Document Indexing Using Aho-Corasick State Machines

Damien DuBois, Justin Kamerman, Ramanpreet Singh

October 26, 2011

Abstract

The Aho-Corasick algorithm was originally proposed as a bibliographic search mechanism, efficient enough to preclude the construction and maintenance of a search index. As the size of document collections have grown since the development of the algorithm, document collections have grown to the point where cost of rescanning a corpus for every search has become prohibitive and indexing, in some form or another, is an essential optimization for any modern information retrieval system. In this paper we demonstrate the use of Aho-Corasick, not as an alternative to indexing but as a lexical scanning tool employed in the construction of an inverted index. Consistent with our analysis, an empirical evaluation of our indexer implementation shows significant improvement over a naïve algorithm. The modular architecture of our indexer allows the indexing task to be parallelized over multiple physical nodes. The results of parallelization are presented and offer interesting insights towards further horizontal scaling.

Keywords: inverted index, Aho-Corasick state machine, finite automata, pattern matching, parsing

1 Introduction

The basis of most document retrieval systems is the term-document incidence matrix. This matrix is typically sparse and more efficiently represented as an inverted-index which maps terms to the parts of a document where they occur.

In order to construct an inverted index, documents must be scanned to determine term frequencies. The Aho-Corasick string matching algorithm[1] is a simple and efficient text scanning algorithm. The algorithm constructs a finite state machine to scan for a given set of keywords. It is, in effect, a reduced grammar regular expression parser of the type described in [2]. The algorithm is simple and efficient, construction time being proportional to the sum of the length of the keyword set, and the number of state transitions required to scan a document is independent of the number of the size of the keyword set. In this paper we demonstrate the use of Aho-Corasick, not as an alternative to indexing

but as a lexical scanning tool employed in the construction of an inverted index. We provide analytic predictions of our implementation against a naïve algorithm and finally, an empirical evaluation of our inverted indexer implementation is conducted and the results analyzed.

2 Implementation

The Aho-Corasick indexer was implemented using Java and MySQL relational database. Upon startup, the indexer reads a list of index keywords and synonyms from the database and constructs an Aho-Corasick state machine. The indexer then enters a loop in which it uses the state machine to scan batches of documents retrieved from the database. The document batch size is varied manually to match the physical capabilities of the host. A possible enhancement to our implementation could include an adaptive loading component. The output of the state machine is used to augment a global index maintained in the database.

Figure 1 shows how multiple indexer instances can be deployed in parallel, using the database to synchronize their access to the corpus and index. The individual indexers do not interact directly with one another, making for a simple deployment and operation model.

Section describes the tests conducted to examine the effect of dictionary, document, and corpus size on the performance of a single Aho-Corasick indexer and in we compare our indexer performance to that of a naïve implementation. Construction time of the state machine as a function of dictionary size was tested in order to quantify the overhead it imposes on the indexing task. In section 3.5 we explore the performance characteristics of a parallel indexer deployment.

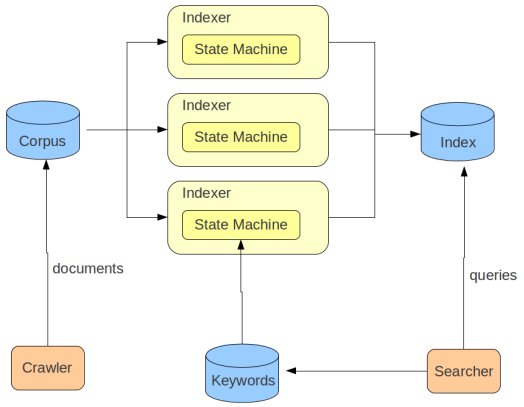

Figure 1: *Indexer deployment model showing multiple indexer instances operating in parallel*

All tests were run on a single Intel Core 2 Duo 2GHz processor, 1GB RAM, running a 64-bit Linux 2.6.35 SMP kernel. The Java Virtual Machine used was

version 1.6.0-24.

