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

Damien DuBois
Justin Kamerman 3335272
Ramanpreet Singh 3395913

June 6, 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. Often, information retrieval algorithms lack an analytical model that would allow one to make formal statements as to the performance characteristics 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 ?

To drive information retrieval research in one direction or another, it would be useful to have a physical implementation framework within which one could empirically explore the operating characteristics 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. 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 significant rework. This paper presents a survey of existing literature in support of our test-bed concept, as well as the specific algorithms employed 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

2.1 Indexing

Information Retrieval is a fast becoming the dominant form of information access, overtaking traditional relational database style searching [8]. 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.

2.2 Evaluation

In the case of machine learning and classification systems, the specification of a particular set of misclassification costs or false and positive misclassification rates define a particular *operating point* of the system.

This concept applies equally to *information retrieval* systems whereby it is misleading to represent the operating characteristics of a particular system with a scalar metric. In addition, the evaluation of information retrieval systems suffers from the the inherent subjectivity with respect to *relevance* [9]. Non-scalar evaluation methods try to characterize the performance of a system over a range of operating conditions. Although this makes direct performance comparisons ambiguous, we are more likely to be able to select the algorithm which will perform best in the real world by applying domain expertise to the results of a non-scalar evaluation [3].

2.3 Parallelization

Once an objective measure has been established, the performance of an information retrieval system can be improved iteratively by making various optimizations. These optimizations may take the form of algorithm enhancements, implementation efficiencies, or parallelization.

Algorithm enhancements usually represent the forefront of research in the field. Work to validate a new concept or algorithm is usually performed in a very controlled environment and on relatively small document collections. As the field matures, improvements are harder fought and often involve increasingly complex algorithms and corpus approximation methods into which it becomes more difficult to *fold-in* new documents. In addressing the considerable computation cost of re-calculating *LSI* [4] approximations, [11] introduced new update methods which saw improved average precisions but were forced to conclude that the improvements were at a computational premium over previous methods. A semi-discrete matrix decomposition for LSI, *SDD*, performs as well as the original SVD-based LSI method for equal query times and uses less storage [6]. The decomposition time of the new method, however, is significantly larger than the original.

Enhancement of implementation efficiency in [7] saw a six-fold timing improvement in *LSI* searches on large document collections. The same study achieved nearly 180 times improvement through the use of parallelization. The study made no algorithmic improvements and the result is consistent with increased use of parallelization by large commercial systems to cope with the ever increasing bodies of data that must be processed.

Map-Reduce, popularized by Google, is emerging as an important programming model for large-scale data-parallel applications such as web indexing [5].

Fundamentally, Map-Reduce breaks up a computation into smaller tasks which are made to run in parallel on different physical machines. The *map* and *reduce* phases are inspired by the functions of the same name used in *functional programming*. The model scales well to large clusters mostly because of the decoupled nature of the tasks themselves. An initial *map* phase breaks the data into separate chunks and assigns them to a task. The *reduce* phase assembles the results of the map tasks to produce the final result.

Hadoop is an open source implementation of the MapReduce model, developed mostly by Yahoo, and now maintained by the Apache Software Foundation. Hadoop can be configured to automatically create speculative copies of slow running map tasks (stragglers). This facility is intended as a pre-emptive failure recovery mechanism as well as a means of optimizing overall throughput. The metrics used to identify stragglers as well as the algorithm for determining the target node on which speculative copies of a task should be launched, is examined in [10]. The paper introduces an alternative scheduling algorithm, *LATE*, to address excessive task speculation issues with the native Hadoop scheduler in a heterogeneous hardware environment. Experiments comparing the native Hadoop scheduler, a non-speculative scheduling strategy, and LATE were conducted using hosted virtual machines (VMs) in the Amazon EC2 environment as well as a dedicated local cluster. Variance in node performance was introduced by varying the distribution of VMs across physical machines and also introducing network and disk transfer loads. Results show speculative execution (native Hadoop and LATE) outperforming non-speculative execution and LATE outperforming native Hadoop in most cases.

