Table of contents

List of figures

List of Tables

Abstract

This is a research-based project and the basic point motivating this project is learning and implementing algorithms that reduces time and space complexity.

In the first part of the project, we reduce the time taken to search a given record by using a B/B+ tree rather than indexing and traditional sequential access. It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system. Therefore, the objective is to minimize the number of disk accesses and thus, this project is concerned with techniques for achieving that objective i.e. techniques for arranging the data on a disk so that any required piece of data, say some specific record, can be located in a few I/O's as possible.

In the second part of the project, Dynamic Programming problems were solved with Recursion, Recursion With Storage, Iteration with Storage, Iteration with Smaller Storage. The problems which have been solved in these 4 variations are Fibonacci, Count Maze Path, Count Board Path, and Longest Common Subsequence. All the 4 variations are an improvement over one another and thus time and space complexity are reduced significantly as we go from Recursion to Iteration with Smaller Storage.

Chapter 1

Introduction

B/B+ trees are extensively used in Database Management Systems because search operation is much faster in them compared to indexing and traditional sequential access. Moreover, in DBMS, B+ tree is used more as compared to B-Tree. This is primarily because unlike B-trees, B+ trees have a very high fan out, which reduces the number of I/O operations required to find an element in the tree. This makes the insertion, deletion, and search using B+ trees very efficient. However, the indexing of column to be searched is also efficient but the downside of it is that when searching is to be done on large collections of data records, it becomes quite expensive, because each entry in B/B+tree requires us to start from the root and go down to the appropriate leaf page. This operation takes only O(log n) time. Hence we would also like to implement the efficient alternative, B+ tree.

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees, it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where his the height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since it is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Chapter 2

Problem Description and objectives

2.1 Problem Description

Given a collection of data records, we want to create a B+ tree index on some key field.

A. One approach is to insert each record into an empty tree. It is less expensive as compared to the indexing of that key field because each entry in the B/B+ tree requires us to start from the root and go down to the appropriate leaf page.
B. An efficient alternative to indexing is to use B/B+ tree. Hence, we want to implement a B+ tree and compare its performance with normal l indexing, with different inputs and document the results.

2.2 Objective

A. Implement a Table Scan for the relational database management System.
B. Implement Index Seek for Relational Database Management System .
C. Compare the efficiency of Index Seek and Table Scan in the relational database.
D. Implement Multilevel Indexing in Relational Database Management System
E. Implement B+Tree for better query optimization
F. Better search in less time.
G. Enhancing user efficiency.
H. Creates a connection by making it easy for others to plug into the platform.

Chapter 3

DBMS File Organisation

Relative data and information are stored collectively in file formats. A file is a sequence of records stored in binary format. A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks.

File Organization defines how file records are mapped onto disk blocks. We have four types of File Organization to organize file records −

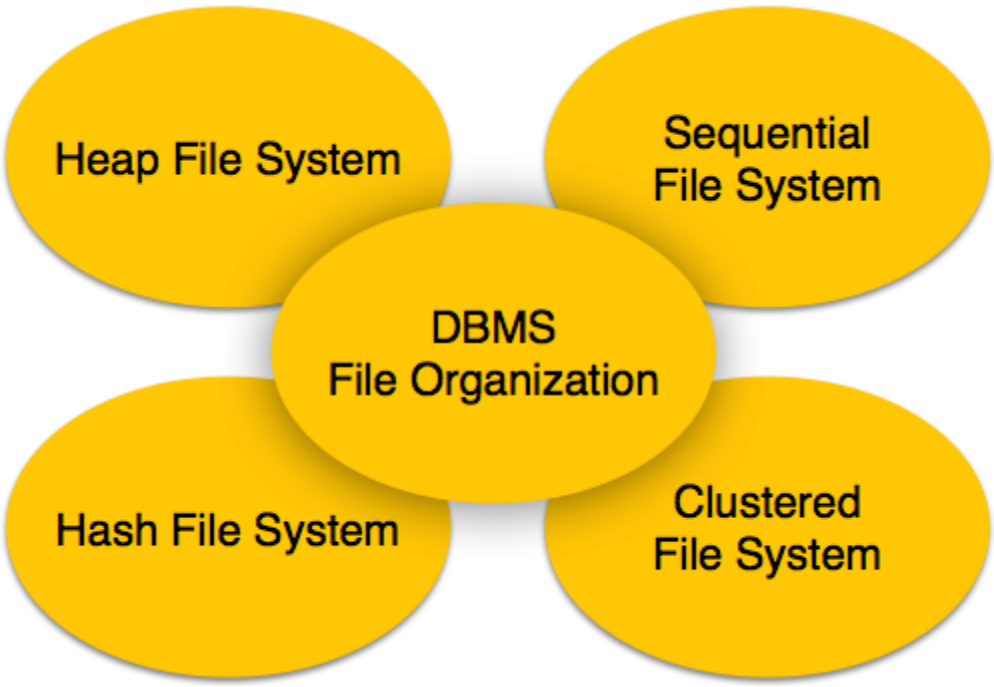

Fig 3.1 Database File System

Reference(www.javatpoint.com)

3.2Heap File Organization

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

3.3 Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In a sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

3.4 Hash File Organization

Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of the disk block where the records are to be placed.

3.5 Clustered File Organization

Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

3.6 File Operations

Operations on database files can be broadly classified into two categories −

3.6.1Update Operations

3*.6.2Retrieval Operations*

Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than the creation and deletion of a file, there could be several operations, which can be done on files.

