Data Structures May 8, 2017

A Comparison of AVL Tree and Hash Table for Containing Inverted File Index

Aviraj Sinha and Patrick Yienger

Contents

Introduction

In this paper we will look at the ability of both an AVL Tree and Hashtable to manage, store and retrieve data from inverted indices. The importance of this task can be seen in any book where a search term is linked to data, most importantly to the location where the search term was found. We will be doing a test of the speed of an AVL tTee implementation versus the speed of a Hashtable implementation for an inverted index file.

Considering how data is stored

In this project, the data for the inverted index file was written out and stored in a text file. In order to do this, the necessary data was placed in the appropriate data structure and the count variable was incremented every time, updating with every word entered into the data structure. The process of incrementing the count variable was tedious, because every occurrence of a word had to be compared to see whether or not the count variable should be increased. However, insertions from the index and into the underlying data structure were different. In writing the data structure to the index, the count variable did not need to be updated. This is because the counts were already updated and stored from parsing the PDF file and reading the data structure out to the inverted index file.

Analyzing the results from reading in from the inverted file index will have different results than analyzing the insertion into the data structure from the parser. By analyzing the data of only reading into the data structure from the inverted index, the effects of calculating the counts of occurrence each time a word is added to the data structure will be eliminated. So, the time it takes to read into the AVL Tree and Hashtable will be much slower when the counts need to be updated. Because of this, and due to the fact that eliminating the need to update the count variable every time is more in line with real world uses of HashTables and AVL Trees, we will be using the data of reading in the inverted index into the data structure.

Why the experiment is important:

Due to manipulation of the storing of indices, such as keeping the count, word pages, and documents occurred, the performance will have even more variance due to these variables. As a node is inserted, the iterations of maintaining every node has its own vector structure to store documents. Logically the hashtable will also have a big O of N to maintain. These deviances require a more empirical approach performance.

Steps taken:

First we will discuss the different implementation of AVL tree and hashtable. Then we will take into account step of gathering data will account for different types of data sizes. Data sizes will be represented in a variety of different formats. The rate at speed performance will then be a gathered and then analyzed.

Data structure overview

Data being stored

The data has already been parsed and therefore does not need to calculate multiple counts for the words. Still, it must add the document objects and numbers. The inverted index links the word to all the docs found along with the number found in each document.

Example: word_being_read-docone|3|doctwo|3

Introduction to AVL tree implementation:

In the AVL tree each inverted index is a node within the tree. This node can hold any data that each inverted index can maintain, in this case a document object with a count.

AVL trees are descended from BST which additionally have the capability of maintaining balance difference max height of two. This reduces the cost of insertion that can occur as a regular BST develops an unbalanced ratio and drifts away into linear data structures.

Searching pseudocode AVL:

For searching however we do not need to insert. So it is faster as the rebalancing is never needed.

Initiate search from root
Using a temp pointer to the index

```
while(node ptr is not at a null leaf)
{
  if(the keyword is greater than the current index)
     go down righ node
  else if(the keyword is less than the current index)
     go down left node
  else if(keyword is at its index)
     return the documents where that word occurs
```

}
Not found comes here

Inserting AVL:

Inserting into the avl is a similar algorithm but instead of returning in the the documents where that word occurs it either updates causing an unbalancing and then balancing effect or instead an increase in count if the that word has been there before. Inserting into a tree of index is therefore slower than just searching a tree of indexes.

Introduction to Hashtable implementation:

The hashtable is a data structure that is similar to the functions of an array. In the hashtable written for the program, each spot on the table contained an item object. The item object held inside of it four different variables: a string to hold the word, a string to hold the PDF name, an int to hold the number of times it was in each PDF, and a vector of item objects used to store items with the same key value as the item. In order to place the item object in a specific index on the hashtable, a hashing function was used. The purpose of the hashing function is generate a key value, (a unique number) for each word that needs to be inserted into the hashtable. The hashing function generates this unique number ,and then stores the item object at the index of the number on the hashtable. In this way, each item object has a unique location on the hashtable, and that location can be found by running the word being searching for through the hashing function. For the case of any collisions (item objects that generate the same number when run through the hashing function), the colliding item object was added to the vector of the already existing item object. This way, the hashtable does not write over previously stored data.

When reading in form the inverted index, the hashtable converts the data into an item object. It then takes the word, and puts it through the hashing function, and stores the item at this index on the hashtable. The bigO of this insertion operation is O(1) or constant time. If the hashtable goes to insert, and there is already another item present, the hashtable just adds the colliding item to the end of the vector. This insertion operation is also O(1).