3 Research Plan

The research design and methods section should include the following:

- an overview of the experimental design
- a detailed description of specific methods to be employed to accomplish the specific aims
- a detailed discussion of the way in which the results will be collected, analyzed, and interpreted
- a projected sequence or timetable (work plan)

- a description of any new methodology used and why it represents an improvement over the existing ones
- a discussion of potential difficulties and limitations and how these will be overcome or mitigated
- expected results, and alternative approaches that will be used if unexpected results are found
- precautions to be exercised with respect to any procedures, situations, or materials that may be hazardous to personnel or human subjects

Suggestion:

Number the sections in this part of the application to correspond to the numbers of the Specific Aims.

Give sufficient detail. Do not assume that the reviewers will know how you intend to proceed.

Avoid excessive experimental detail by referring to publications that describe the methods to be employed.

Publications cited should be by the applicants, if at all possible. Citing someone else's publication establishes that you know what method to use, but citing your own (or that of a collaborator) establishes that the applicant personnel are experienced with the necessary techniques.

If relevant, explain why one approach or method will be used in preference to others. This establishes that the alternatives were not simply overlooked. Give not only the "how" but the "why."

If employing a complex technology for the fast time, take extra care to demonstrate familiarity with the experimental details and potential pitfalls. Add a coinvestigator or consultant experienced with the technology, if necessary.

Document proposed collaborations and offers of materials or reagents of restricted availability with letters from the individuals involved.

4 Design

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.

