Performance Analysis and Optimization on *Lucene*[*]

David Chi-Chuan Su
609 Escondido Road
Stanford, CA 94305
+1 650 497 7416
dsu800@stanford.edu

ABSTRACT

This paper discusses the efficiency of the popular Java-Language-based search engine *Lucene* in terms of its space usage and executing speed. Emphasis of this analysis is based on Lucene's index writing package of the search engine that which includes token scanning, token parsing, document inverting, frequency count, postings sorting, files merging and disk writing. We pinpoint out that Lucene performs not-as-optimum in certain special cases and we show some simple modification can obviate these problems. Alternatively, we attempt to outperform Lucene with the presentation of an implementation that which is based on the standard tries data structure, a commonly used pattern matching data structure in information retrieval. However, the runtime experimental measurement fails to meet what Lucene has achieved. Nevertheless, we discuss and compare the algorithmic complexity of the two approaches and examine some issues related to the proposed implementation and what we might have done instead. We also propose other practical extension that which maybe the valid extension of this project for the future.

General Terms

Languages, Algorithms, Design, Performance, Experimentation

Keywords

Index writing, text analyzing, tokenizing, lexical scan, backtracking, inverted document, standard tries, compressed tries, sorting

INTRODUCTION

Jakarta Lucene search engine version 1.2 is a Java-Language-based search engine that combines efficient batch indexing, fast storing, and powerful searching capabilities. Not only Lucene is highly optimized, it is also very highly modularized; Lucene's functionalities can be grouped primarily into two categories: indexing and searching. Indexing functionality is supported by the encapsulation of original document, tokenizing through buffered stream, repeated parsing (stop words and or Porter stemming) passing the analyzing stage, inverting the documents, recording the frequency counts, sorting the postings, encoding the term positions, merging the sub-results if necessary, and storing them onto the disk. The summary of the entire Lucene indexing process is illustrated in Fig. 1. Searching functionality, on the other hand, involves the parsing of query, performing some versions of optimized search, reading the relevant data from disk and returning the results.

More specifically for the indexing part, Lucene document directory contains the class package that does the encapsulation of data field. *Analysis* package, under Lucene analysis directory, includes the set of *Tokenizer* and *Analyzer* classes that are used to tokenize and parse the raw text data. And then we have Lucene index package contains classes that records and provides the basic functionalities of inverted indexing, i.e., counting, recording and sorting mechanism of terms of document. To avoid runtime memory overflow, designers of Lucene also included classes in the Lucene index package that will carefully divide document, especially long documents, into several segments, sort them individually, and then merge the sorted segments at the end. Finally, we have the Lucene store package that takes care of the final writing of δ-encoded postings onto the disk.

The main purpose of this project is to focus primarily on Lucene's indexing functionality, of which not only deals with Lucene's ability to efficiently processing documents and storing them onto disk but also consumes the bulk of the available resources. We will first begin by examine closely the underlying data structure employed by

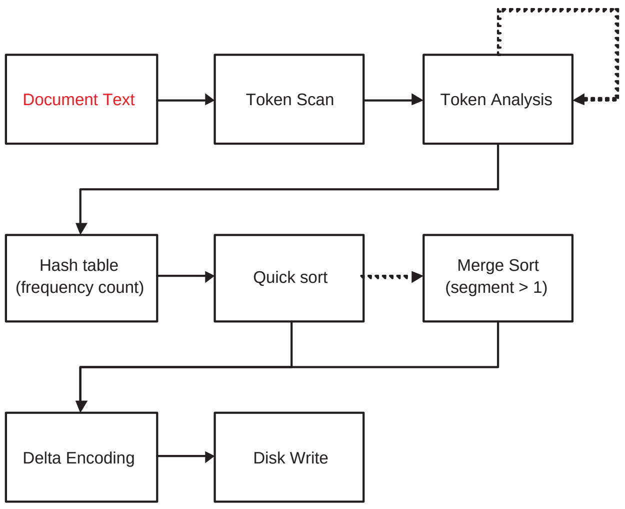


Figure 1: Lucene indexing

Lucene and make some judgment about its implementation and then we will attempt to improve Lucene by changing its data structure and its algorithms.