3 Experiment Results

3.1 Number of Keywords

To study the impact of dictionary size on performance keywords, a 10,000 keyword dictionary of the most popular English words [3] and a corpus of 50,000 blog posts of varying size were used. First, the time taken to construct the Aho-Corasick state machine was measured for different numbers of dictionary keywords. The results of this test, shown in figure 2, indicate construction time to be linear with respect to the number of keywords. This is consistent with [1] which formally proves that the state machine construction algorithm is linearly proportional to the sum of the lengths of the keywords used to construct the state machine.

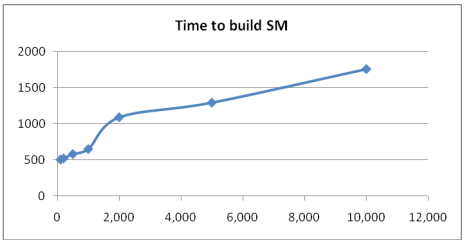

Figure 2: *Time to build Aho-Corasick state machine as a function of the number of keywords. Time taken is shown in milliseconds on the y-axis and the number of keywords on the x-axis*

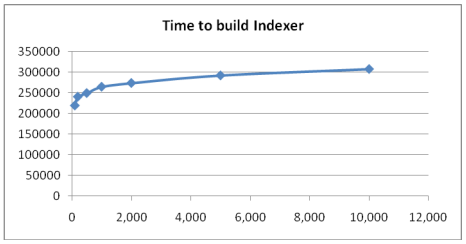

Figure 3: *Time to build the inverted index as a function of the number of keywords. Time taken is shown in milliseconds on the y-axis and the number of keywords on the x-axis*

Second, the time taken for a state machines to build an inverted index of our corpus was measured. The test was repeated using state machines constructed with a varying number of keywords and the results plotted in figure 3. According to [1], which proves that the number of state transitions involved in processing an input string is independent of the number of keywords used to construct the state machine, we would not expect the time complexity to increase with the size of the keyword set. However, as can be seen in 3, the time to build the index increases logarithmically with the number of keywords. This is likely due to the fact that as the number of keywords increases, so does the size of the index and the amount of database interactions required to update the index.

It is important to note that for this experiment, the time taken to construct the state machine is two orders of magnitude lower than the time taken by the indexing task. This is an important result as it validates the practical use of the Aho-Corasick algorithm to index large corpora, the overhead of state machine construction time being negligible relative to the indexing task. Also, the space

used by the index in the database is proportional to the number of words we are using.

3.2 Size of Corpus

To examine the effect of corpus size on the performance of the indexer, a 5,000 keyword dictionary and a corpus of 800,000 blog posts of various sizes were used. The timing results of indexing corpora of varying sizes, shown in figure 4, indicate that the time to build the index with the state machine is linear with respect to the number of posts.

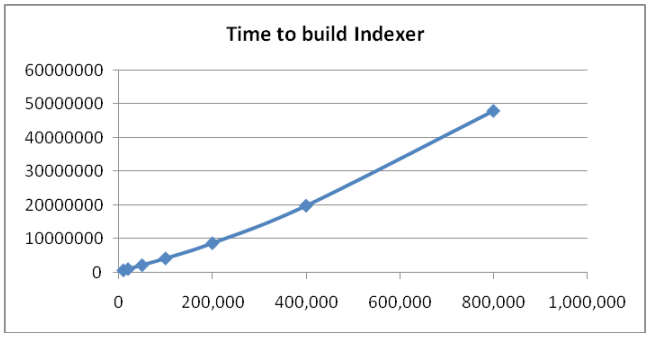

Figure 4: *Time to build the inverted index as a function of the size of the corpus. Time taken is shown in milliseconds on the y-axis and the number of documents in the collection on the x-axis*

3.3 Document Size

To examine the effect
of document size on indexer performance, a 5,000 keyword dictionary and 5,000 blog posts were used. Tests were run in which the the size of the blog posts was varied, generating the performance curve in figure 5. As expected the time to build the indexer is almost linear with respect to the average document size.