Open − A file can be opened in one of the two modes, **read mode** or **write mode**. In read mode, the operating system does not allow anyone to alter data. In other words, data is read-only. Files opened in reading mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.

- **Locate** − Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.

- **Read** − By default, when files are opened in reading mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Write** − User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows doing so.
- **Close** − This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
    - removes all the locks (if in shared mode),
    - saves the data (if altered) to the secondary storage media, and
    - releases all the buffers and file handlers associated with the file.

The organization of data inside a file plays a major role here. The process to locate the file pointer to the desired record inside a file various based on whether the records are arranged sequentially or clustered.

3.7 DBMS Data Storage System

Databases are stored in file formats, which contain records. At the physical level, the actual data is stored in an electromagnetic format on some device. These storage devices can be broadly categorized into three types −

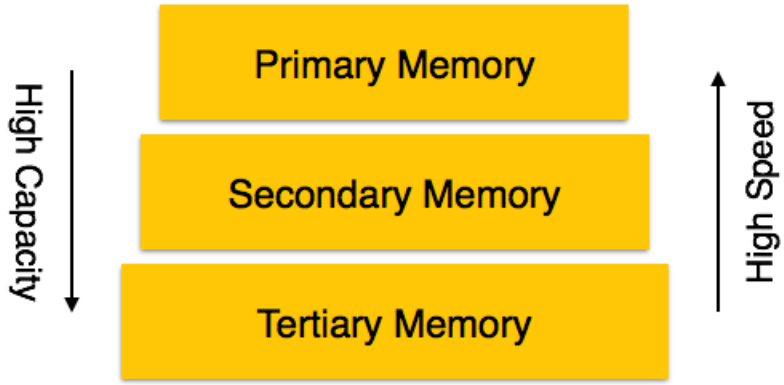

Fig 3.2 Memory Hierarchy

Reference(www.javatpoint.com)

3.7.1Primary Storage − The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires a continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.

3.7.2Secondary Storage − Secondary storage devices are used to store data for future use or as a backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.

3.7.3Tertiary Storage − Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

Chapter 4

Study of Indexing

4.1 Indexing

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a storage structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types −

- PrimaryIndex − Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation.
- Secondary Index − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- Clustering Index − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types −

- Dense Index
- Sparse Index

4.1.1Dense Index

In the dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

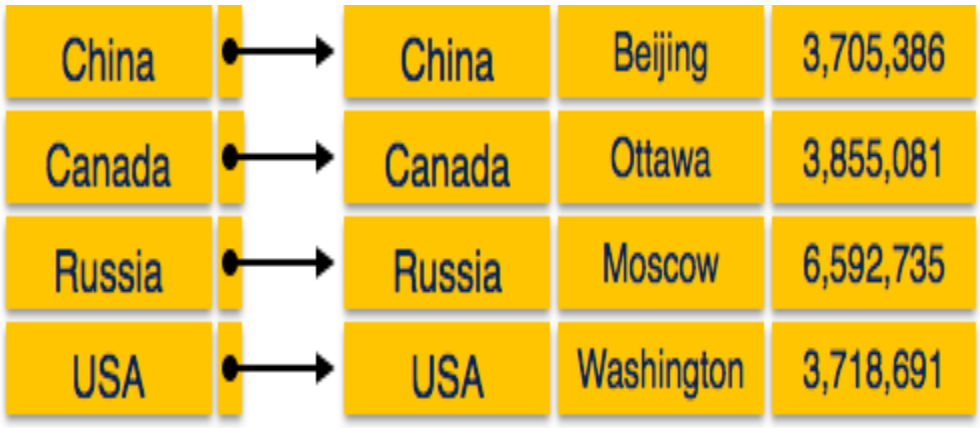

Fig 4.1 Dense Indexing

Reference(https://en.wikipedia.org/wiki/B)

4.1.2Sparse Index

In the sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts the sequential search until the desired data is found.

4.1.3 Multilevel Index

Index records comprise search-key values and data pointers. The multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If a single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.

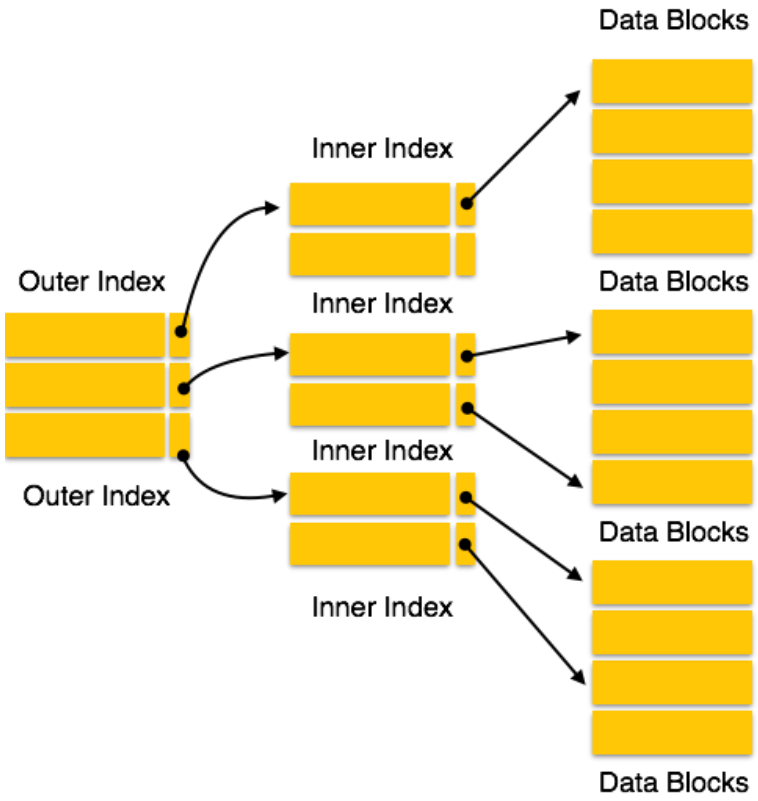

Fig 4.2 Multilevel Indexing

Reference(https://en.wikipedia.org/wiki/B)

Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

Index records comprise search-key values and data pointers. The multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If a single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.