ANALYSIS

For each individual document, *DocumentWriter* class will create the posting table by hashing the term, storing occurrences using extendable array, and at the end perform a quick sort to alphabetize postings before writing them back to disk. Some algorithmic analysis is done on the Lucene *IndexWriter* and can be summarized by the following indexing steps:

1. Create tokenized stream from fields of document. The result is an enumeration of terms with stop words filtered and possibly normalized and or Porter stemmed.
2. Invert the documents by counting occurrences through the used of hash table. Positions are recorded in posting table using extendable array.
3. Sort the posting table using an in-placed quick sort.
4. Write sorted postings to disk using δ encoding.
5. Merge sort the segments representing the entire document.

Let's assume that n terms appearing in the m documents. The cost is linear time $O(mn)$ for reading the corpus and the for the iterations of tokenizing and filtering of all documents in step 1). Since it takes constant time to add, to find and or to update the weight of a specific token in the hashtable (ignoring collisions), the total time to construct the hashtable would be $O(mn)$, with the storage overheads for hashtable $\approx 1.5n$ (for each document) and by then the table must fit in main memory. Sorting the unique terms using quicksort in step 3) incurs statistically of complexity $O(mnlog(n))$ and since it is done "in-place," the space overheads are n as there are no additional overheads associated with the sort. Step 5) is just there because the original designer had the concern for machines that don't have sufficient runtime memory for text processing. Combining with step 4), since these two steps involve heavy disk reads and writes, it will be disproportional with our running time calculation and so we chose not to include that in our discussion [1]. Consequently for our runtime analysis, we simply reset the maxFieldLength field to Integer.MAX_VALUE from the original 10000 to avoid merging and also disable the disk writing to prevent disk IO operations from ever occuring.

Lucene is quite efficient in processing documents of small sizes as hashing and quick sort incur very little overhead in computation. However, in case where the text field is large, step 1) will be quite inefficient because of the cost involving with the number of iterations through the input stream (depending on the degree of filtering), which must be done to yield valid tokens. The Porter stemmer algorithm, in particular, is time consuming, as we need not only to record the input vowel-consonant combinations by one forward scanning but also need to scan backward to match suffix for its six stemming steps [2].

In addition, the drawback appears in the way that repeated occurrences are stored in the postings. When having large duplicate terms appearing in the document, the extendable array data structure requires heavy deep copying whereas a list data structure would simply require one append operation.

Very unlikely, however, it may be the case where document terms are already appearing in some lexical order (as for document containing indexes of articles for instance) and thus quick sort with the middle index does not always appear to be the most optimum case in some situation [3].

PROPOSAL

Two proposals came up during the course of this project. The first one involves simply correcting the above-mentioned weak points of Lucene. For the second model, we come up with the idea of making token parsing, frequency counting, and sorting an one-pass process, i.e., instead of requiring iteration of token scanning, hashing and sorting, we claim that postings table can be built and sorted as the terms are filtered, and stemmed. The idea is to add finite state machines to

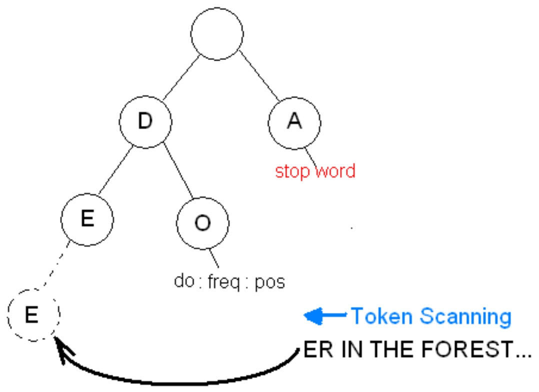


Figure 2: Standard tries

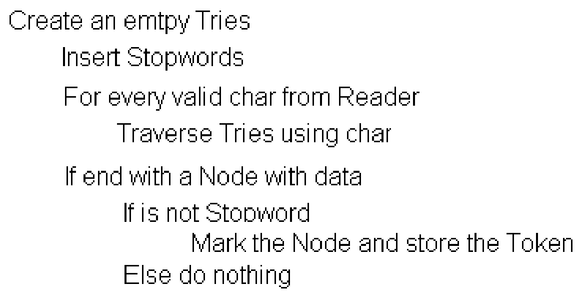
Tokenizer to guide the filtering process and to use tries data structure as the backbone storage and sorter as to replace the existing quick sort and hashtable combination. More specifically, a standard tries as in Fig. 2 will be used and so that each leaf will correspond to an unique term.