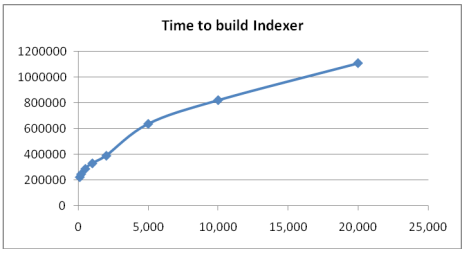

Figure 5: *Time to build an inverted index as a function of the average document size. Time taken is shown in milliseconds on the y-axis and the average document size in characters on the x-axis*

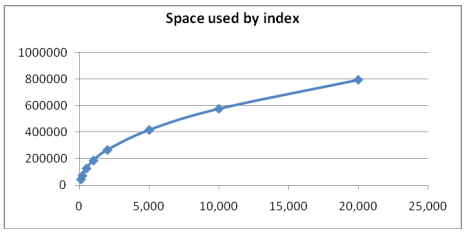

Figure 6: *Size of index as a function of the average document size. The number of database rows used to store the index is shown on the y-axis and the average document size in characters on the x-axis*

Examining the size of the index created, the larger the documents the more

records created. However, the relationship is not linear since a single record is stored to represent one or many occurrences of a particular keyword in a document. This relationship is shown in figure 6.

3.4 Comparison with Naïve Indexer

In order to quantify the benefits of using the Aho-Corasick algorithm for indexing, our indexer performance was compared to that of a naïve implementation. The naïve indexer builder has the same input as our system i.e. a list of keywords and a list of documents, and the same output i.e. an inverted index corresponding to the keywords and documents. Algorithm 1 is used in principal for both implementations. However, whereas the Aho-Corasick implementation uses a state machine to scan, the naïve indexer tokenizes the document and compares each token to the keyword dictionary.

Algorithm 1 Build inverted index

```
for all keyword do
  Build a vector v
  for all document do
    if keyword ϵ document then
      Add the row (keyword.id, document.id) into v
  end for
  Save v into the database
end for
```

Algorithm 1 has a complexity proportional to the number of keywords n, the number of documents m, and the average document size $\bar{c}$. For our chosen architecture, execution time is also effected by database latency. In the case of the naïve indexer, we perform one database transaction per keyword and in the Aho-Corasick implementation, we perform a database transaction per document. For analysis purposes, we designate the average database latency per transaction as $\bar{t}$.

For the naïve indexer implementation described above, the expected time complexity relationship is shown in equation 1. For the Aho-Corasick indexer, the expected theoretical complexity relationship is given by equation 2.

$$Time \propto n * (m * \bar{c} + \bar{t}) \tag{1}$$

$$Time \propto m * (\bar{c} + \bar{t}) \tag{2}$$

In the context of these equations, we describe and interpret our empirical results in the following subsections, testing for the effects of document size c, number of keywords n, and number of documents m.

3.4.1 Document Size

To examine the effect of document size on indexer performance, the number of keywords was fixed at 1,000, the number of documents at 5,000, and tests run against corpora composed of varying average length. From equations 1 and 2, we would expect both implementations to behave linearly with respect to average document size and to differ by a constant scaling factor n, the size of the keyword dictionary. The experiment results are shown in figure 7 and support this analysis. Also, we observe that for smaller documents, the naïve method performs better. This is attributable to the fact that for smaller documents, the database advantage of the naïve implementation is enough to overcome the keyword dictionary size scaling factor. To see this effect more clearly, experiment results are plotted on a logarithmic scale in figure 8.