Chapter 5

B+ Tree

5.1 Introduction

A simple B+ tree example linking the keys 1–7 to data values d1-d7. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is different.

A B+ tree is an N-ary tree with a variable but often a large number of children per node. A B+ tree consists of a root, internal nodes, and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node,[1] typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

The ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories. NTFS uses B+ trees for directory and security-related metadata indexing. EXT4 uses extent trees (a modified B+ tree data structure) for file extent indexing. Relational database management systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8 Sybase ASE, and SQLite support this type of tree for table indices. Key-value database management systems such as CouchDB and Tokyo Cabinet support this type of tree for data access.

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b = 7$	Example $b = 100$
Root Node (when it is the only node in the tree)	Records	1	$b - 1$	1–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil$	b	4–7	50–100

Table 5.1 Node v/s child

5.2 Structure

Every leaf node is at an equal distance from the root node. A B^+ tree is of the order **n** where **n** is fixed for every B^+ tree.

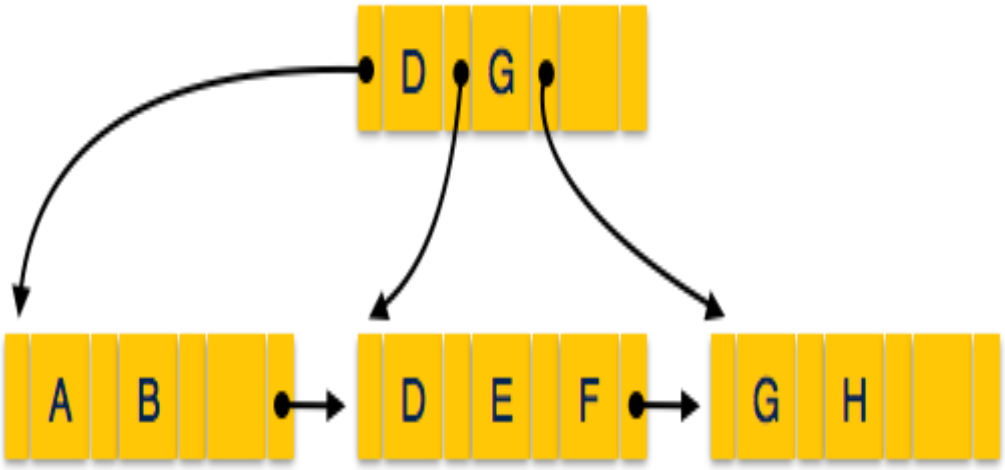

Fig 5.1 B+Tree Structure

Reference**(https://en.wikipedia.org/wiki/B)**

Internal Nodes

A. Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
B. At most, an internal node can contain **n** pointers.

Leaf Nodes

A. Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.
B. At most, a leaf node can contain **n** record pointers and **n** key values.
C. Every leaf node contains one block pointer **P** to point to the next leaf node and forms a linked list.

The order, or branching factor, b of a B+ tree measures the capacity of nodes (i.e., the number of children nodes) for internal nodes in the tree.

The actual number of children for a node, referred to here as m, is constrained for internal nodes.

The root is an exception: it is allowed to have as few as two children. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children but are constrained that the number of keys must be at least.

In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary and at most.

5.3Properties:

A. All leaves have to be in the same way that they between are at the same level.

B. A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk

C. Every node except root must contain at least t-1 keys. The root may contain the minimum key.

D. All nodes (including root) may contain at most 2t – 1 keys.
A number of children of a node are equal to the number of keys in it plus 1.
All keys of a node are sorted in increasing order. The child between two keys k1 and

E. k2 contains all keys in the range from k1 and k2.
B-Tree grows and shrinks from a root which is unlike Binary Search Tree.

F. Binary Search Trees grow downward and also shrink from downward.
Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

5.4 Algorithms

5.4.1 Search

A. The root of a B+ Tree represents the whole range of values in the tree, where every internal node is a subinterval.

B. We are looking for a value k in the B+ Tree.

C. Starting from the root, we are looking for the leaf which may contain the value k.

D. At each node, we figure out which internal pointer we should follow.

E. An internal B+ Tree node has at most $\leq$ children, where every one of them represents a different sub-interval.

F. We select the corresponding node by searching on the key values of the node.

5.4.2 Insertion

G. Perform a search to determine what bucket the new record should go into.

H. If the bucket is not full (at most entries after the insertion), add the record.

I. Otherwise, before inserting the new record split the bucket.

J. original node has $\lceil (L+1)/2 \rceil$ items

K. new node has $\lfloor (L+1)/2 \rfloor$ items

L. Move $\lceil (L+1)/2 \rceil$-the key to the parent, and insert the new node to the parent.

M. Repeat until a parent is found that need not split.

N. If the root splits, treat it as if it has an empty parent and split as outlined above.

O. B-trees grow at the root and not at the leaves.

5.4.3Deletion

P. Start at the root, find leaf L where entry belongs.

Q. Remove the entry.

R. If L is at least half-full, done

S. If L has fewer entries than it should,

T. If sibling (adjacent node with the same parent as L) is more than half-full, re-distribute, borrowing an entry from it.