IMPLEMENTATION

For the first model, we exchange the extendable array data structure used in the position storing to a linked list from java.util package [4]. Also, the existing quick sort algorithm is modified into a randomized version so that a more uniform performance can be expected. For efficiency, we store strings of stop words in a hashtable to recognize them in constant time. Also, instead of using complex *Term* class objects as the key for the lookups in hashtable, we change the key to simple *String* objects to increase the speed and to rid of unnecessary memory usage.

To keep the second implementation manageable we choose to adopt the simplest approach that is to ignore all numbers and punctuation and use only case-insensitive unbroken strings of alphabetic characters as tokens and without any stemming. In detail, a *TrieNode* class is used a basic data holder (equivalent to a Lucene *Posting* class) in a *Trie* data structure, which provides the construction, traversal, and printing capabilities. We made the transition from Lucene smooth by using the *Scanner* class as a bridging class between the original Lucene and the new data structure. In addition, the normalizing of letters is now done within the *Trie* structure during descending; also, we add the *TokenCheck* class to replace the existing *Analyser*. Further, we built a *Tester* class to make ensure all class function correctly.

The executing process begins by the interception of



```
Create an emtpy Tries
    Insert Stopwords
    For every valid char from Reader
        Traverse Tries using char
    If end with a Node with data
        If is not Stopword
            Mark the Node and store the Token
        Else do nothing
```

Figure 3: Psudocode for tries construction

Reader before the tokenizing step by the *Scanner* class. We proceed by first building the *Trie* data structure with all of the stop words nodes inserted into the tree. Then we start scanning the input stream one character at a time till we reach a non-valid character, descending down the *Trie* from the root. Each positioned node will check if there is any occupying data string, in case of the occupying is a stop word, we ignore the node, else we update the frequency count of that node and record the position. If the data string is empty then we simply set the string to the current token. The whole process can be described by the pseudocode at Fig. 3.

BENCEMARK

Sample corpus consists of the set of 6,400 TREC-like format files converted from the condensed version. The three versions of Lucene implementations are run against each other using the sample set of size 50, 100, 200, 500, and 1000 documents. All experiments are under Windows environment and the results are shown in Fig. 4. Moreover, the three implementations are run against each other on a set of 9 longer digests, with the doubling of size of documents at each subsequent test. The results are shown in Fig. 5.

From the Fig. 4, we see that the modified version matches closely with the original version with the breakpoint at around the set of size 200 documents where before that size the modified version outruns the original version and vice versa after size of 200 documents. The tries implementation, on the other hand, takes average three times as long to complete than the original version in all test cases. In Fig. 5, we see a slow down of modified version versus the original Lucene implementation. However, we see that there is an huge improvement of the tries versus the original Lucene implementation as the time cost ratio of the original implementation to tries is down to less than one half, as opposed to one third for the shorter documents.