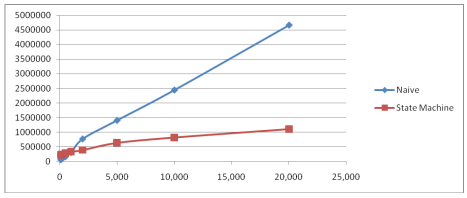

Figure 7: *Time to build an inverted index as a function of the average document size. Time taken is shown in milliseconds on the y-axis and the average document size in characters on the x-axis*

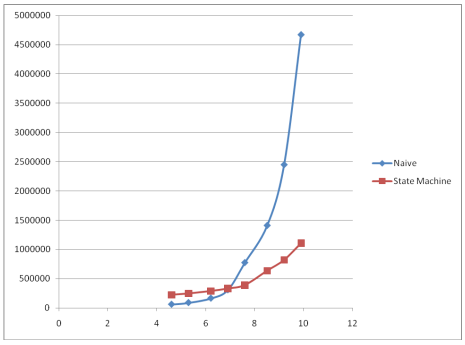

Figure 8: *Time to build an inverted index as a function of the average document size. Time taken is shown in milliseconds on the y-axis and the average document size in characters on the logarithmic x-axis*

3.4.2 Number of Keywords

To examine the effect of keyword dictionary size on indexer performance, the number of documents was fixed at 5,000, and tests run using varying numbers of keywords. From equation 1, we would expect the naïve implementation to have a linear dependency on the number of keywords and this is supported by the test results. From equation 2 we would not expect the Aho-Corasick implementation to depend on the number of keyword however, the test results seem to indicate a linear dependency. This can be attributed to the fact that the larger the keyword dictionary, the more likely we are to find matches in a document and incur the expensive database latency cost.

The experiment results are shown in figure 9 and on a logarithmic scale in figure 10.

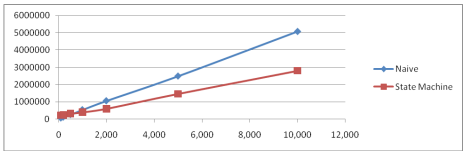

Figure 9: *Time to build an inverted index as a function of the number of keywords. Time taken is shown in milliseconds on the y-axis and the number of keywords on the x-axis*

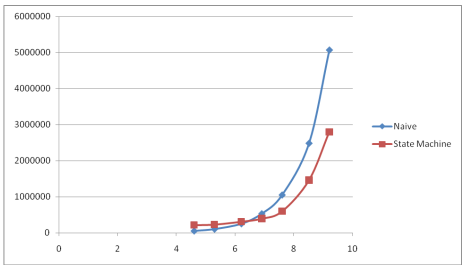

Figure 10: *Time to build an inverted index as a function of the number of keywords. Time taken is shown in milliseconds on the y-axis and the number of keywords on the logarithmic x-axis*

3.4.3 Number of Documents

From equations 1 and 2, the time complexity is linearly related to the number of documents being indexed. The experiment results, in this respect, are shown in figure 11 and on a logarithmic scale in figure 12.

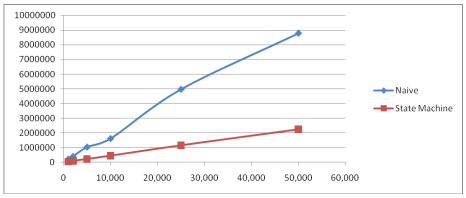

Figure 11: *Time to build an inverted index as a function of the number of documents being indexed. Time taken is shown in milliseconds on the y-axis and the number of documents on the x-axis*

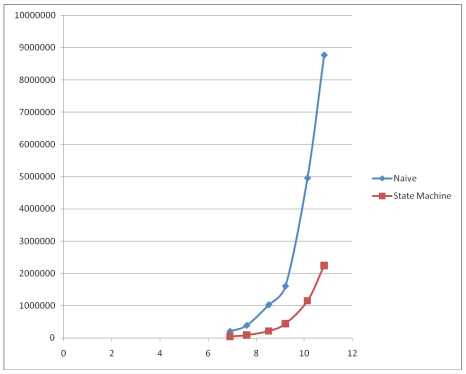

Figure 12: *Time to build an inverted index as a function of the number of documents being indexed. Time taken is shown in milliseconds on the y-axis and the number of documents on the logarithmic x-axis*

3.5 Parallelization

Given the modular architecture of our indexer, it is possible to deploy multiple instances of the program to work concurrently on the same indexing task. The instances interact via the database which synchronizes their operations on the corpus and the index itself. To investigate the effect of parallelization on

the indexer operation, the indexer was deployed using a single database server storing documents, keywords, and the resulting index. Four indexer nodes were used, each running a single instance of the indexer program.