When searching for a word, the hashtable just inserts the word into the hash function. This generates the index of where the word would be stored on the hashtable if present. Then it just searches all of the items at that index for that word. The bigO of this operation is constant time or O(1).

Hypothesis

Statement:

If the data sized is increased, then the time taken for Hashtable will increase more slowly than for the AVL tree. This because hash table has a better performance due to its use of keys, hash functions and buckets.

Note:

Notice that we are comparing specifically storing inverted indices. In the real world, inverted index usage will not result in clear cut results. It will not be as straight forward because of the nature of inverted indices the space for the hash table needs to be allocated and the overall impact of chaining as a worst case needs to inspected. Corresponding to the O(n) growth in the hash structure, the AVL tree will most likely follow somewhat of a O(nlogn) rather than a O(logn).

Many considerations can quickly become complex but for the purpose of a straight-forward experimental analysis, we choose our null hypothesis is that the AVL tree has equal performance to hash table, while our alternative hypothesis is that the hash table has better performance overall.

Gathering Data

Corpus data set collection strategy

The corpora chosen is based on size(mb), pages, and words.

The corpus selections is more varied for the reason of having more than just words indexed as a variable to speed. As seen in *Table 1* Many factors other than words indexed can impact speed of words indexed. As seen from set, which has less words indexed but since it has more documents it must store a much larger amount of data in the inverted indices. Thus, two factors can be derived as affecting the performance: both the amount of words indexed but even more importantly the document names. The documents result that there is still the confounding variable of having a large number of documents increasing the number document objects being stored and written out.

Inserting from a persistent stored index is a common way of making the data search faster. This action removes the overhead and variables of having to count multiple

amounts of the word and parsing from the pdf. Therefore, will be reading in from a data set directly from an inverted index.

Table 1

Data Set comparison									
documents	A	B	C	D	E	F	G	H	I
Size (mb)	1.1	4.8	16.1	61.6	71.1	86.7	92.2	378.6	403.5
words indexed	3354	3639	10587	24151	181,632	95546	39235	85619	155556
pages	93	63	198	1015	5545	4424	1457	3626	10094

Measuring Performance as Time

Table 2

Attributes collected:

- Deciseconds per type of corpus are collected to make it easier to read.
- Words Indexed per second are shown for easier comparison of corpus size impact on structure functioning.(see Table 2 and Table 3)

Corpus set	AVL time	AVL time	percent difference in times
A	0.221	1.2	-4.429864253
B	0.281	1.234	-3.391459075
C	1.046	1.668	-0.594646272
D	2.412	2.338	0.030679934
F	14.74	7.332	0.502578019
G	6.318	3.679	0.417695473
H	1.249	3.225	-1.582065653
I	19.371	6.92	0.642764958
F	18.77	8.81	0.53063399

Time data collection strategy:

First the average of loading from the same inverted index file ten times was done five times. Then the median of 7 of the averages was chosen as the time taken for that combination of structure and data set. This strategy made the collection of data without running out memory and also allowed a very accurate two step

statistic-finding solutions. The Linux time command and user time were used to time the combination of corpus and data structure used.

Table 3

Index insertion times in deciseconds				AVL w/sec	Hash w/sec
Documents used	Words indexed	AVL time	Hash time	15176.47	2795
A	3354	0.221	1.2	12950.18	2948.947
B	3639	0.281	1.234	10121.41	6347.122
C	10587	1.046	1.668	10012.85	10329.77
D	24151	2.412	2.338	12322.39	24772.5
F	181632	14.74	7.332	15122.82	25970.64
G	95546	6.318	3.679	31413.13	12165.89
H	39235	1.249	3.225	8030.355	22479.19
I	155556	19.371	6.92	4561.481	9718.388
F	85619	18.77	8.81		

Table 4

Analysis

Correlation Graphing

Graphing the correlation between data amount and time for both structures to their theoretical upperbounds.