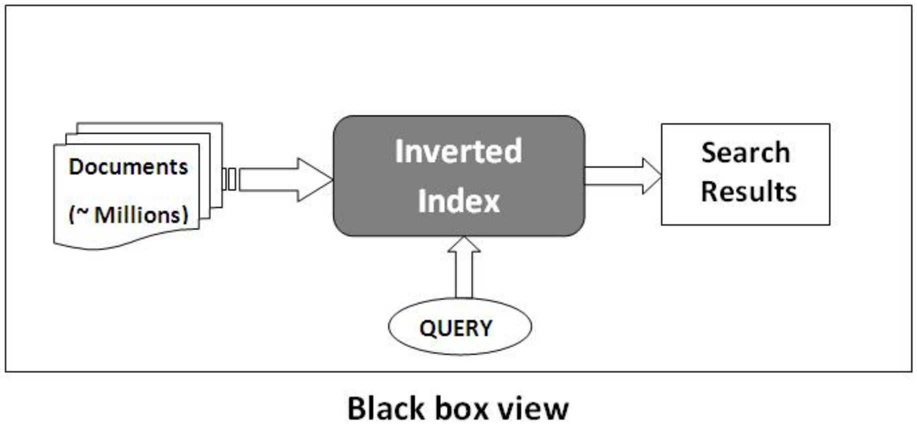

Figure 1: A black-box view of the indexer

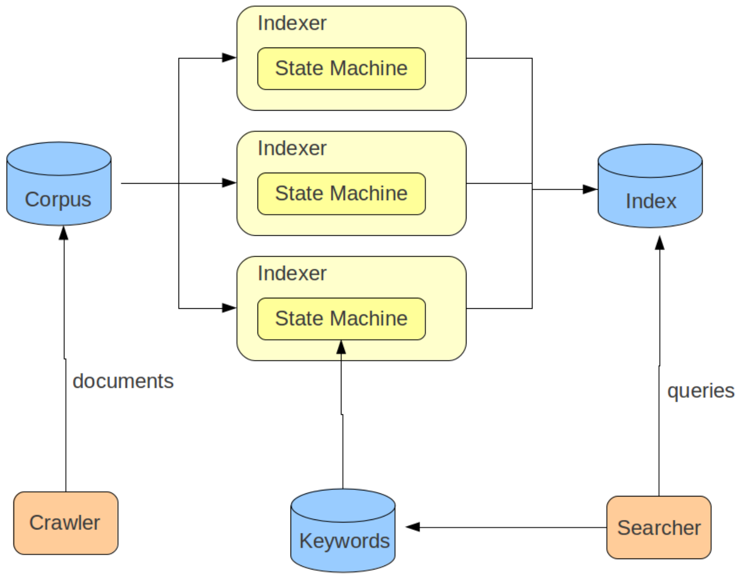

Figure 2: Indexer deployment model

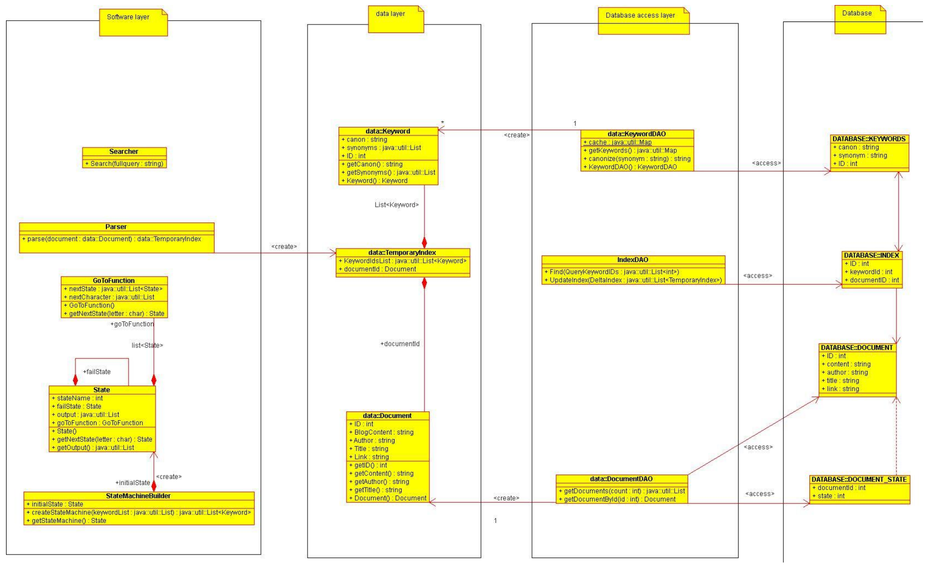

Figure 3: Overall UML class diagram design

- **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 5.

5 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.

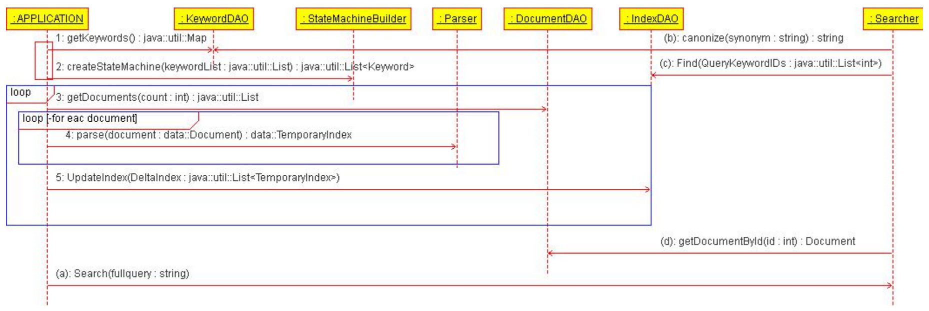
: APPLICATION
: KeywordDAO
: StateMachineBuilder
: Parser
: DocumentDAO
: IndexDAO
: Searcher
1: getKeywords() : java::util::Map
(b): canonize(synonym : string) : string
2: createStateMachine(keywordList : java::util::List) : java::util::List<Keyword>
(c): Find(QueryKeywordIDs : java::util::List<int>)
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

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 [8]. 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 4.

5.1 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.

5.1.1 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.

5.1.2 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.