In this experiment a corpus of 100,000 documents and 1000 keywords were used, each document 5000 words in length. Tests were run using various node combinations and the time taken to index the corpus was measured and normalized to a throughput metric (documents/second) for convenient comparison and to account for differences in the physical capabilities and configuration of the worker nodes. A series of tests were performed using a corpus of 800,000 documents to investigate the effect of corpus size on performance. The results of this experiment are shown in figure 13.

For the 100,000 document corpus, increasing the number of nodes increases throughput but with diminishing returns. As the database connection pool is saturated, this becomes the limiting factor of the architecture. If the number of nodes were increased further, one would expect the curve to plateau at some point. After this point, adding additional nodes would offer no improvement and may even decrease performance as the worker processes spend an increasing amount of time context switching and contending for locks. These factors could be mitigated up to a point by increasing batch sizes and tuning the database. However, the characteristics of the curve are ultimately unavoidable in the current architecture.

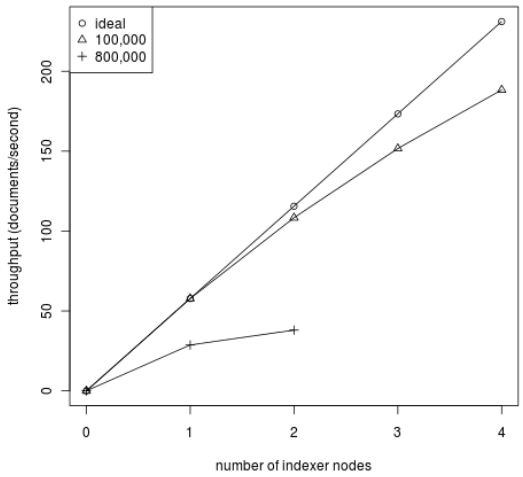

Figure 13: *Performance characteristics of parallel indexer deployment. The ideal curve shows the theoretical performance curve based on linear combination of average individual node performance over the corpus*

For the 800,000 document corpus, the performance curve shows an even more severe degradation than the smaller corpus. Since we normalize the performance metrics for the size of the document collection, one would expect this curve to be more similar to that of the 100,000 document collection. A possible explanation for the deviation is that for the larger collection, more documents were present in the database and the resulting index larger. This results in longer table scan

times to retrieve and update records. This effect could be alleviated through the use of indices and other database optimization techniques, however, as for the smaller document collection, this degradation is ultimately unavoidable.

4 Conclusions

The experimental results tend to prove what we could expect from theory. Our system is faster than the naïve indexer with respect to the number of keywords being indexed, size of the document collection, and average document size. In particular, our indexer implementation performs relatively well with respect to the number of documents being indexed. Moreover, as soon as we are dealing with a number of keywords that can be representative of real-world needs, the state machine implementation proves also to be significantly faster than the naive indexer.

One of the primary goals of this research project was to leverage the effect of parallelization in reducing the time to index a document collection. We successfully demonstrated that performance gains could be achieved for the indexing task through parallelization as well as the fact that our architecture offers diminishing returns as the number of nodes is increased and as the corpus becomes larger.

Other avenues of interest include, but are not limited to:

- Compare the Aho-Corasick state machine indexer implementation to less naïve indexer implementations.
- Use compression techniques to our algorithm in order to reduce the index size in the database.
- Conduct further parallelization experiments to determine the performance characteristics of our indexer over a wider range of operating conditions.
- Study the time performance improvements possible deploying the Aho-Corasick state machine indexer within a commercial *MapReduce* framework such as *Hadoop*.

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun.ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] E. Price. 10,000 common english words. MIT Computer Science and Artificial Intelligence Laboratory. [Online]. Available: http://www.mit.edu/ ecprice/wordlist.10000