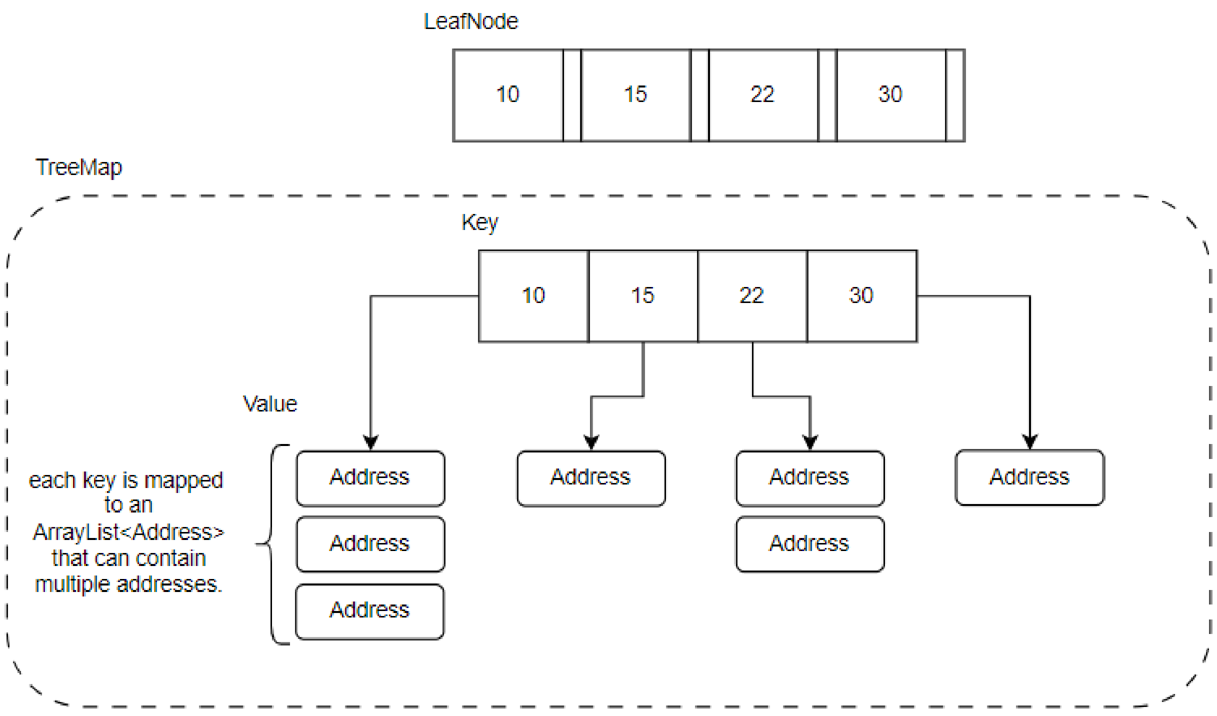


Figure 5: Illustration of handling of duplicate keys

Insertion of records with duplicate keys

When record with existing key is inserted:

1. Check if the key is in both the mapping and keys
2. If yes, it means there are already records associated with that key in the LeafNode. Then we add the new record to the existing list of records for that key using
3. If not in keys but in mapping, it means that this key was moved to a new LeafNode during a split operation, then we add the key to keys ArrayList and associate the new record.

Retrieval of records with duplicate keys

1. `getAddressesPointedByKey` method is called with a specific key as its argument
2. The method looks up the key in the mapping TreeMap using and returns the list of Address objects associated with that key. This list includes all records associated with duplicate keys, effectively handling duplicates during the retrieval process.

4 Experiment Results

We have chosen a disk capacity of **500 MB** to run the following experiments.

4.1 Experiment 1

Initial number of lines in dataset txt file (excl. line with column names)	26651
Lines with missing values that were ignored	99
Total number of records loaded into database	26552

As mentioned in 2.2, rows with missing values are dropped. Hence, although the total number of records read is 26651, since 99 rows containing missing values were dropped, we have a refined total number of 26552 records in our cleaned dataset that are loaded into the database.

No. of records	26552
Size of a record	40 B
No. of records stored in a block	10
No. of blocks used for storing the data	2656

4.2 Experiment 2

Parameter n of B+ tree						32				
No. of nodes of B+ tree						24				
No. of levels of B+ tree						2				
Content of root nodes										
0.295	0.308	0.321	0.337	0.351	0.369	0.387	0.412	0.433	0.446	0.457
0.474	0.486	0.506	0.533	0.544	0.558	0.568	0.591	0.603	0.614	0.633

4.3 Experiment 3

No. of index nodes accessed	2
No. of data blocks accessed	719
Average of 'FG3_PCT_home'	0.39
Running time (nanoseconds)	182292
No. of data blocks accessed by Brute Force method	2656
Running time by Brute Force method (nanoseconds)	32355834
Decrease in no. of data blocks accessed due to cache hit	114

The running time is measured by using `System.nanoTime()` to find the start time before calling the `searchKey()` method and the end time after the results are returned, then calculating the difference between the start time and the end time.

4.4 Experiment 4

No. of index nodes accessed	7
No. of data blocks accessed	767
Average of ‘FG3_PCT_home’	0.53
Running time (nanoseconds)	5874125
No. of data blocks accessed by Brute Force method	2656
Running time by Brute Force method (nanoseconds)	32544000
Decrease in no. of data blocks accessed due to cache hit	145

4.5 Experiment 5

No. of nodes of B+ tree	19									
No. of levels of B+ tree	4									
Keys to be deleted										
0.25, 0.257, 0.266, 0.269, 0.269, 0.274, 0.275, 0.277, 0.277, 0.278, 0.279, 0.279, 0.282, 0.283, 0.284, 0.286, 0.288, 0.288, 0.289, 0.289, 0.29, 0.291, 0.292, 0.293, 0.293, 0.293, 0.294, 0.298, 0.299, 0.301, 0.302, 0.303, 0.304, 0.305, 0.306, 0.307, 0.308, 0.309, 0.31, 0.311, 0.312, 0.313, 0.314, 0.315, 0.316, 0.317, 0.318, 0.319, 0.32, 0.321, 0.322, 0.323, 0.324, 0.325, 0.326, 0.328, 0.329, 0.33, 0.333, 0.337, 0.338, 0.339, 0.34, 0.341, 0.342, 0.343, 0.344, 0.345, 0.346, 0.347, 0.348, 0.349, 0.35										
Content of root nodes										
0.369	0.387	0.412	0.433	0.446	0.457	0.474	0.486	0.506	0.533	0.544
0.558	0.568	0.591	0.603	0.614	0.633					
Running time (nanoseconds)	12964584									
No. of data blocks accessed by Brute Force method	2230									
Running time by Brute Force method (nanoseconds)	16840750									
Decrease in no. of data blocks accessed due to cache hit	277									

5 Installation Guide

Prerequisites

1. Ensure that you have Java Development Kit (JDK) installed on your system.
2. Verify that you have a compatible Integrated Development Environment (IDE) installed (preferably IntelliJ IDEA)

Project Setup

3. Git Clone or download the project's source code from the project repository from the following Github Link : https://github.com/karthikstar/SC3020_Project1

Run project

4. Install SDK - 18 (Oracle Open JDK version 18.0.1)
5. Open the Main.java file (located in src folder) in the preferred IDE, and run the Main.java file.

Usage

6. Experiment results can be viewed on the output in the terminal window.