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

Damien DuBios
Justin Kamerman 3335272
Raman Singh

May 13, 2011

1 Introduction

Fundamental to the real-world application of *Information Retrieval* techniques is the ability to objectively measure the quality of a particular algorithm or system and compare it to others. So often, information retrieval algorithms lack an analytical model that would allow one to make formal statements as to the performance boundaries of a particular system. In addition, as with machine learning and classification problems, using a scalar value to represent the performance of a system as a whole is misleading and does not account for the variety of operating conditions under which the system may be expected to perform.

In measuring the performance of an information retrieval system, the following factors constitute a set of operating conditions which qualify the result:

- Accuracy of the retrieval result with respect to the query. Common measures in this respect are *Precision* and *Recall*. Search techniques employing latent semantics of document collections are subtly more difficult to quantify, though measurable nonetheless.
- Speed with which queries are answered. This metric is a function of the number of search terms and the size of the corpus.
- Size of the document collection and, if not static, the rate at which new documents are being added and/or removed.
- The computational cost of the technique. Multiple activities may contribute to this aspect: searching, indexing, preprocessing, and post processing. It is important to consider whether the technique allows indices or intermediate approximations of the document collection to be altered incrementally or whether they must be regenerated to incorporate new documents. Also significant is whether the technique can be parallelized and thereby benefit from concurrency.
- The storage requirements of the technique. How do the storage requirements evolve with the size of the collection and/or degree of concurrency employed ?

To drive information retrieval research in one direction or another, it would be useful to have at ones disposal a physical implementation framework within which one could empirically explore the operating parameters of a particular

algorithm and/or technique along the lines described above. Such a system would allow researchers to investigate how the system responds to variation in any one of these operating parameters. Does the query speed increase as more computational capacity is added ? How do storage requirements vary with the size of the document collection ? The aim of this project is to create such an information retrieval *test-bed*.

The initial reference implementation of this information retrieval test-bed is based on a simple *inverted index* but has been architected such that the specific algorithm could easily be changed without major rework. This paper presents a survey of existing literature in support of our test-bed concept, as well as the specific algorithmic techniques embodied in the reference implementation. In addition, a detailed architectural overview of the system is presented, giving insight into how the system would be modified for alternative algorithms. Lastly, an evaluation of our inverted index reference implementation is conducted and the results analyzed.

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 [2].

3 Motivation

4 Research Plan

5 Design Component

A black-box functional view of the indexing system is show in figure 1. The system processes a continuous incoming flow of documents and distribute them to parallel indexing threads running on physically separate, heterogeneous 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 multiple 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 database. These keywords are used to construct 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 construct 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:

- **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.