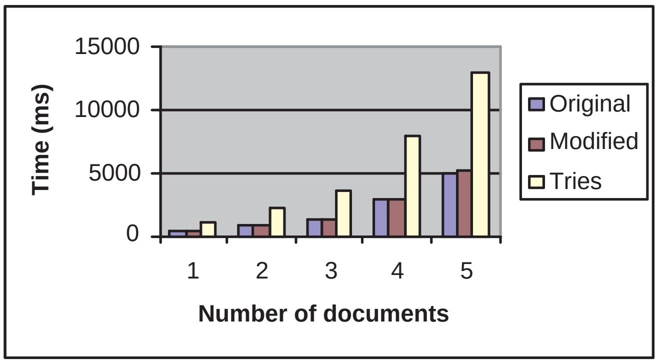


Figure 4: Runtime measures

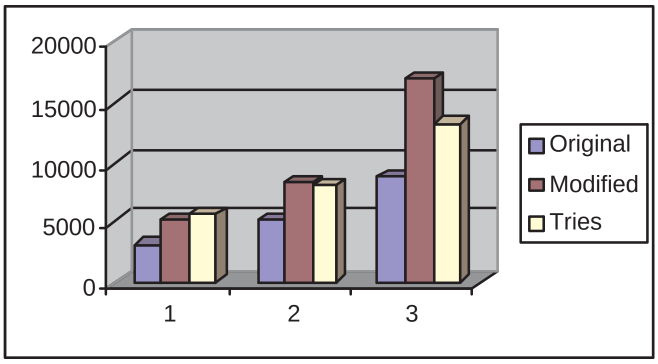


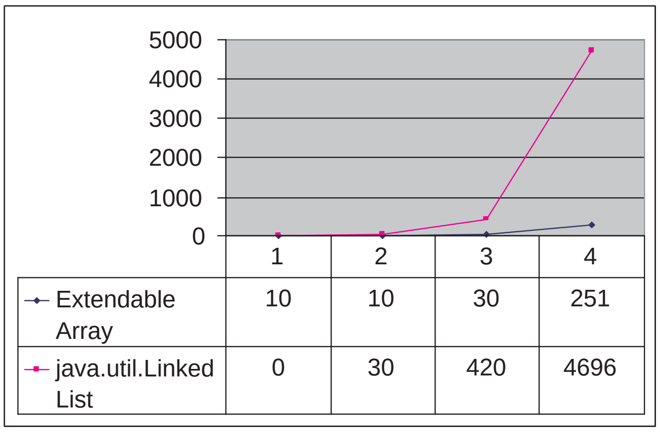
Figure 5: Runtime for longer documents

POST ANALYSIS

One deviation from norm is the outrun of Lucene's original version to the modified version, as much as twice as fast, especially when the documents get long. One conception is that list with constant time edge insertion and deletion will always outperform the extendable array with amortized runtime of $O(n)$. To demonstrate the speed of the two strucutes, we perform runtime analysis on just the linked list from java.util package and extendable integer array. The two data structures are tested on the sets of 1000, 10000, 1000000, and 1000000 integers and the results are shown in Fig. 6. From the graph we see that list outperforms extendable array when the number of insertion is low. This is well expected, as the extendable array requires several deep copy of entire array to a larger array. However, extendable array outruns list when the number of insertion gets huge. The explanation lies on the overhead build-ups of using *Integer* wrapper class for every integer operation in Java. This is, evidently, not only time consuming but is also very memory inefficient.

As for the tries implementation, we will assume that there are n terms of approximately d characters each appearing in the m documents. The cost is linear time $O(mnd)$ for the combined operations of tries construction as the reading the corpus, the normalizing of tokenizing, and the descending of each token down the tries incur the total cost linear to mnd, i.e., the total number of valid characters in all d documents. The space, on the otherhand, is fixed at $O(nd)$, which yields the estimate executing time $O(mnd)$ required by the traversal and the printing of the tries.

The explanation behind the loss is that the tries data structure requires a great deal of resource to construct in runtime. By our deployment of "lazy construction" for tries, the data node will only be created as needed during in runtime. Such process will be particularly expensive for short documents as each inverting of document amounts to building a new tries data structure from the ground. Another reason for the loss is with the use of tries as a stop word filter versus the use of hashtable. Suppose there are k stop words appearing in a document set each with average of v characters length. Whereas it requires time proportional to total characters of all stop words appearing in the document ($O(kv)$) for tries, it takes just $O(k)$ for hashtable by the original version, with the neglect of scanning time for both implementations. Evidently as shown by Fig. 5, by sharing the cost of construction of tries structure, the tries implementation is capable catching up the speed of the normal Lucene implementation.