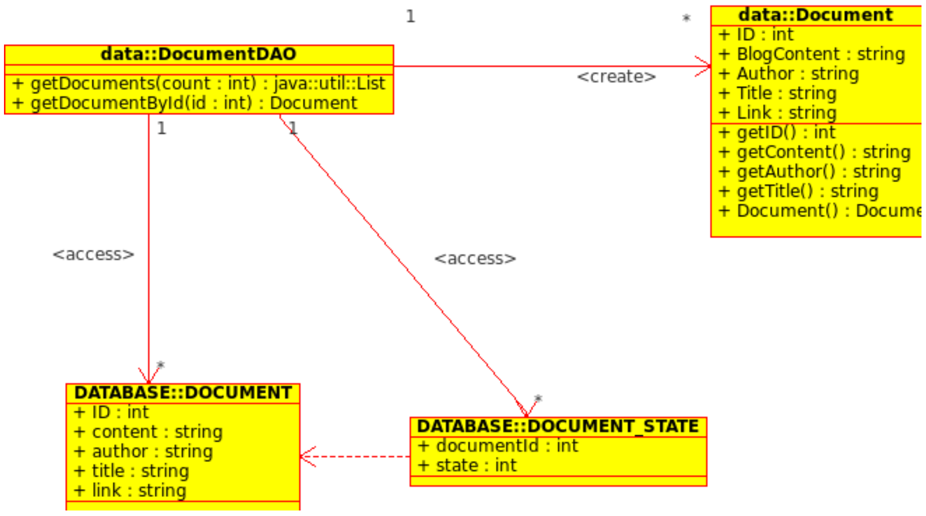

Figure 5: Document class diagram

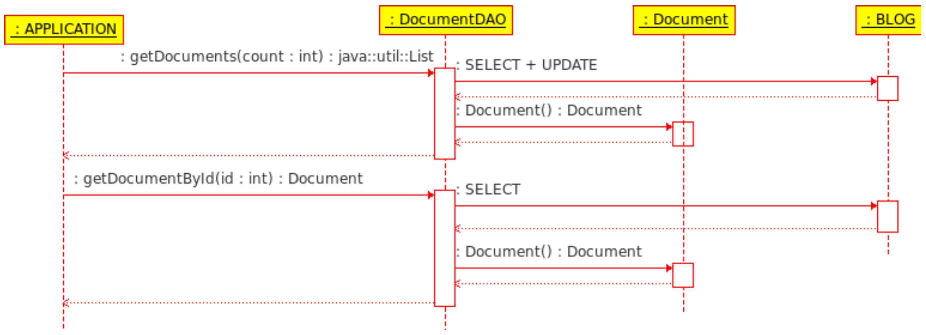

Figure 6: Document sequence diagram

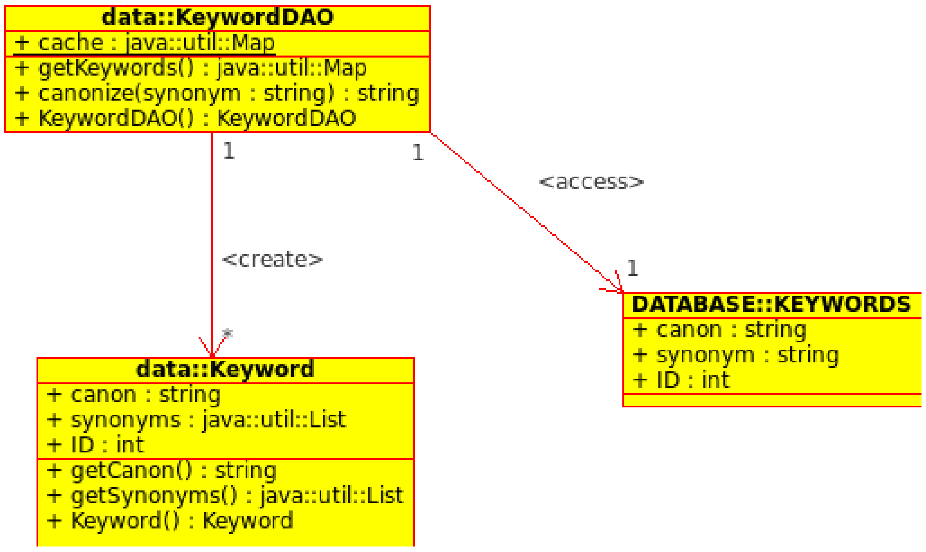
data::KeywordDAO
+ cache : java::util::Map
+ getKeywords() : java::util::Map
+ canonize(synonym : string) : string
+ KeywordDAO() : KeywordDAO
1
1
<access>
<create>
1
*
DATABASE::KEYWORDS
+ canon : string
+ synonym : string
+ ID : int
data::Keyword
+ canon : string
+ synonyms : java::util::List
+ ID : int
+ getCanon() : string
+ getSynonyms() : java::util::List
+ Keyword() : Keyword