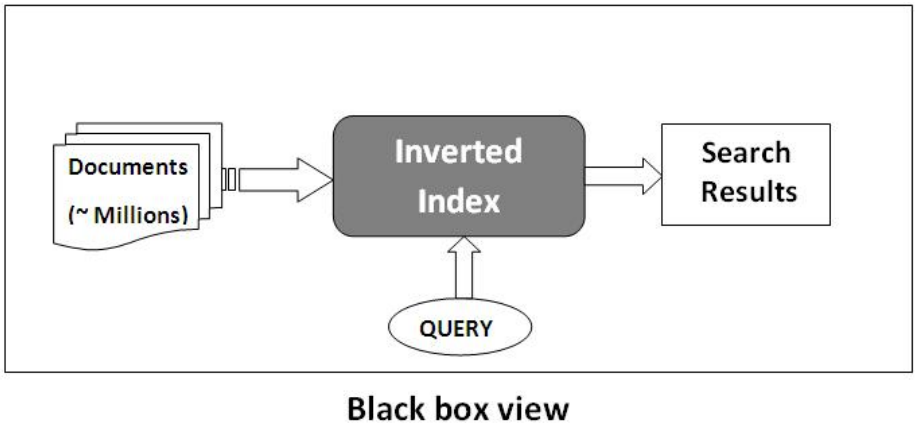

Figure 1: A black-box view of the indexer

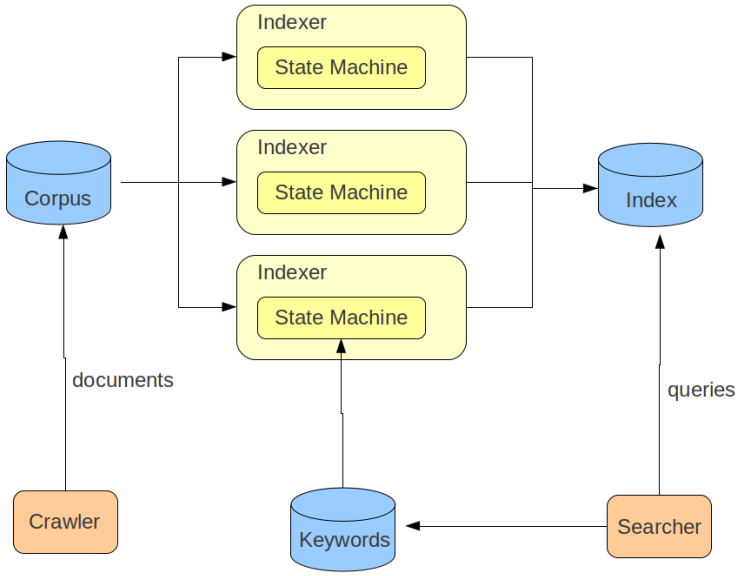

Figure 2: Indexer deployment model

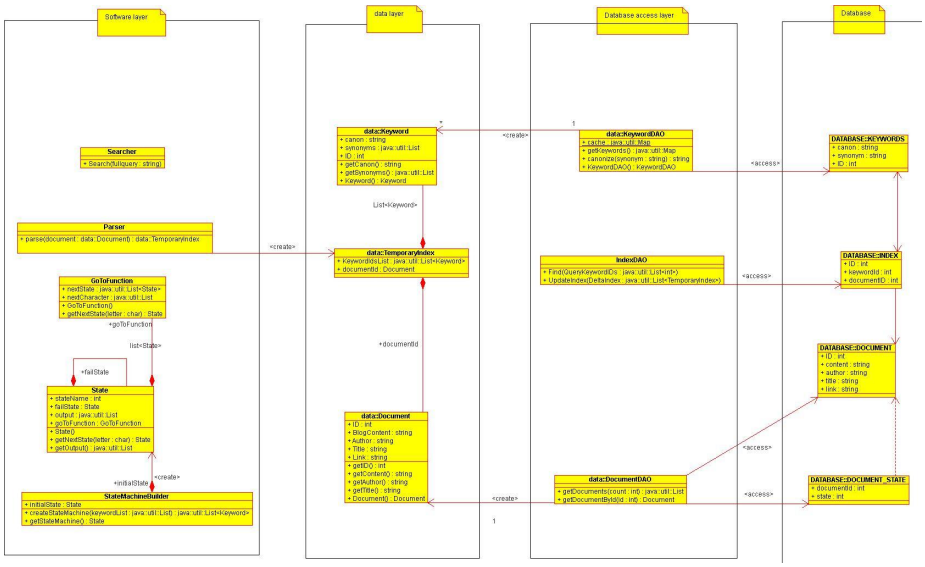

Figure 3: Overall UML class diagram design

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.
- **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 implemen-

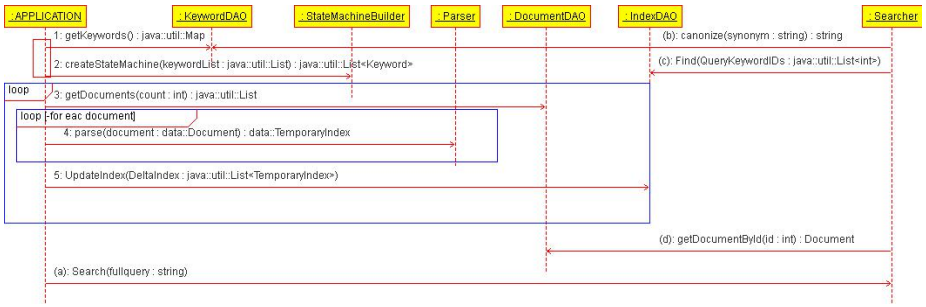

Figure 4: Overall UML sequence diagram design

tation is based on boolean retrieval methods described in [3]. 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 configuration is operationally simple and easy to implement but the database 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 respect.