U. Otherwise, a sibling is exactly half-full so we can merge L and sibling.

V. If merge occurred, must delete the entry (pointing to L or sibling) from a parent of L.

W. Merge could propagate to root, decreasing height.

5.4.4Prefix key compression

A. It is important to increase fan-out, as this allows to direct searches to the leaf level more efficiently.

B. Index Entries are only to 'direct traffic', thus we can compress them.

5.4.5Bulk-loading

Given a collection of data records, we want to create a B+ tree index on some key field. One approach is to insert each record into an empty tree. However, it is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. An efficient alternative is to use bulk-loading.

A. The first step is to sort the data entries according to a search key in ascending order.

B. We allocate an empty page to serve as the root and insert a pointer to the first page of entries into it.

C. When the root is full, we split the root, and create a new root page.

D. Keep inserting entries to the rightmost index page just above the leaf level, until all entries are indexed.

Note :

E. when the right-most index page above the leaf level fills up, it is split;

F. this action may, in turn, cause a split of the right-most index page on step closer to the root;

G. splits only occur on the right-most path from the root to the leaf level.

Chapter - 6

PROCESS SELECTION

6.1 JAVA

I chose Java because of its following features:

6.1.1 Simple

According to Sun, Java language is simple because :

A. the syntax is based on C++ (so easier for programmers to learn it after C++).
B. removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading, etc.
C. No need to remove unreferenced objects because there is Automatic Garbage Collection in java

6.1.2 Object-oriented

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Fig 6.1 OOP concept

Reference(www.javatpoint.com)

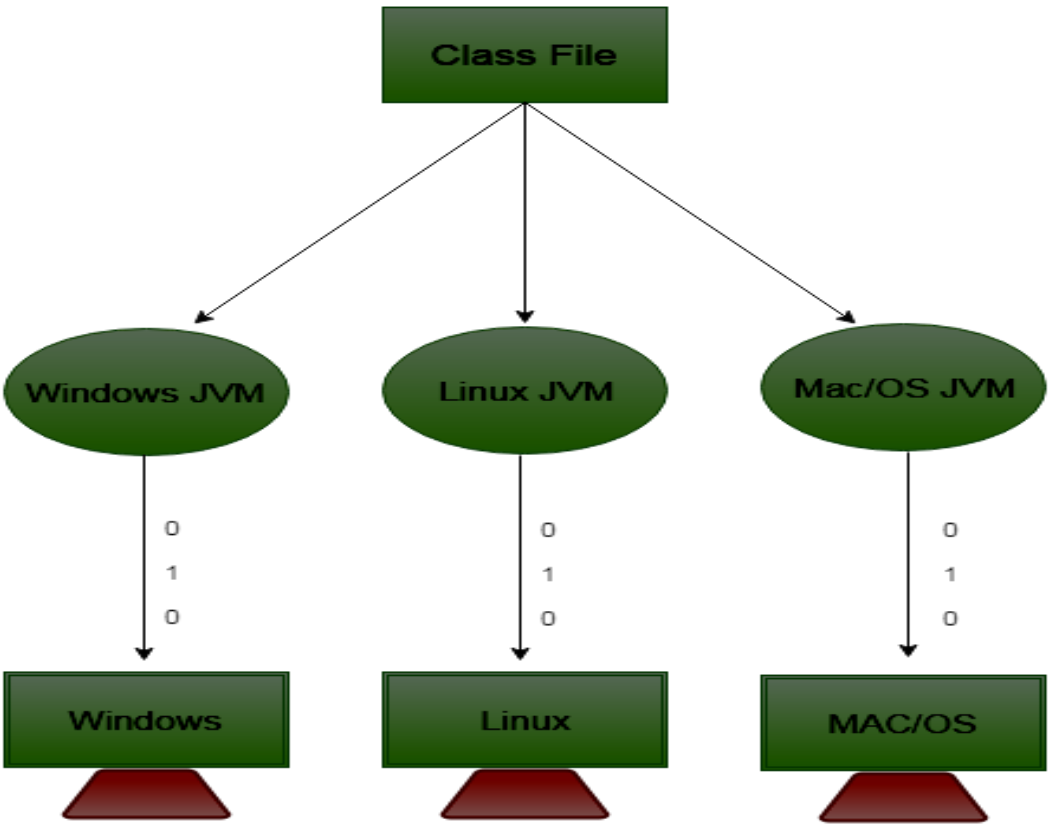

Fig 6.2 Java platform Independency

Reference(www.javatpoint.com)

6.1.3 Secured

Java is secured because:

A. No explicit pointer

B. Java Programs run inside virtual machine sandbox

C. **Class loader**: adds security by separating the package for the classes of the local file system from those that are imported from network

