Understanding B-Tree Search and its Application to Databases

Duncan Morrissey

Graduate Student

BU MET College of Computer Science

CS566 – Belyaev

Fall 2017

**Table of Contents**

Abstract………………………………………………………………………………………………………………………..…………..3

Introduction……………………………………………………………………………………………………………………………...3

The B-Tree Data Structure………………………………………………………………………………………………..……….3

B-Tree Search………………………………………………………………………………………..………………...……………….5

Program Design………………………………………………………………………………………………………..……………….5

Source Code………………………………………………………………………………………………………………………………5

Results……………………………………………………………………………………………..……………………………………….8

Conclusions………………………………………………………………………………………..……………………………………11

References and Source Code Link....……………………………………………………..…………………………………12

**Abstract**

*Accessing data on disk efficiently is a primary requirement for databases and file systems, and presents a different set of problems than traversing a data structure in memory. B-Trees present a tidy solution to disk access minimization. With a sufficient number of keys per node, the height of a B-Tree will decrease significantly and lower the number of reads required. The intention of this research is to clearly examine and highlight the benefit of differently structured B-Trees with regard to the unique problems that databases and disk access present.*

**Introduction**

Although whether B-Tree is actually short for “Balanced Tree” is a source of debate, the data structure’s importance to computer systems over the last 50 years is universally accepted. The B-Tree was first envisioned by Rudolf Bayer and Ed McCreight in the 1960’s while working to optimize data retrieval [5]. Data structures like binary trees, stacks and lists are traditionally more applicable to accessing or organizing data that is already in main memory. Accessing data that is stored on disk however is a different problem. Disk access is orders of magnitude slower than accessing data already in main memory and thus, requires a different solution [3].

The B-Tree data structure was created to address this problem. A B-Tree is a generalization of binary trees or search trees. Each node of a B-Tree can have a variable number of keys and children, and the keys are stored in non-decreasing order [3]. Each node of the tree has a minimum number of keys allowed, and the children of that node fall into ranges between those keys. In order to access children, you can simply compare what key you are looking for to the keys of the current node, and then iterate down the tree structure [7].

This structure largely addresses the problem of slow disk access. Rather than optimize traversal of the data structure itself as we would in a search structure in memory, the primary goal is to minimize data retrieval time from disk, and that means retrieving the data in as few disk reads as [2]. A disk read occurs when we need to retrieve data from a child node, as that child nodes data will live on a different disk page. Therefore, in order to minimize disk reads, the ideal structure of a B-Tree is one with a very low height and large number of keys in each node, as there will be fewer child node iterations required [6]. However, even though disk read minimization is the primary focus, tree traversal of a B-Tree can still be completed in O(nlogn) time [4]. This height minimization is particularly applicable today to databases and file systems.

These benefits manifest themselves in today’s most popular database systems. A database is inherently a structure with a significant quantity of data and the disk read minimization issue is paramount [2]. When accessing different pages for data, the low height B-Tree structure appropriately minimizes I/O and facilitates transfer of significant quantities of data. Microsoft SQL Server, Oracle MySQL and PostgresSQL all implement variations of B-Trees with the number of keys in each node maximized.

**The B-Tree Data Structure**

A B-Tree is a generalization of a search tree structure with several specific rules [1]. It is first initialized with a minimum integer value (t) and then is bound to the following:

1. The root node can have as few as one element within it, but all other nodes have at least the minimum (t)
2. The max number of keys in a node is twice the minimum
3. The elements of a node are sorted in ascending order
4. The number of children in a non-leaf node is always one greater than the number of keys
5. An element at index i is greater than all elements in subtree number i of the node, and less than all elements in subtree i + 1

For example, if we initialize our B-Tree with a minimum value of 1 and the below “values” array, we could get a B-Tree with the following:

Values: 2, 4, 5, 6, 9, 10, 11, 14, 15, 24

A more applicable generalization of this diagram to databases would be 100+ key trees similar to the below. Rather than represent value in these nodes, the numbers represent the number of keys within them [2]:

101 children

. . . .

. . . .

. . . .

The larger number of keys per node will decrease the height, and the height of the tree will dictate how many disk reads are required when accessing data in our database. Given a minimum (t) ≥ 2, our height will be dictated by [2]:

**B-Tree Search**

Searching for an element within a B-Tree follows the same basic idea as searching for an element within a binary tree, just augmented for the additional number of keys. The search process ends when we’ve found our node in question or reached a leaf node with no key equal to our value. We begin at the root node and check whether the key in question is within the root. If the key is not, we then perform two initial comparisons: whether the search value is less than the smallest key in our node, or greater than the largest key. This can result in several scenarios:

1. The search value is less than the smallest key. We then iterate down the smallest child node value and repeat.
2. The search value is greater than the largest key. We then iterate down the largest child node value and repeat.
3. The value is between our smallest and largest key within the node. We then search through all keys between our smallest and largest values for the node itself or child node needed to iterate down. Since the keys within a node are essentially a sorted list, we can use binary search within nodes to speed up our search time.
4. The root node is null (value is not found)
5. A leaf node is reached and our value isn’t present

Even with nodes with 1000+ keys, search on B-Trees will still run in O(logn) time. If we encounter scenario 1 or 2, our search is bound by the height of the tree, and scenario 3 will run in binary search time. Both of these scenarios are O(logn) [1].

