# CS6999 Web Data and Text Mining: Parallel Text Indexer Project Report

Damien DuBios
Justin Kamerman 3335272
Raman Singh

May 12, 2011

## 1 Introduction

Briefly in at most one page describe the main idea of the project and why is it important

## 2 Literature Review

The Aho-Corasick string matching algorithm[1] is a kind of dictionary matching search algorithm that constructs a finite state machine to scan for a given set of keywords. It is, in effect, a reduced grammar regular expression parser described in [**?**]. In our search implementation, the finite state machine is constructed from a list of keywords. Then the documents of the test corpus are read from secondary storage and fed through the finite state machine. As soon as a document is found to contain at least one instance of each of a set of search terms, parsing of that document ends.

## 3 Motivation

## 4 Research Plan

## 5 Design Component

A black-box functional view of the indexing system is show in figure 1. The system proceses a continuous incoming flow of documents and distribute them to parallel indexing threads running on physically seprate, heterogenous nodes. Document queries are performed using an evolving search index. The operational balance is to timeously index new additions to the corpus while servicing concurrent search requests. A view as to how the components of the indexer system are deployed is show in figure 2.

As can be seen in figure 2, the indexer processes are symmetrical and deployed on muliple physical nodes. The individual indexers do not interact directly with one another, making for a simple deployment and operation model. The execution loop of each indexer is as follows:

1. Initialize the indexer by retrieving a collection of index keywords and synonyms from a relational databse. These keywords are used to construct

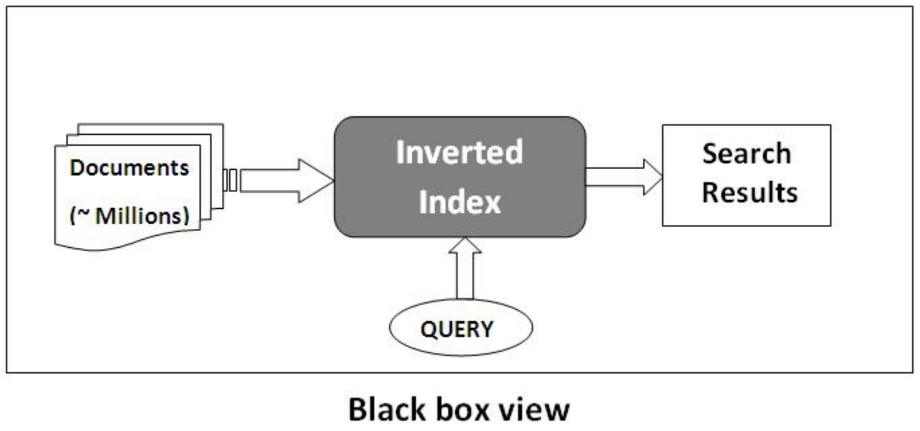

Figure 1: A black-box view of the indexer

a lexical parser which will be used to scan and index documents.

2. Retrieve a batch of unprocessed documents from a relational database. The document batch will be sized according to the physical capabilities of each node. In this implementation, this tuning task is a manual exercise but future enhancements may include an adaptive loading component.
3. Parse each document retrieved and contruct an inverted index representing the batch. This *delta* index, as we shall call it, is then used to augment the global index maintained in a relational database.
4. Repeat from step 2.

As shown in figure 2, the searcher component can be executed from any node which has access to the relational database housing the document index. The execution path of a single search query wold be as follows:

1. Canonize search terms based on keyword synonyms defined in the keyword store.
2. Execute a boolean query against the inverted index.
3. Return the list of corpus documents containing the intersection of the canons of the search terms.

A guiding architectural principal of the indexer design is to separate the implementation into three logical layers:

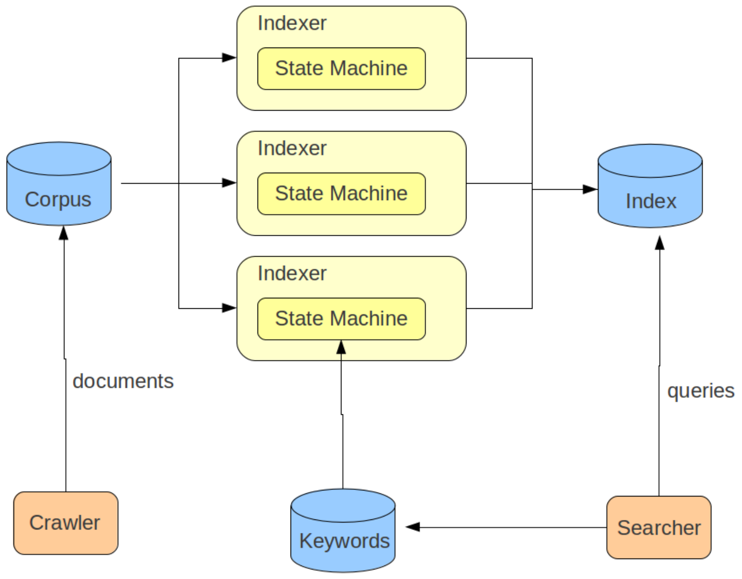

Figure 2: Indexer deployment model

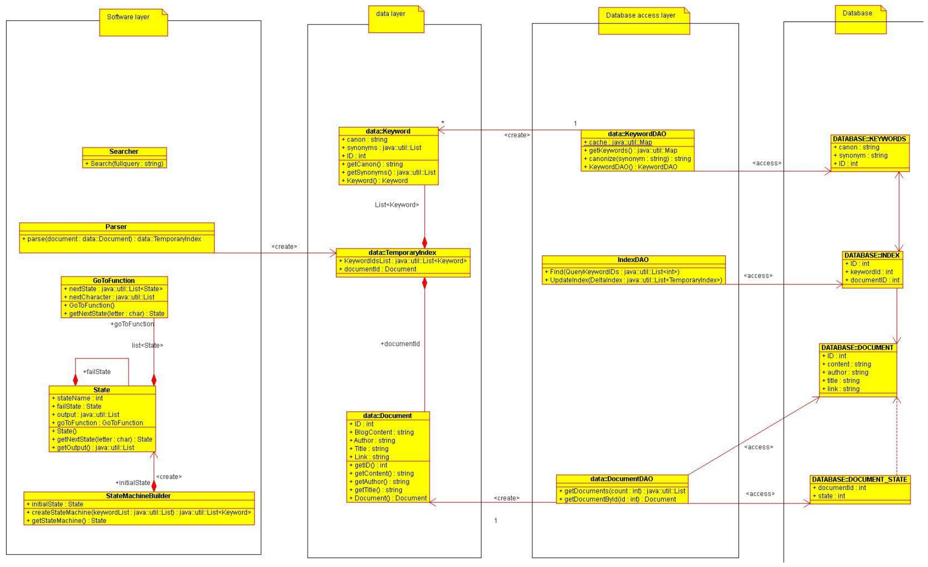

Figure 3: Overall UML class diagram design

- **Software Layer:** implements parsing, indexing and search algorithms.
- **Data Access Layer:** implements an *object-relational mapping*, insulating algorithm implementation from the persistence details. This layer implements classes which encapsulate persistent entities within the system.
- **Database:** the persistent store for the document collection and inverted index.

The implementation classes and their relationships are represented in a UML class diagram in figure 3. Figure 4 is a UML sequence diagram showing how the layers interact at a high level. The implementation of each of these layers is described in more detail in section 6.

## 6 Implementation

The individual worker processes of the indexer system are implemented as Java programs. The programs make use of the following thirdparty libraries:

- **DBPool:** a database connection pooling library, used to manage connections to relational databases.

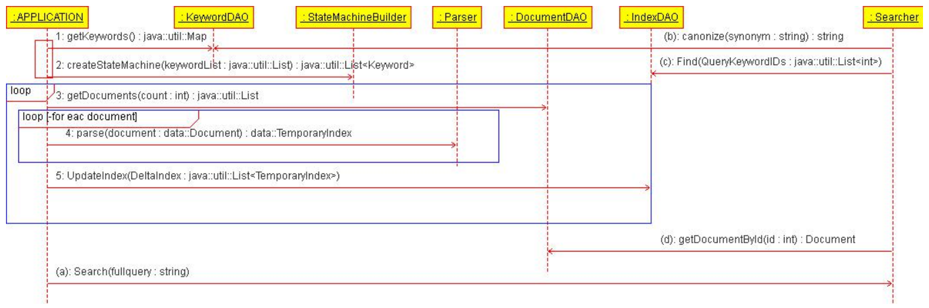
: APPLICATION
: KeywordDAO
: StateMachineBuilder
: Parser
: DocumentDAO
: IndexDAO
: Searcher
1: getKeywords() : java::util::Map
(b): canonize(synonym : string) : string
2: createStateMachine(keywordList : java::util::List) : java::util::List<Keyword>
(c): Find(QueryKeywordIDs : java::util::List<int>)
loop
3: getDocuments(count : int) : java::util::List
loop [-for eac document]
4: parse(document : data::Document) : data::TemporaryIndex
5: UpdateIndex(DeltaIndex : java::util::List<TemporaryIndex>)
(d): getDocumentById(id : int) : Document
(a): Search(fullquery : string)

Figure 4: Overall UML sequence diagram design

- **Apache Commons Logging:** a generic logging API used by DBPool.
- **Apache Commons CLI:** a library for parsing command line arguments passed to Java programs.
- **MySQL JDBC driver:** the MySQL Java client driver.

The indexer system maintains an *inverted index* of an evolving corpus and services concurrent search requests against the index. The current implementation is based on boolean retrieval methods described in [2]. An Aho-Corasick state machine [1] is used to scan documents for a predefined set of keywords, and construct a term index for each document (delta index). The term occurrences for each document are used to augment an existing index for documents already processed. The document collection, index, and keyword list are maintained in a single MySQL database instance. It is through this database that the parallel indexer instances are initialized and synchronized. This configuraion is operationally simple and easy to implement but has the potential to become a throttle point as the system is scaled to larger and faster-evolving document collections, and is required to service a higher rate of queries. In order to scale the current implementation to large document collections, a more efficient mechanism would be required in this repect.