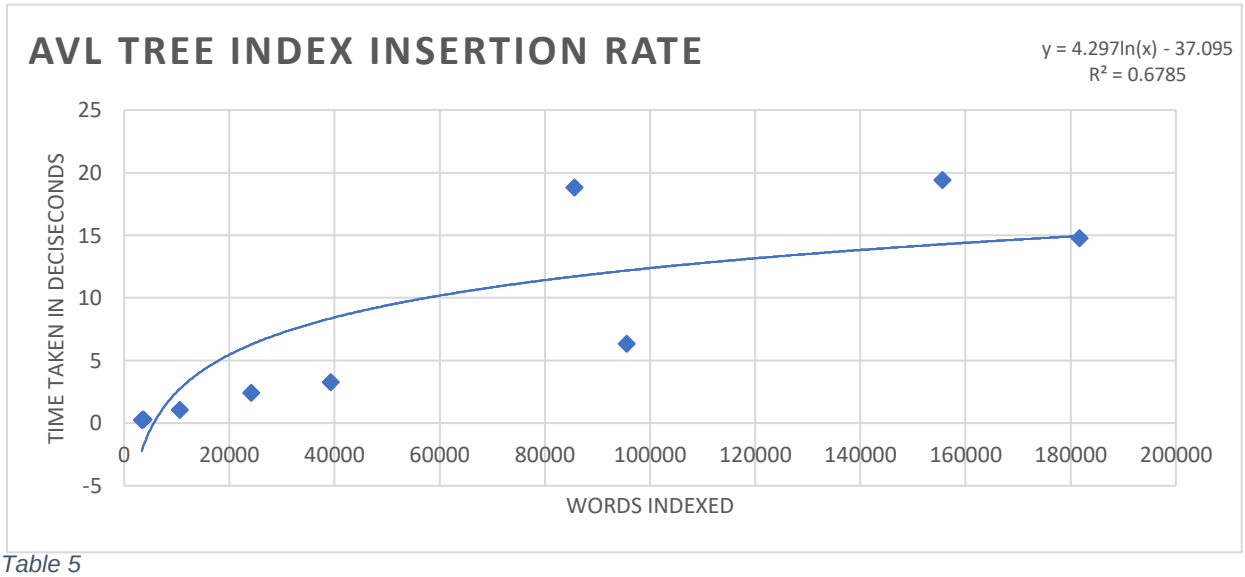

Table 5

From the chart 5 above a log-linear graph may be most appropriate. This is due to the previous assumption of trees that a linear insertion correlation is unreasonable enough

not to be tested. From the correlation of determination we can see that 67% of the points can be explained by logarithmic variation of data.

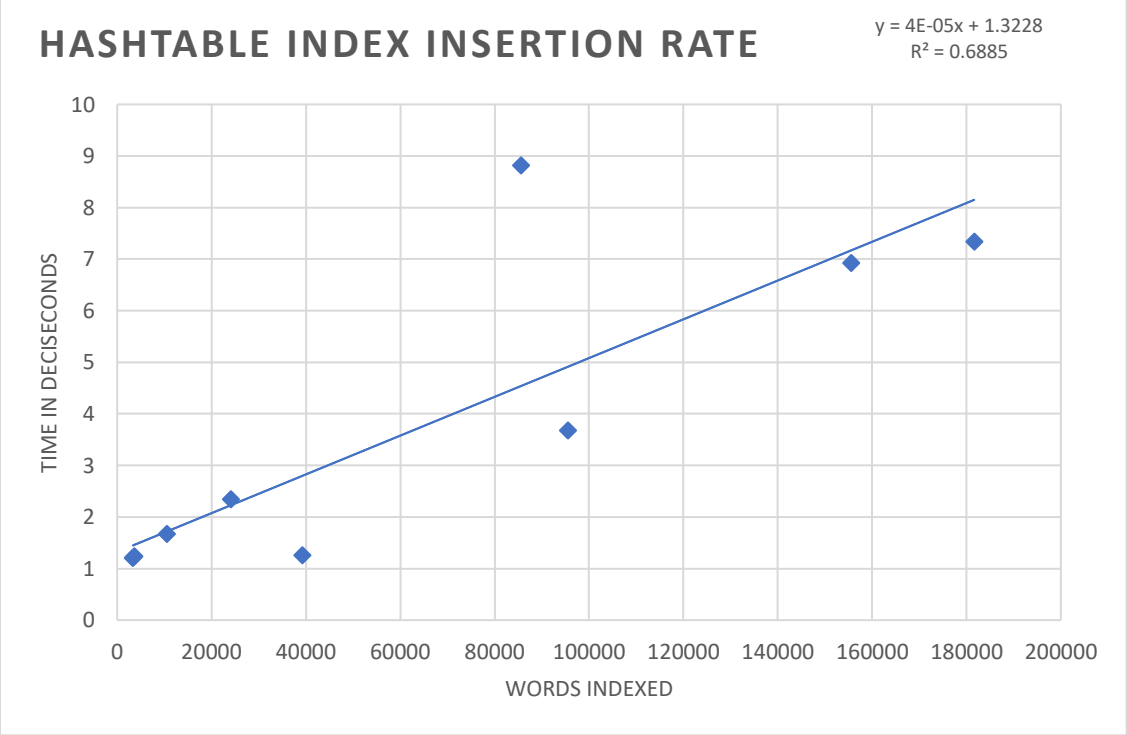

Table 6

The *chart 6* above shows a linear graph may be most appropriate. From the constant time functionality of hash tables, a linear insertion correlation is reasonable enough due to reading in. From the correlation of determination we can see that 69% of the points can be explained by logarithmic variation of data.