**Program Design**

A console application was designed to highlight the benefits of B-Trees for different data input sizes and different tree structures in light of the set of problems that disk access introduces. The purpose was twofold: First, to demonstrate as node size of a tree increases, the number of comparisons to find an element still performs in O(logn) time. Second, even with millions of elements, the height of a B-Tree will still be small, minimizing disk reads required.

Arrays of size 10 – 10,000,000 were populated with random values between 0 and 10,000,000. For each of these arrays, B-Trees with node size of 2 – 100 were created. The number of comparisons required to find a random value in the B-Tree were averaged over 1,000 repetitions, and the height of each tree was measured.

**Source Code**

Below is the Main class for the java application, created using a Dell XPS Laptop with the Eclipse IDE. Excluded from this printout are modifications to the B-Tree class used, specifically the *getNodeComparisons* and *getNodeDepth* methods created. The full source code can be found at <https://github.com/dmorrissey14/MET_CS_566_Final_Project/tree/master/src/bTreeAnalysis>

package bTreeAnalysis;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

import java.io.IOException;

import java.nio.charset.Charset;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

public class Program {

/\*\*

\* Use our BTree program to iterate through different BTree structures and sizes

\* to compare # of comparisons and tree height for search. Output will be to a pipe

\* delimited file for easy Excel analysis.

\* @throws IOException

\*/

public static void main(String[] args) throws IOException {

Random rand = new Random();

List<String> linesComparison = new ArrayList<String>();

List<String> linesHeight = new ArrayList<String>();

//output paths

Path file = Paths.get("C:/Users/dunca/Google Drive/BU\_MET/MET\_CS\_566\_Final\_Project/output/btree\_analysis\_\_comparison.txt");

Path file2 = Paths.get("C:/Users/dunca/Google Drive/BU\_MET/MET\_CS\_566\_Final\_Project/output/btree\_analysis\_\_height.txt");

//Iterate through different array sizes for our BTree comparisons

for (int i = 10; i <= 10000000; i \*= 10){

StringBuilder lineComp = new StringBuilder(i + "|");

StringBuilder lineHeight = new StringBuilder(i + "|");

//Create and populate our test array with values between 1 and 10,000,000

int testArray[] = new int[i];

for (int j = 0; j < i; j++){

testArray[j] = rand.nextInt(10000000) + 1;

}

//Test out average number of comparisons and height for differently structured BTrees

int treeSize = 2;

while (treeSize <= 100){

//create and populate our BTree

BTree<Integer> tree = new BTree<Integer>(treeSize);

for (int a : testArray){

tree.add(a);//this is the bulk of program run time

}

//Test number of comparisons needed to find random number in tree 1000 times

double comparisonAverage = 0;

for (int x = 0; x < 1000; x++){

int tester = rand.nextInt(i);

comparisonAverage += tree.getNodeComparisons(testArray[tester]);

}

comparisonAverage /= 1000;

//check level of max value of tree, this is the height of our tree

int max = testArray[0];

int height;

for (int val : testArray){

if (val > max){

max = val;

}

}

height = tree.getNodeDepth(max);

//add data gathered to our output

lineComp.append(comparisonAverage + "|");

lineHeight.append(height + "|");

if (treeSize == 2){

treeSize = 10;

}else{

treeSize += 10;

}

}

linesComparison.add(lineComp.toString());

linesHeight.add(lineHeight.toString());

}

//write output data to files

Files.write(file, linesComparison, Charset.forName("UTF-8"));

Files.write(file2, linesHeight, Charset.forName("UTF-8"));

}

}

**Results**

Below we have two tables and corresponding graphs demonstrating the output from our console application. The y-axis is the number of keys within a node, the x-axis is the size of our input array.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Height** |  |  |  |  |  |  |  |
|  | 10 | 100 | 1000 | 10,000 | 100,000 | 1,000,000 | 10,000,000 |
| 2 | 1 | 4 | 6 | 7 | 9 | 11 | 12 |
| 10 | 1 | 2 | 3 | 4 | 5 | 5 | 6 |
| 20 | 1 | 2 | 2 | 3 | 4 | 4 | 5 |
| 30 | 1 | 2 | 2 | 3 | 4 | 4 | 5 |
| 40 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |
| 50 | 1 | 1 | 2 | 3 | 3 | 4 | 4 |
| 60 | 1 | 1 | 2 | 2 | 3 | 4 | 4 |
| 70 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |
| 80 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |
| 90 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |
| 100 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |

As you can see, even as the input array creates a B-Tree with 10,000,000 elements, our height is still only four if the number of keys per node is sufficiently high. This stands in comparison to a B-Tree with two keys per node, as that tree would require a height of 12. Considering disk access, this lower height saves us significant amounts of time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Number comparisons** | |  |  |  |  |  |  |
|  | 10 | 100 | 1000 | 10,000 | 100,000 | 1,000,000 | 10,000,000 |
| 2 | 6.409 | 11.728 | 16.41 | 22.696 | 26.944 | 32.802 | 36.608 |
| 10 | 12.134 | 20.633 | 34.521 | 42.953 | 60.969 | 77.455 | 85.006 |
| 20 | 11.644 | 28.883 | 63.302 | 67.685 | 90.04 | 129.964 | 124.573 |
| 30 | 12.198 | 52.271 | 62.656 | 105.524 | 112.852 | 134.229 | 163.885 |
| 40 | 11.976 | 51.78 | 77.136 | 105.066 | 147.125 | 192.503 | 235.509 |
| 50 | 11.86 | 104.384 | 86.395 | 143.267 | 173.995 | 194.581 | 236.506 |
| 60 | 11.722 | 100.762 | 95.122 | 202.415 | 176.679 | 240.834 | 277.037 |
| 70 | 12.186 | 102.49 | 123.715 | 203.854 | 234.173 | 303.756 | 317.751 |
| 80 | 11.966 | 102.586 | 135.63 | 202.921 | 234.884 | 324.345 | 341.783 |
| 90 | 11.93 | 101.982 | 134.584 | 210.507 | 236.815 | 314.501 | 413.388 |
| 100 | 11.998 | 101.75 | 135.966 | 224.039 | 326.439 | 327.14 | 405.657 |

In Fig. 1, the y-axis shows the number of comparisons, the x-axis is the number of keys per node in the tree, and individual lines are differently input sizes.

**Fig. 1**

This demonstrates a relatively small difference between the number of comparisons between differently structured B-Trees, even as the number of keys is increased. Our B-Tree with the largest number of keys is performing only about 300 more comparisons for our largest input.

Each tree individually is also still searching in logarithmic time, as shown by the isolated number of comparisons of our 100 node B-Tree (Fig. 2):

**Fig 2:**

As our input size progresses from 10 to 10,000,000 , our number of comparisons match a logarithmic best-fit curve with an = 0.9753. Clearly, even our B-Tree with the highest number of keys is still performing search extremely quickly.

Ultimately, looking at the sets of data, the B-Trees with a large number of keys are performing as expected. Given an input size of 10,000,000, a B-Tree with 2 keys per node will only take about 12 comparisons to find a search value, while a B-Tree with 100 keys will take closer to 300-400. However, that B-Tree with 2 keys will take about 8 more disk reads to access all of that data. Considering a disk read takes 10-50ms, while in memory operations are measured in nanoseconds, the disk reads saved by having a lower height B-Tree more than offset the 300 nanoseconds lost to less efficient tree traversal. Finally, all of these trees are searching extremely efficiently in log(n) time.

**Conclusion**

The core ideas validated here are still relevant today, even as research has gone beyond the basic B-Tree structure. A more standard implementation today for databases are B+ Trees, which only store data in their leaf nodes, which facilitates a transfer of larger amounts of data. Database Tables are structured as B-Trees themselves, and often have supplemental indexes on them structured as B-Trees, allowing for optimizing search of fields besides the primary field.

Even as NoSQL and Hadoop backed data stores gain popularity, the backbone B-Tree structure of relational databases remains core to the technology industry. Today cutting edge research is performed on scalable and distributed B-Trees by Microsoft and HP. Although the concept may be 40 years old now, its brilliant initial implementation remains relevant and essential to today’s programs.

**References and Links**

[1] Main, M. (2012). *Data structures & other objects using Java*. 4th ed. Boston, Mass: Pearson Addison-Wesley, pp.556-581.

[2] Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (n.d.). *Introduction to algorithms*. 3rd ed. Cambridge, Massachusetts: The MIT Press, pp.484-499.

[3] Bluerwhite.org. (2017). *B-Trees*. [online] Available at: https://www.bluerwhite.org/btree/ [Accessed 10 Nov. 2017].

[4] GeeksforGeeks. (2017). *B-Tree | Set 1 (Introduction) - GeeksforGeeks*. [online] Available at: http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/ [Accessed 17 Nov. 2017].

[5] Perforce.com. (2017). *A Short History of the BTree | Perforce*. [online] Available at: https://www.perforce.com/blog/short-history-btree [Accessed 10 Dec. 2017].

[6] Web.csulb.edu. (2017). *B+-Tree Indexes*. [online] Available at: http://web.csulb.edu/~amonge/classes/common/db/B+TreeIndexes.html [Accessed 1 Dec. 2017].

[7] Opendatastructures.org. (2017). *14.2 B-Trees*. [online] Available at: http://opendatastructures.org/versions/edition-0.1g/ods-python/14\_2\_B\_Trees.html [Accessed 23 Nov. 2017].

Source Code:

<https://github.com/dmorrissey14/MET_CS_566_Final_Project/tree/master/src/bTreeAnalysis>