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. The following subsections detail the individual data entities handled by the data access later.

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. Figures 5 and 6 show class and sequence diagrams respectively for non-trivial classes and methods involved in the implementation of the Document data access layer.

6.1.2 Keyword

The `KeywordDAO` singleton class provides methods for retrieving `Keyword` objects from the document database. Because the keyword set is relatively static and accessed often, the full set of keywords will be loaded from the database and cached when the class initializes. The class also provides methods for converting search terms to their canonical form.

- `List<String> getKeywords ()`: this method will returns a list of all `Keyword` objects defined in the keyword database.

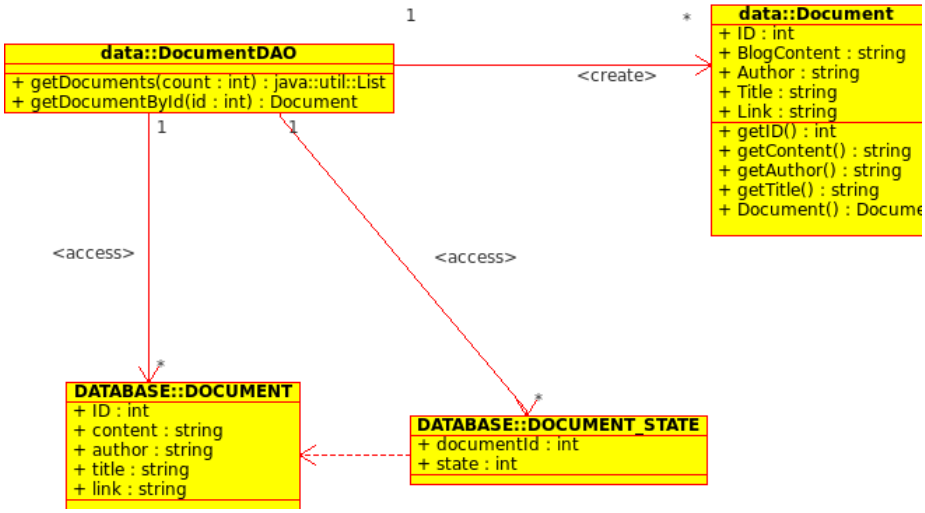

Figure 5: Document class diagram

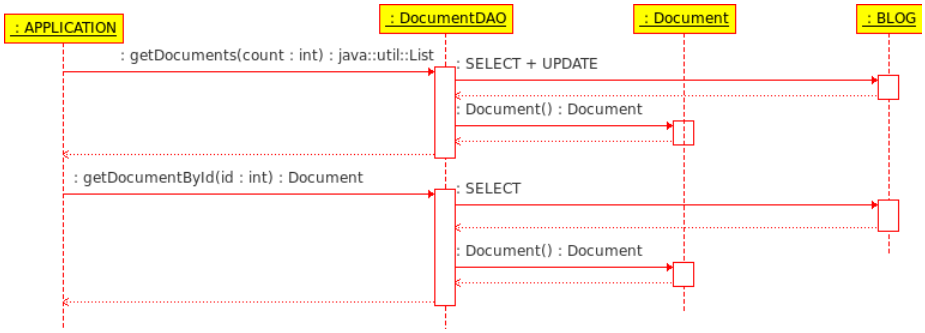

Figure 6: Document sequence diagram

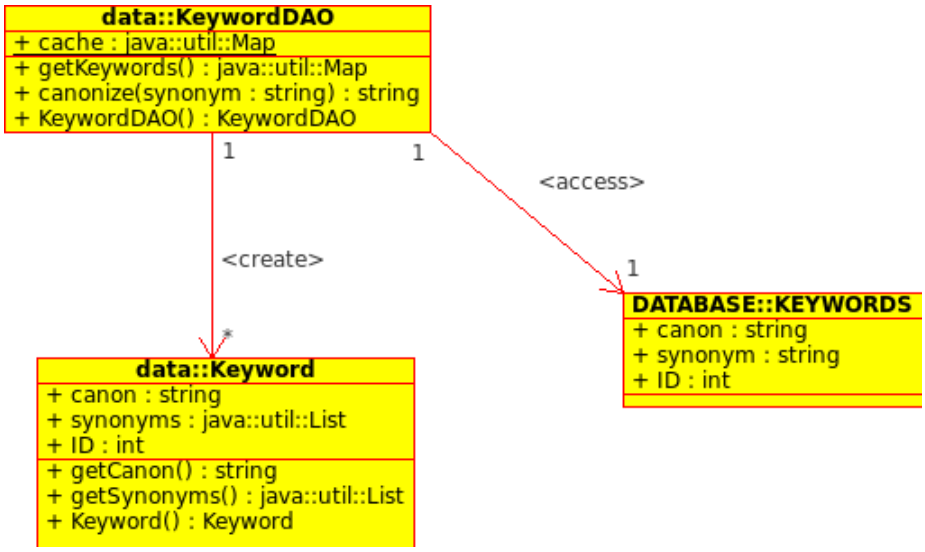

Figure 7: Keyword class diagram

- `List<Integer> canonize (String query)`: return the list of IDs corresponding to a comma-separated list of query terms.