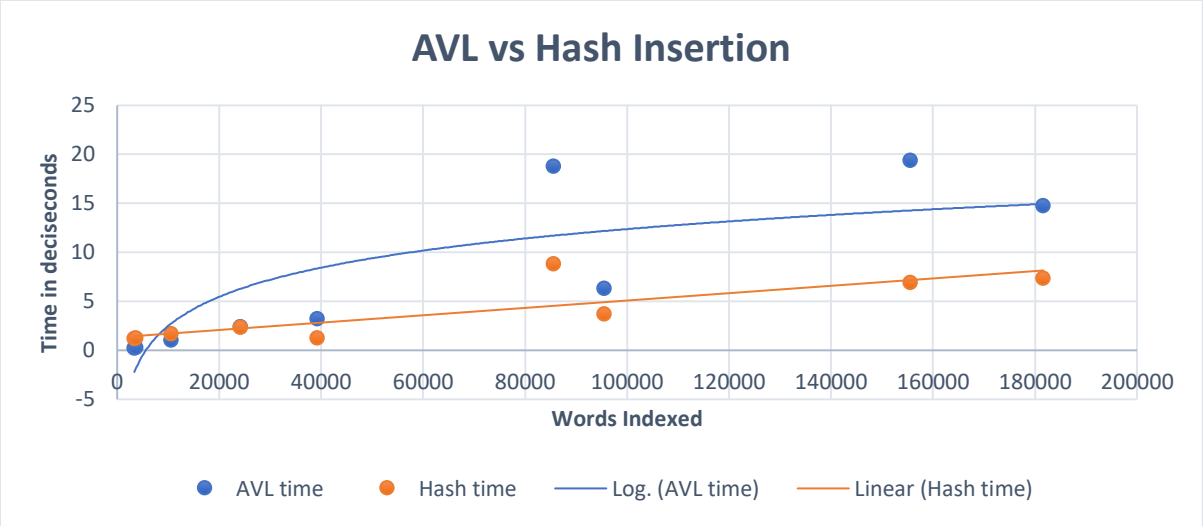

Table 7

As one can see from even from this small data set the trend is apporimately more logarithmic and hashtable is more linear; note having a polynomial, constant or exponential trendline is confounded. From a minimal, visual glance of *Table 7* we can see that the hashtable is consistently faster than the AVL tree at every point past the intersection. This logically means hashtable speed is a an upperbound to the avl tree's rate speed. It can also be seen that for very small corpora the AVL tree is faster.

Matched Paired T test

Reason for test

We did a matches pair t test of mean times of corpus of each. The reason for using this test is that the populations are correlated. Usually used in before and after studies in this case the two data structures' speed is not independent because the data size is impacting it. Therefore, many other test that require independent samples cannot be used. We know it is close to normal and sample size is large so t test is used.

Checking conditions:

- The sampling method for each sample is simple random sampling.
- The test is conducted on paired data. (tests are not independent)
- The sampling distribution is approximately normal, because of central limit theorem.

Running test

AVL (ds)	Hashtable(ds)	Mean difference = AVL-Hash
0.221	1.2	-0.979
0.281	1.234	-0.953
1.046	1.668	-0.622
2.412	2.338	0.0739999999999998
1.249	3.225	-1.976
14.74	7.332	7.408
6.318	3.679	2.639
19.371	6.92	12.451
18.77	8.81	9.96

Table 8

Setting our rejection rate as p of alpha as .1

H0: that both methods produce equal times: mean difference = zero

HA: that avl tree takes more time: mean difference>0

T 1.722

p-val 0.0617

P val is less than alpha we are able to reject the null hypothesis that the hashtable is equally as fast as the AVL tree

Conclusion

From the above empirical tests of correlation and significant difference of paired t statistics, it can be concluded that *overall* the hash table is the more powerful tool in most scenarios.

From the graphing of the two compared data sets, only for small inverted index sizes (approximately 10,000 and less) the AVL tree is faster because of the fact that the hash table is spending time allocating memory it does not need. Results for the matched paired t-tests reveal that that there is a statistically significant performance gap between the hashtable and the AVL tree.

More possible research on searching or deleting can be done but these to operations are not done on a large scale compared to containing the index itself.