D. **Bytecode Verifier**: checks the code fragments for illegal code that can violate access right to objects.

E. **Security Manager**: determines what resources a class can access such as reading and writing to the local disk.

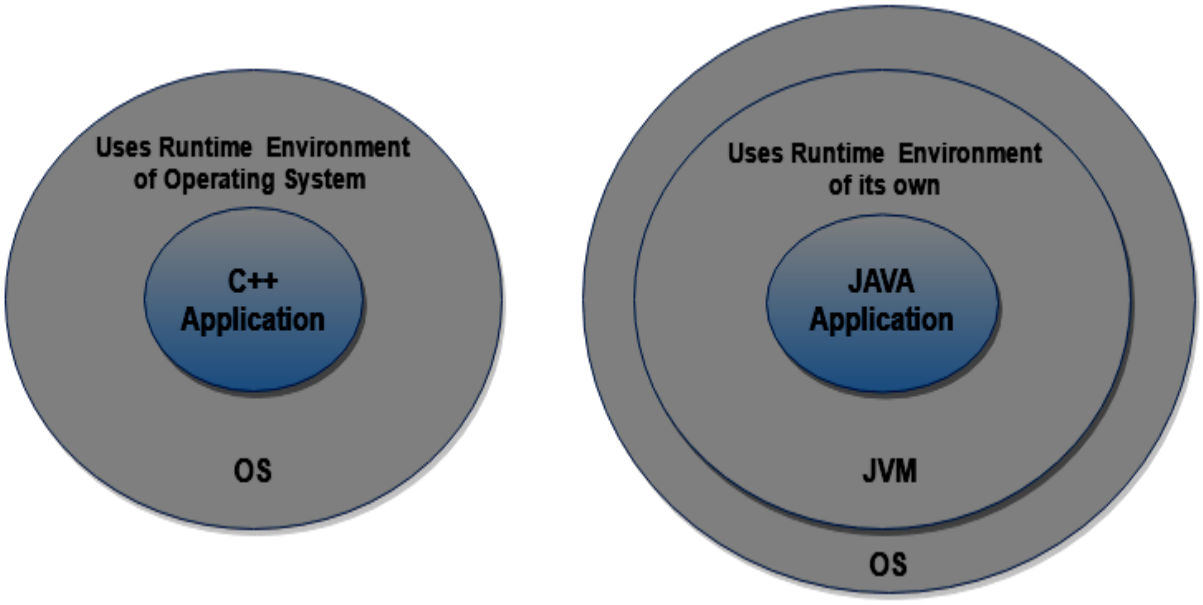

Fig 6.3 Java Security

Reference(www.javatpoint.com)

6.1.4 Architecture-neutral

There is no implementation dependent features e.g. size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64-bit architectures. Hence, this property also makes Java Portable.

Chapter 7

Implementation Details

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+tree to actually provide an efficient structure for housing the data itself (this is described in[4]:238 as index structure "Alternative 1").

If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where.Although theoretically, the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case, a reasonable choice for block size would be the size of the processor's cache line.

The space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the i-th entry of an internal block contains the first key of the block . Instead of storing the full key, we could store the shortest prefix of the first key of block that is strictly greater (in lexicographic order) than the last key of block i. There is also a simple way to compress pointers: if we suppose that some consecutive blocks are stored

contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case, searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression technique is that the number of stored elements may vary considerably from a block to another depending on how well the elements are compressed inside each block.

7.1 Implementation Blocks

7.1.1 Insertion

1) Initialize x as root.

2) While x is not a leaf, do the following

a) Find the child of x that is going to to be traversed next. Let the child be y.

b) If y is not full, change x to point toy.

c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. The else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is a leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

7.1.2 Searching

1) Initialize x as root.

2) While x is not a leaf, do the following:

a.) find the key value which equal to value to search, if found return the index.

b.) otherwise go to the node which may contain value to searched i.e. if the value to be searched is valid, then go to the node which has key values x<val<y.

c.) Recursively call the node.

3) The recursion in step 2 stops when x is leaf and we have not found value to be searched. In that case, we return key not found.

7.2 Code Structure

- 2 classes are made for the BTree implementation: One is the client which is used to run all the functions of class BTree.
- 4 data members are taken for class BTree which are M(for the degree of the BTree), a private class Node which stores array of Entry references which is another private class in BTree , n for the number of key-value pairs(key-value pair is counted as one data value), height for the height of BTree.
- Private class Entry has 3 data members: key, value, and reference of type Node.
- BTree constructor is used to initializing an empty BTree.
- isEmpty() function –returns true if the symbol table is empty.
- size() function returns the number of key-value pairs
- height() function returns the height of the tree(useful for debugging).
- get() function returns the value associated with the key that is passed as the parameter.
- get() function is implemented as the algorithm discussed earlier.
- put() function inserts the key-value pair into the symbol table overwriting the old value with the new value if the key is already in the symbol table.
- put() function uses function insert (for the node which is not full) and function split (for insertion in a node which is full) and is implemented as the algorithm discussed earlier.