- `Keyword getKeywordById (int id)`: return the `Keyword` object with the given identifier.

The `Keyword` field encapsulates a canonical search term and a list of its synonyms. Figures 7 and 8 show UML class and sequence diagrams respectively for non-trivial classes and methods involved in the implementation of the Keyword data access layer.

6.1.3 Index

The `IndexDAO` class provides methods for updating and searching the inverted index table stored in the database. There are two methods defined in this class:

- `void UpdateIndex (List<TempIndex> input)`: this method takes input of type List¡TempIndex¿ (Note: Temporary index is a user defined data class, which contains Document ID and List of ¡keyword ids¿ as its attributes) and generates an SQL query which updates the INDEX table in the Database.

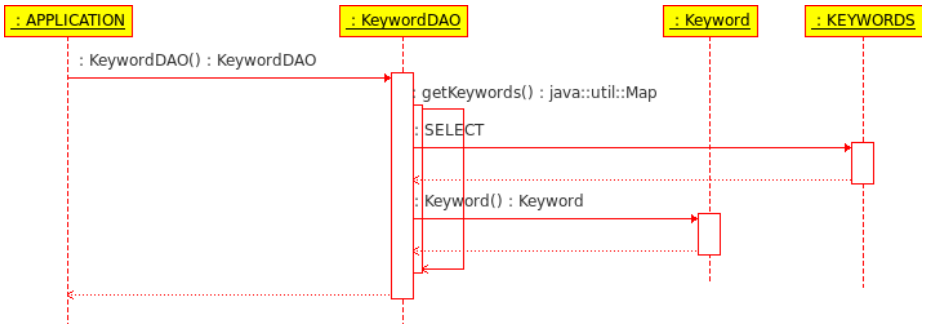

Figure 8: Keyword sequence diagram

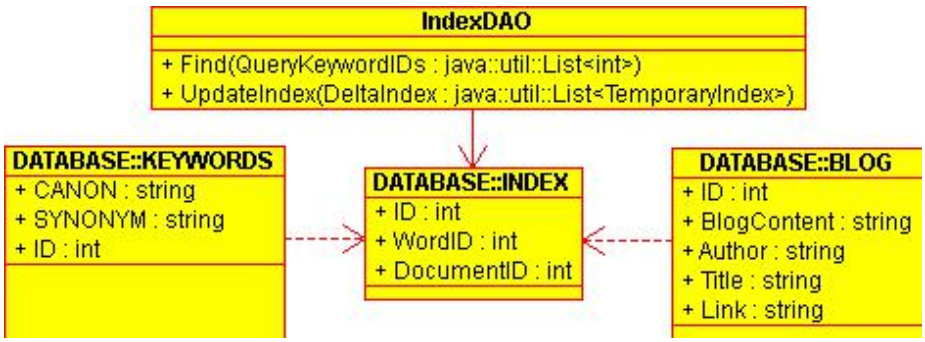

Figure 9: Index class diagram

- `List<Integer> Find (List<Integer> QueryKeywordIds)`: this method takes a list of `Integers` (which corresponds to mapped `String` query into keyword ids) as input and searches the INDEX table in DB to find all the document ids which are common between for the given query keywords.