Figure 7: Keyword class diagram

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

5.1.3 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

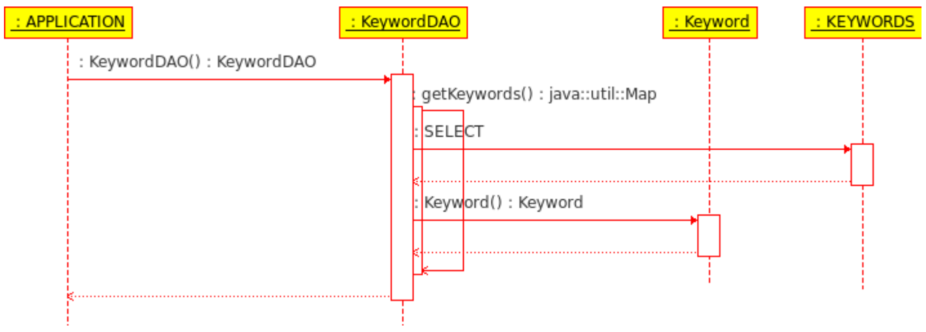

Figure 8: Keyword sequence diagram

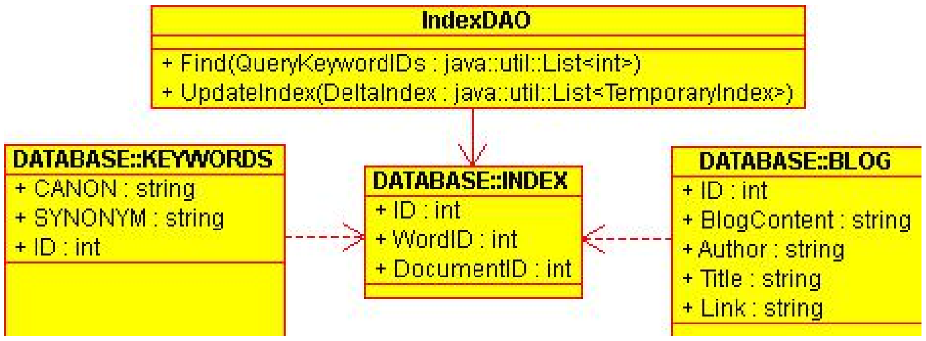

Figure 9: Index class diagram

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.

- `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.

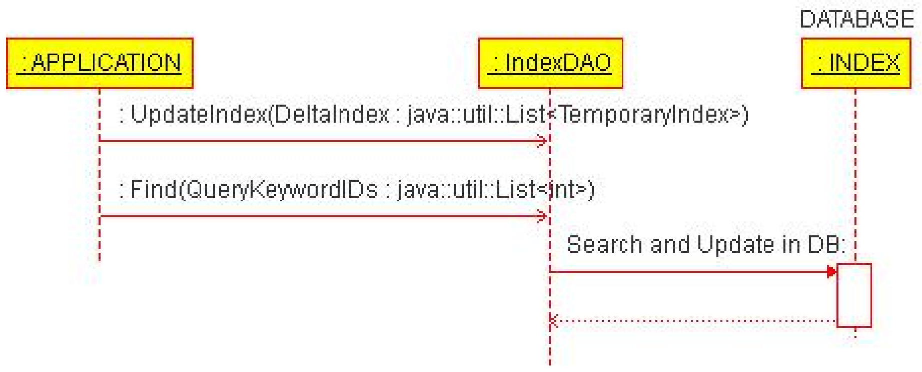
DATABASE
: APPLICATION
: IndexDAO
: INDEX
: UpdateIndex(DeltaIndex : java::util::List<TemporaryIndex>)
: Find(QueryKeywordIDs : java::util::List<int>)
Search and Update in DB:

Figure 10: Index sequence diagram

5.2 Software Layer

5.2.1 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 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.