The following subsections detail the implementation of the architectural layers identified in section 5.

## 6.1 Data Access Layer

Access to persistent storage is via objects of the data access layer. Confining data access to a logical grouping of classes insulates the other application components from changes in database schema.

### 6.1.1 Document

The DocumentDAO singleton class provides methods for retrieving Document objects from the document database: List ¡Document¿ getDocuments (int count): retrieve a specified number of unprocessed documents from the database. The documents retrieved will be atomically marked as processed so that processing is not duplicated on other nodes. In this scheme, a document is considered processed to all other processes once retrieved from the database. This may cause problems from a fault recovery perspective if the processing node fails. A proposed enhancement to this scheme is to use another state, processing, to indicate that the document has been retrieved and to update the state to a processed state once processing is actually complete. For the initial implementation we will use the processed state only. Document getDocumentById (int id): retrieve a specific document from the database, referenced by its ID. Int getID (): accessor method. String getContent (): accessor method. String getAuthor (): accessor method. String getTitle (): accessor method String getLink(): accessor method.

The Document class encapsulates the BLOG database table, with the addition of a state field that tracks whether a document has been processed or not. This field will be implemented as a new field on the BLOG table or as a new reference table DOCUMENT$_STATE.andshowUMLclassandsequencediagramsrespectivelyfornon-$
$trivialclassesandmethodsinvolvedintheimplementationoftheDocumentdataaccesslayer.$

## 7 Experiment Results

FIX: All tests were run on a Dell laptop with an Intel Core Duo 2GHz processor, 1GB RAM, running a 32 bit Linux 2.6.37 kernel. The Java Virtual Machine used was version 1.6.0-24.

## 7.1 Aho-Corasick

Various tests were conducted to characterize the performance of the Aho-Corasick search algorithm:

- The time taken to construct the Aho-Corasick state machine was measured for different numbers of keywords. The results of this test is shown in

Figure 5: Aho-Corasick state machine construction

Figure 6: Aho-Corasick state machine scanning

figure 5 and construction time can easily be considered linear with respect to the number of keywords. This is consistent with [1] which 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.

- The time taken for the Aho-Corasick state machine to scan different sized corpora was measured. The test was repeated for various state machines, constructed using a 100, 200, and 400 keyword set. The results of this test are shown on figure 6 and indicate that scan time is nearly linear with respect to the size of the corpus and independent of the number of keywords used to generate the state machine. This last result is consistent with [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.

- The time taken to search for different numbers of keywords was measured over different sized corpora. A set of ten keywords was selected randomly and the the corpora searched for $\binom{n}{k}$ enumerated combinations thereof to obtain an average for a particular keyword set size. The results of this test are shown in figure 7 and indicate that search time is independent of the number of terms used in the search and linear with respect to the size of the corpus. This first result is surprising given that the search process exists as soon as a single instance of each search term is found. One would expect the number of documents scanned completely to increase as the number of search term increases. The small average document size and the fact that the search hit rate is low may account for this phenomenon.

Figure 7: Aho-Corasick state machine search

Figure 8: Inverted index creation

Figure 9: Inverted index search

### Inverted Index

Various tests were conducted to characterize the performance of the inverted index search algorithm:

- The time taken to construct the inverted index was measured for different sized corpora. The results of this test is shown in figure 8 and, as expected, indicate a linear relationship between index construction time and the size of the corpora.
- The time taken to search for different numbers of keywords was measured over different sized corpora. A set of ten keywords was selected randomly and the the corpora searched for $\binom{n}{k}$ enumerated combinations thereof to obtain an average for a particular keyword set size. The results of this test are shown in figure 9. If the results for one and ten search words are ignored, search times appear to be independent of the number of search terms and close to linear with respect to the size of the corpus. The unusual results for one and ten keywords warrant further investigation.

## 7.2 Comparison

Comparing corpus scan times for Aho-Corasick and inverted index implementations, figures 6 and 8 respectively, indicates that the inverted index scanner is significantly faster than the Aho-Corasick state machine. Given that the grammar supported by the JFlex-generated scanner is more expressive than that of the Aho-Corasick state machine, one would expect results to more comparable. Reviewing the source code of the generated scanner indicates a compact and efficient implementation using packed data structures to represent the internal state machine. In comparison, the Aho-Corasick implementation is relatively unoptimized, closely following the pseudo-code in [1].

Comparing search times for Aho-Corasick and inverted index implementations, figures 7 and 9 respectively, indicates that the inverted index implementation has an advantage of several orders of magnitude. The initial cost of creating the inverted index is significantly larger than the creation of the Aho-Corasick state machine however this cost is amortized over each subsequent search. The Aho-Corasick implementation rescans the entire corpus for each search, as opposed to a scan of the inverted index alone. This exercise is a classic demonstration of the impetus behind using indexed search techniques for large corpora.

## 8 Conclusions

# Bibliography

[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] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008.