Figures 9 and 10 shows the UML class and sequence diagrams respectively for classes and methods involved in the implementation of the Index data access layer.

6.2 Software Layer

6.2.1 Searcher

This class is designed to perform searches for a query entered by the user. It contains a method named `search ()` which is responsible for finding all the

Figure 10: Index sequence diagram

documents which contained all query keywords.

- `List<Document> search (String)`: given a comma-separated list of search terms, returns the list of documents in which the canonical form of all search terms appear together. This method First calls `List<int> KeywordDAO.canonize (String)`, to map string query keywords into corresponding list of keyword IDs. Note: We are assuming that the query will be formed with the set of keywords, which will be a subset of the global keyword set, with which the parsing state machine is constructed.

  Then it calls `List<Integer> IndexDAO.find (List<querykeywordIDs>)`, which will searches in INDEX DB for a list of Common DOC_IDs corresponding to given query keywords.

  In the end, it calls `List<Document> DocumentDAO.getDocumentById (DOC_ID)`. This will map DOC_ID into a `Document` from the BLOG table in the database.

Figures 11 and 12 shows the UML class and sequence diagrams respectively for classes and methods involved in the implementation of the `Searcher` class.

6.2.2 State Machine

This class is designed to build the state machine from the list of keywords returned by the `KeywordDAO.getKeywords()` method. This class has a static attribute `initialState` that is the first state of the state machine built by the function `createStateMachine` (initializes with the value null). The first state