	1	2	3	4
Extendable Array	10	10	30	251
java.util.Linked List	0	30	420	4696

Figure 6: Extendable array vs. List

POTENTIAL ISSUES

One nice feature for tries implementation to add is the construction of one-time pass (single, non-backtracking iteration of scanning through the text stream) Porter stemmer that which would enable us to stem the token as we descend down the tries. Such proposal turns out to be a cumbersome task as eventually we will be building a recursive descend scanner with huge amount of transitional states. The grammar is obviously not LL(1), which means we will either has to turn to an LR(1) parsing, which is almost manually infeasible, or we have to allow extra lookaheads as in LL(n), which defeats the whole purpose of one-time passing with no backtracking [5].

The original proposal comes with the idea of chaining the bottom leaf nodes of tries during construction so we are obviated of the final traversal of the inner nodes. Such idea is obviously plausible, however, by adding such feature turns out only saves the amount of total memory on the stack required by recursively traversal, not the total running time as there are just about the same number of nodes that we need to traverse to find the previous and or the next node in the chain as in the usual traversal.

The amount of space usage for tries implementation is $O(nd)$, which is reducible to $O(n)$ if a compressed tries, instead of standard tries, is used. However, the construction of compressed tries is usually is done in the style from bottom-up, i.e., we first construct a standard tries and then only after such construction we start merging the nodes bottom-up with no internal node with less than two edges to form the compact version that require nodes one linear (exactly double) to the number of linear nodes [3]. The construction is seemly possible, however, such scheme of top-down construction of compressed tries would require a proof to show that it is doable. Of course, when a scheme of compressed tries is deployed, implementation involving chaining as described previous will no longer be necessary.

FUTURE WORK

While we have not achieved the benchmark set by the original Lucene implementation, we have demonstrated other approach that which might be useful in some special cases. In particular, the modified version is more suitable for smaller set of short documents and tries implementation is good for long document sets. However, more experimentation is needed to evaluate the performance and space consumption.

Also, besides the incorporation the Porter stemmer into tries, chaining of bottom nodes to avoid traversal overhead, and the deployment of top-down compressed tries, we can also combine the computing of IDF's and vector length in the one iteration before the final write to disk to avoid iterating though all tokens twice (after all documents are already indexed).

Finally, pre-allotting the memory to a statistical depth of d (making the structure static) for tries and use the same tries structure for all documents would be a nice idea as we can then rid of the execution time required by consecutive runtime memory allocation. In this case, we only need to be able to identify the data string of the current document from the previous ones.

CONCLUSION

Implementation of Lucene using the data structure tries is a novel alternative toward the conventional IR indexing, which usually involves scanning, parsing, counting, and sorting. The idea involves of normalizing, parsing, recording, and sorting as we scan through the buffered stream character by character. Some similar approaches are taken from the idea of parsing in the art of compiling. The runtime results do not meet the benchmark set by Lucene's implementation; however, we show that tries implementation is approaching Lucene's implementation as the documents lengthens. Suggested as an extension, with some modification to the standard tries we can yield compressed tries, which can dramatic improve its performance in terms of its executing time and space requirement.

REFERENCES

[1] Ian H. Witten, Alistair Moffat, Timothy C. Bell. $ManagingGigabytes : Compressing$

$andIndexingDocumentsandImages$. Second edition, Morgan Kaufmann Publishers, Inc, 1999.

[2] Martin Porter. The Porter Stemming Algorithm. http://www.tartarus.org/ martin/PorterStemmer.

[3] Michael Goodrich, Roberto Tammassia. $DataStructuresandAlgorithmsinJava$. Second edition, John Wiley, and Son, Inc, 2001.

[4] Ivor Horton. $BeginningJava2 : JDK1.3Edition$. Wrox Press Ltd, 2000.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. $Compilers : Principles, Techniques, andTools$. Addison-Wesley, 1986.

[6] Ricardo Baeza-Yates, Berthier Ribeiro-Neto. $ModernInformationRetrieval$. Addison-Wesley, 1999.

[7] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. ICFP, October 4, 2002.