Chapter 8

Result

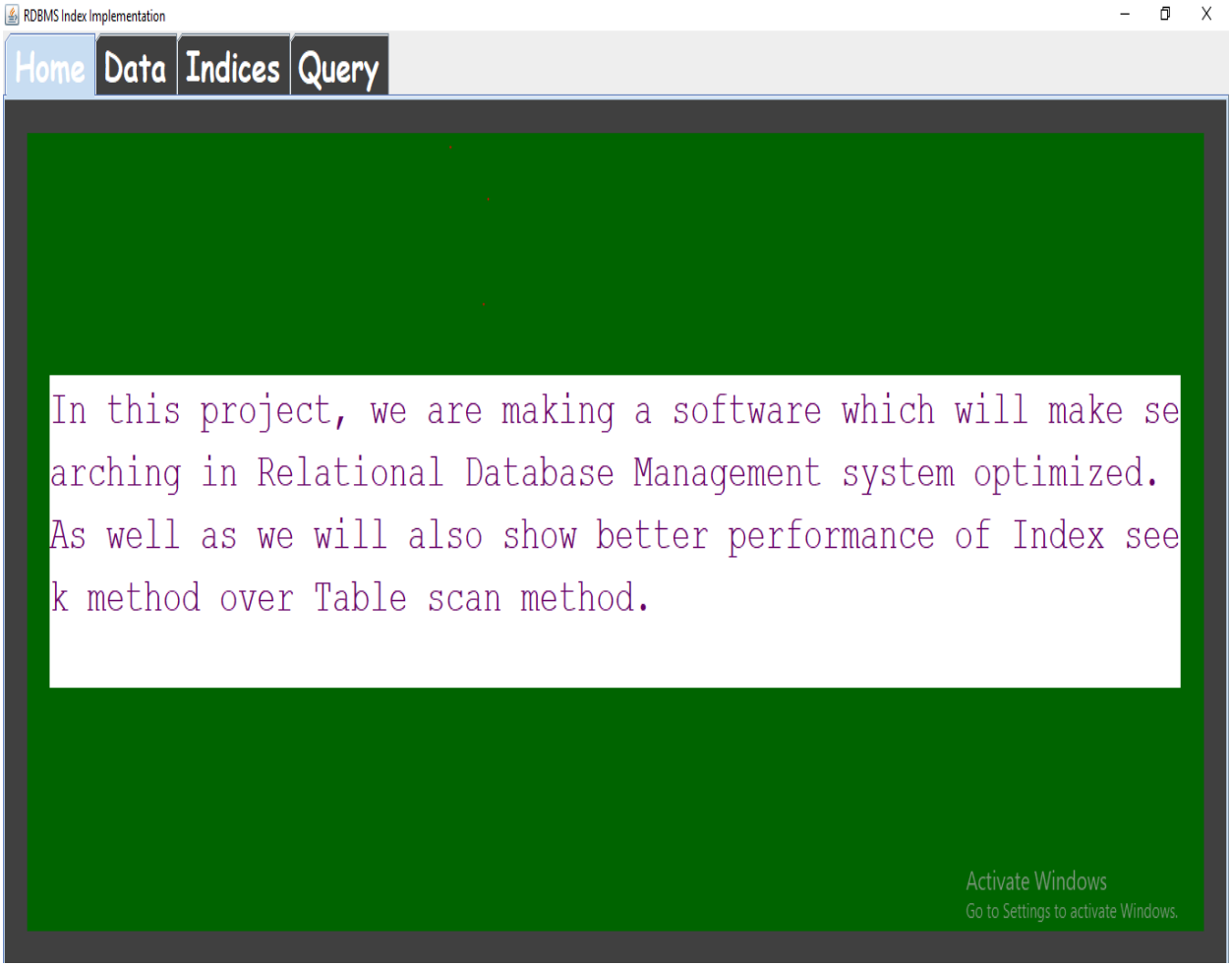

Fig 8.1 Home Page

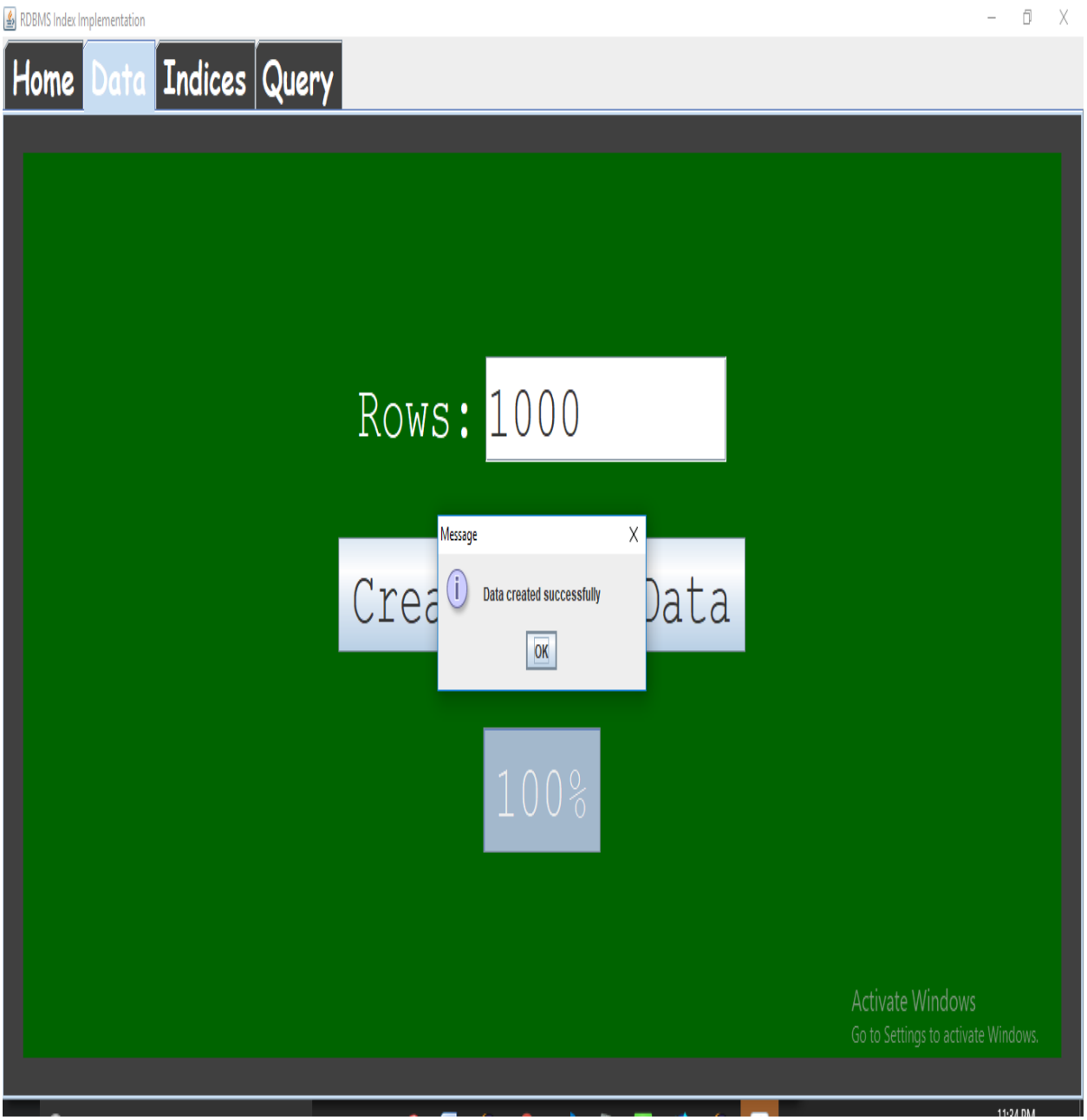

Fig 8.2 Data Page

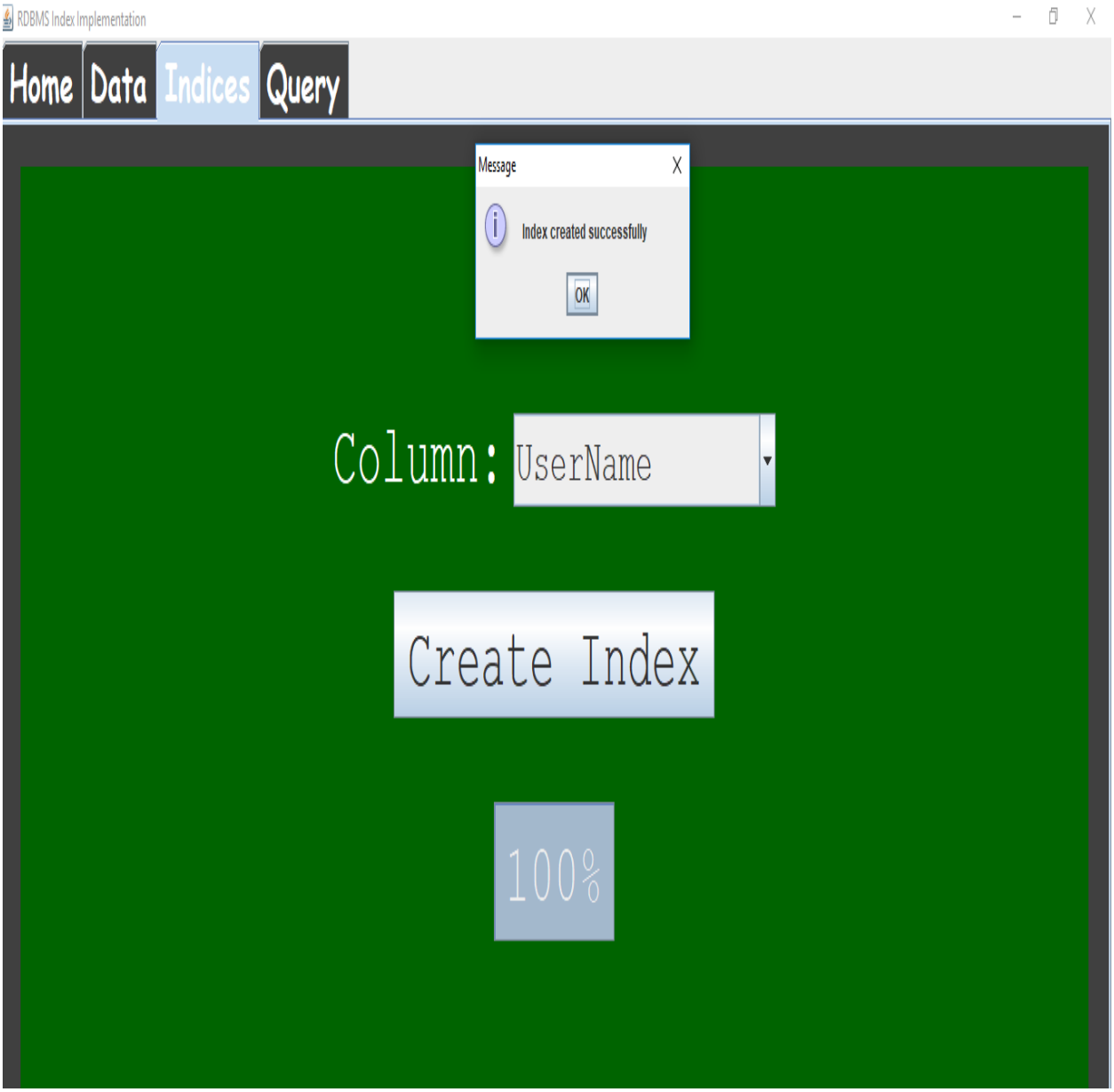

Fig 8.3 Indices Page

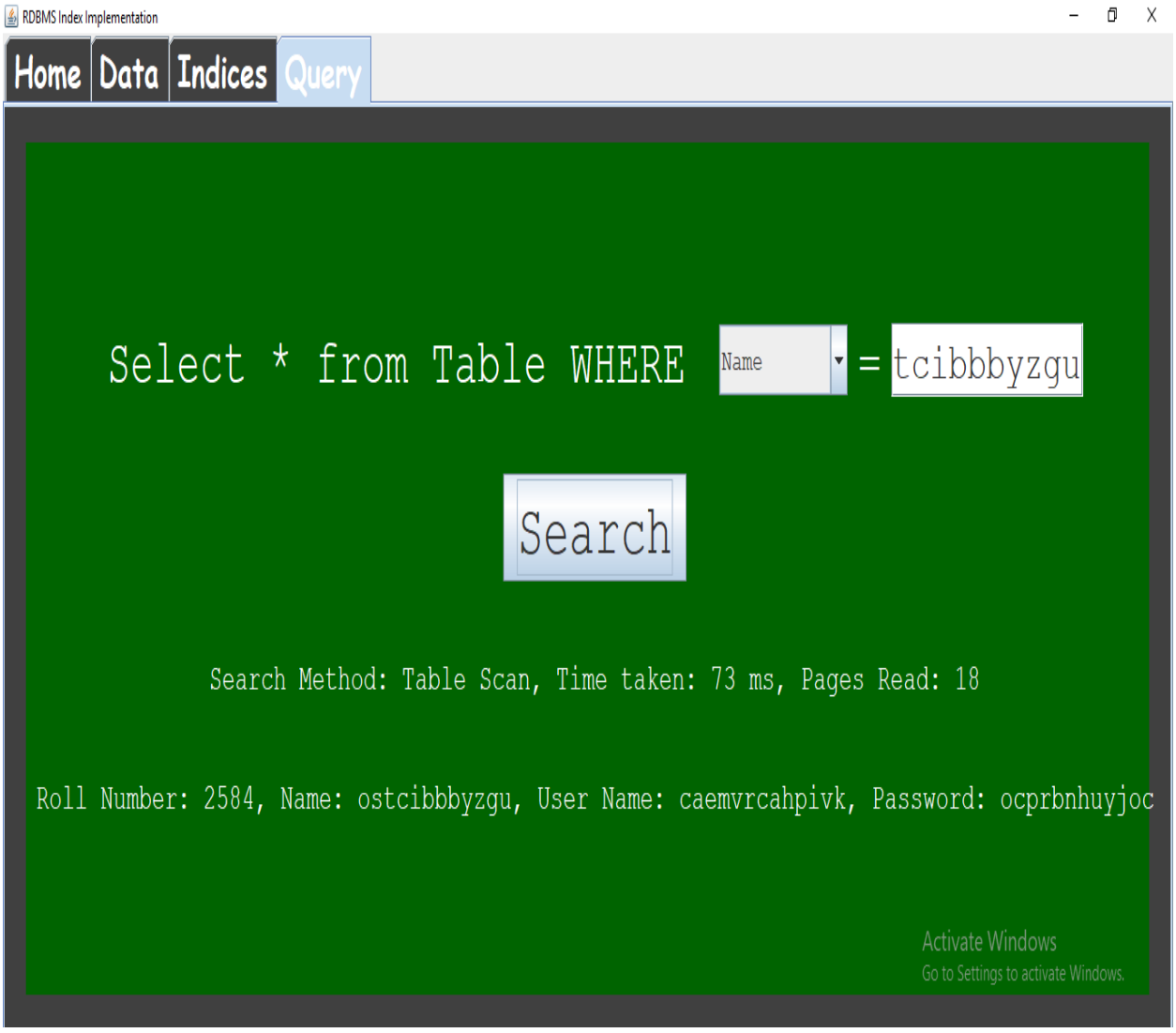

Fig 8.4 Table Scan

Fig 8.5 Index Seek

Chapter 9

SUMMARY

We reduced the time taken to search a given record by using a B/B+ tree rather than indexing and traditional sequential access. It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system.

If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where.Although theoretically, the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case, a reasonable choice for block size would be the size of the processor's cache line.

The space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers

Chapter 10

Conclusions