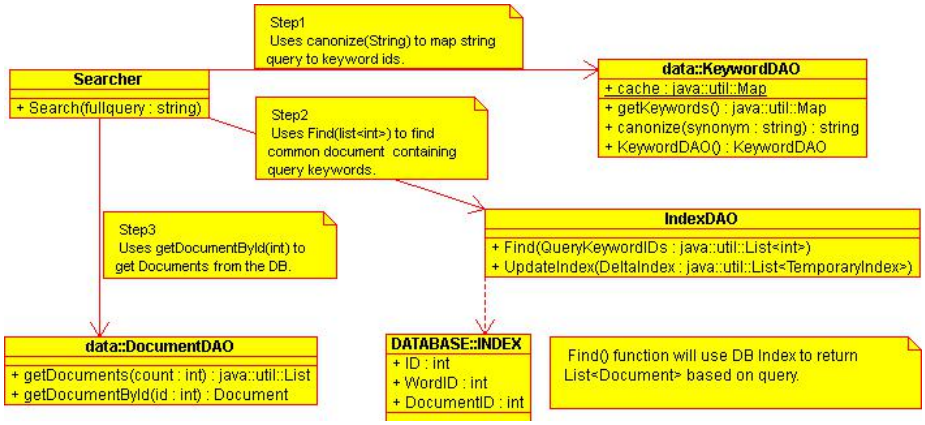

Figure 11: Searcher class diagram

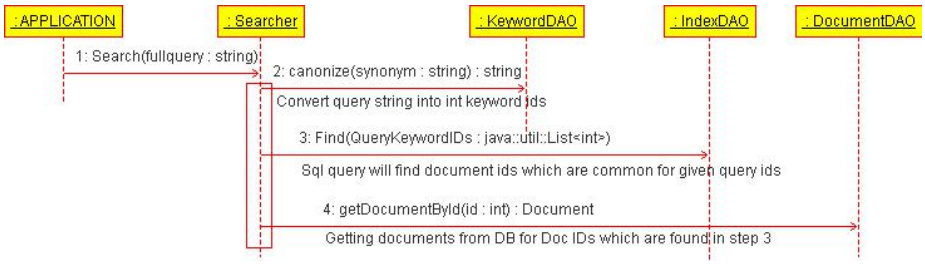

Figure 12: Searcher sequence diagram

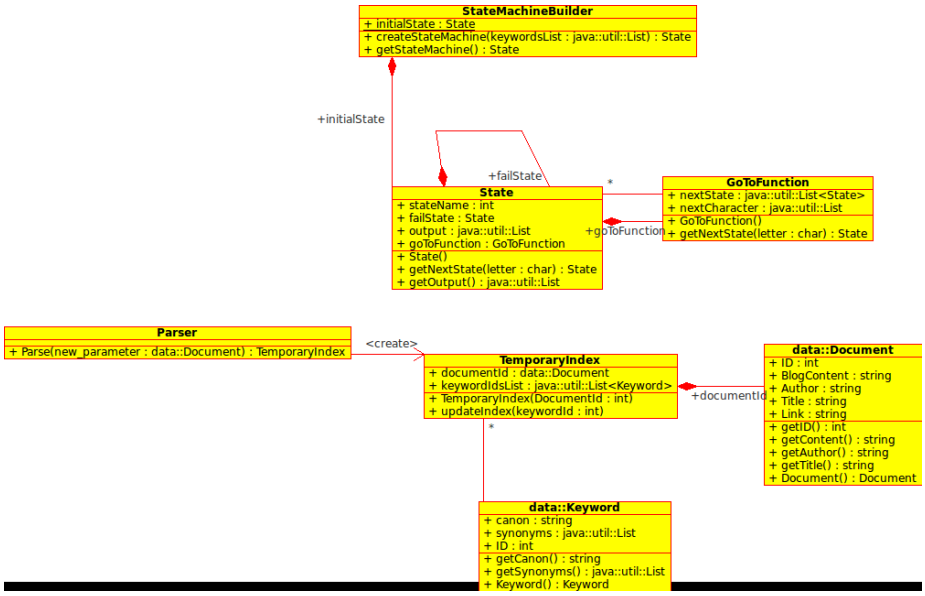

Figure 13: State machine class diagram

gives us the whole state machine thanks to the architecture of a state. Once the state machine is built, the function `getStateMachine` is used to access the initial state stored in the class.

Figures 13 and 14 shows the UML class and sequence diagrams respectively for classes and methods involved in the implementation of the `StateMachineBuilder`.

6.2.3 Parser

The `Parser` implements the document parsing algorithm within the state machine. By default, it uses the state machine contained in the static attribute of `StateMachineBuilder`. To parse a document we need to call the `parse ()` function inside this class, passing it the `Document` class encapsulating the actual text we want to parse. This class returns a `TemporaryIndex` object which encapsulates a reference to the document parsed and the keywords found to occur within the document.

Figures 15 and 16 shows the UML class and sequence diagrams respectively for classes and methods involved in the implementation of the `Parser`.