It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system. Therefore, the objective is to minimize the number of disk accesses and thus, this project is concerned with techniques for achieving that objective i.e. techniques for arranging the data on a disk so that any required piece of data, say some specific record, can be located in a few I/O's as possible

1. From the above observations, it is very clear that B+tree is better than normal indexing in every possible way.

2. Hence it is always desirable to implement B+ tree data structure to search data in an efficient manner.

3. Multilevel Indexing is Better for larger data whereas sparse indexing does well with smaller data

Chapter 11

Future Scope of Work:

1. Bull loading algorithm can be implemented to improve upon insertion in B/B+ tree which is O(log n). The conventional method is to implement using a top-down fashion from the root node to leaf node.

2. Bulk loading can also be implemented using a bottom-up fashion from the leaf node to the root accessing only one level at a time.

3. One could then compare the statistics and decide which way would be better for bulk loading.

4. The great commercial success of database systems is partly due to the development of sophisticated query optimization technology. These techniques can be further applied to all the applications of Dynamic Programming.

Chapter 12

References

1. Database System Concepts taught in class and text reference textbook by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan

2.Bulk loading in https://en.wikipedia.org/wiki/B

3.B+Tree in https://en.wikipedia.org/wiki/B%2B_tree

4.Dynamic Programming: https://en.wikipedia.org/wiki/Dynamic_programming

5. Donald Kossmann and Konrad Stocker, “Iterative Dynamic Programming: A New Class of Query Optimization Algorithms”.

6. Data Structures and Algorithms in Java, Michael T.Goodrich, RobertoTamassia, Michael H. Goldwasser