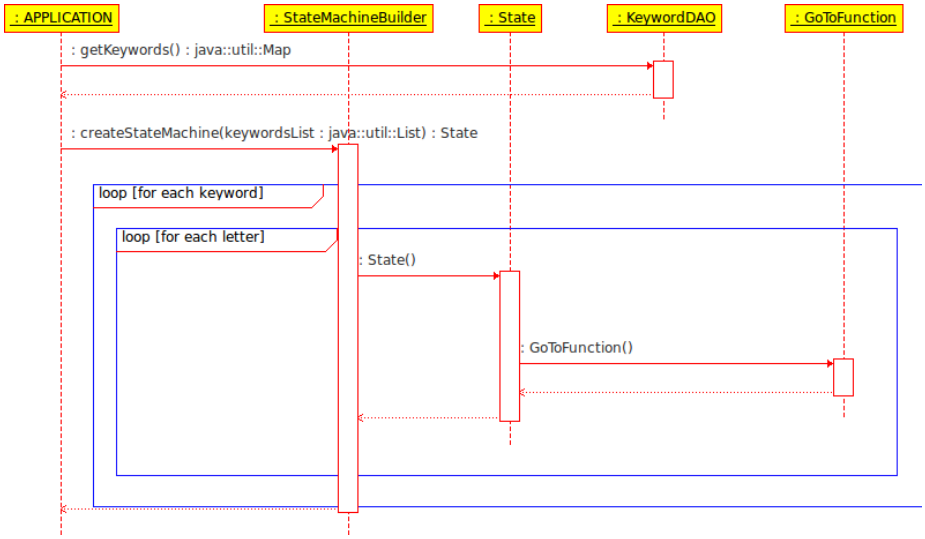

Figure 14: State machine sequence diagram

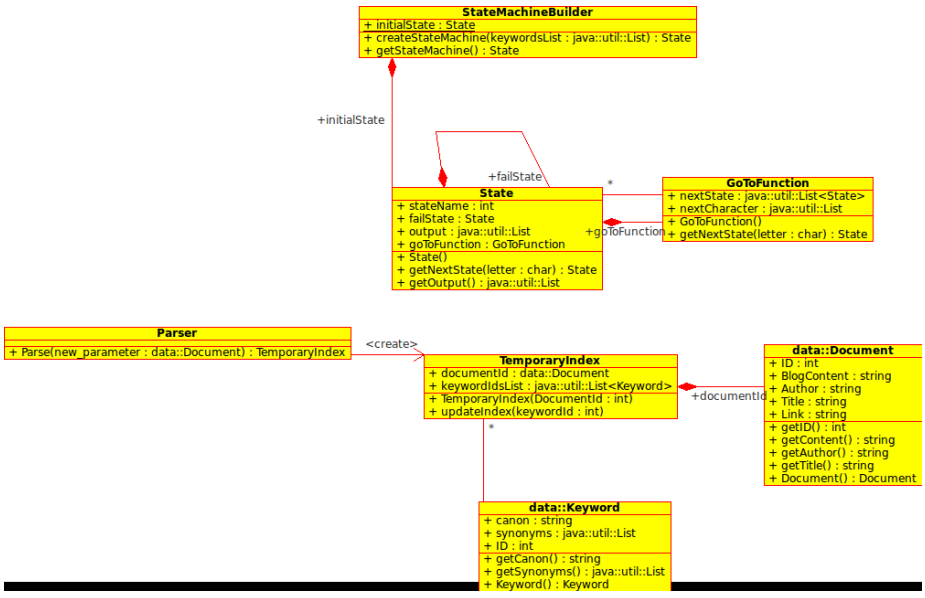

Figure 15: Parser class diagram

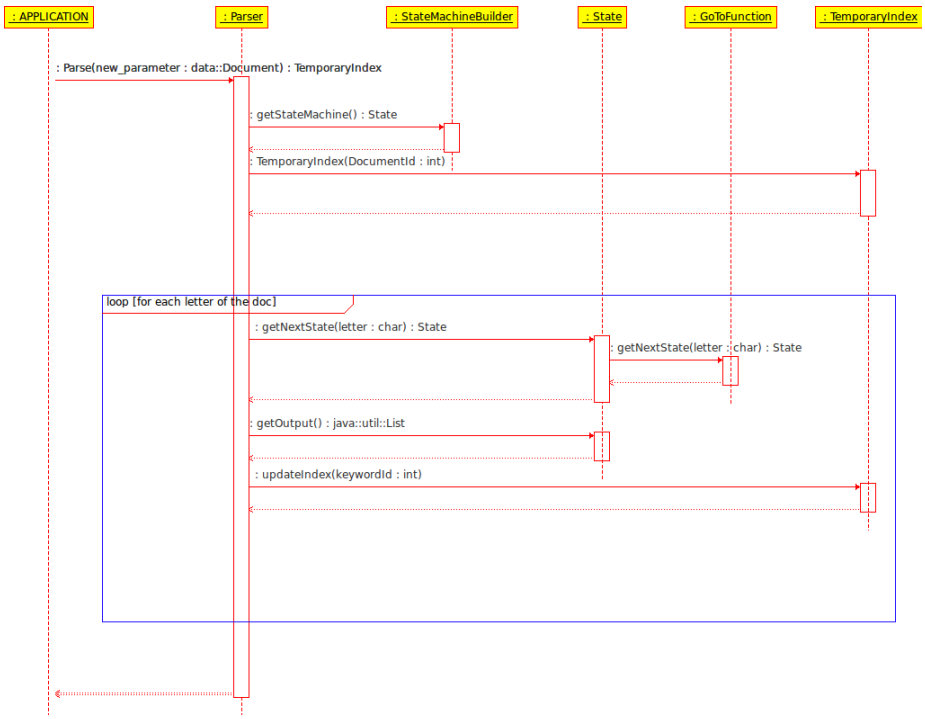

Figure 16: Parser sequence diagram

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.....

8 Conclusions

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