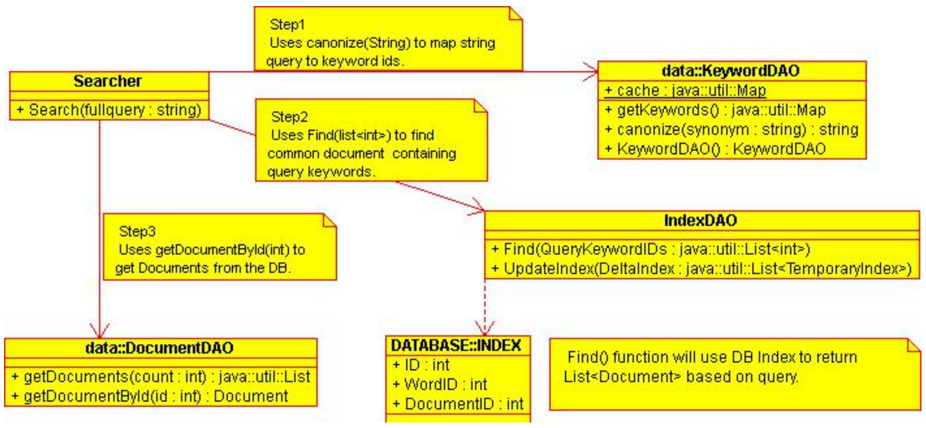
Step1
Uses canonize(String) to map string query to keyword ids.
Searcher
+ Search(fullquery : string)
data::KeywordDAO
+ cache : java::util::Map
+ getKeywords() : java::util::Map
+ canonize(synonym : string) : string
+ KeywordDAO() : KeywordDAO
Step2
Uses Find(list<int>) to find common document containing query keywords.
IndexDAO
+ Find(QueryKeywordIDs : java::util::List<int>)
+ UpdateIndex(DeltaIndex : java::util::List<TemporaryIndex>)
Step3
Uses getDocumentById(int) to get Documents from the DB.
data::DocumentDAO
+ getDocuments(count : int) : java::util::List
+ getDocumentById(id : int) : Document
DATABASE::INDEX
+ ID : int
+ WordID : int
+ DocumentID : int
Find() function will use DB Index to return List<Document> based on query.

Figure 11: Searcher class diagram

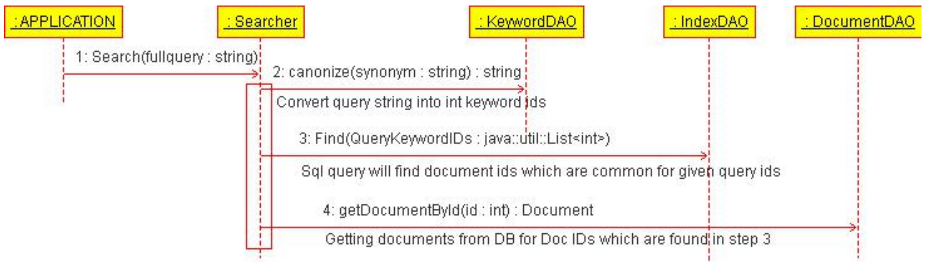
: APPLICATION
: Searcher
: KeywordDAO
: IndexDAO
: DocumentDAO
1: Search(fullquery : string)
2: canonize(synonym : string) : string
Convert query string into int keyword ids
3: Find(QueryKeywordIDs : java::util::List<int>)
Sql query will find document ids which are common for given query ids
4: getDocumentById(id : int) : Document
Getting documents from DB for Doc IDs which are found in step 3

Figure 12: Searcher sequence diagram

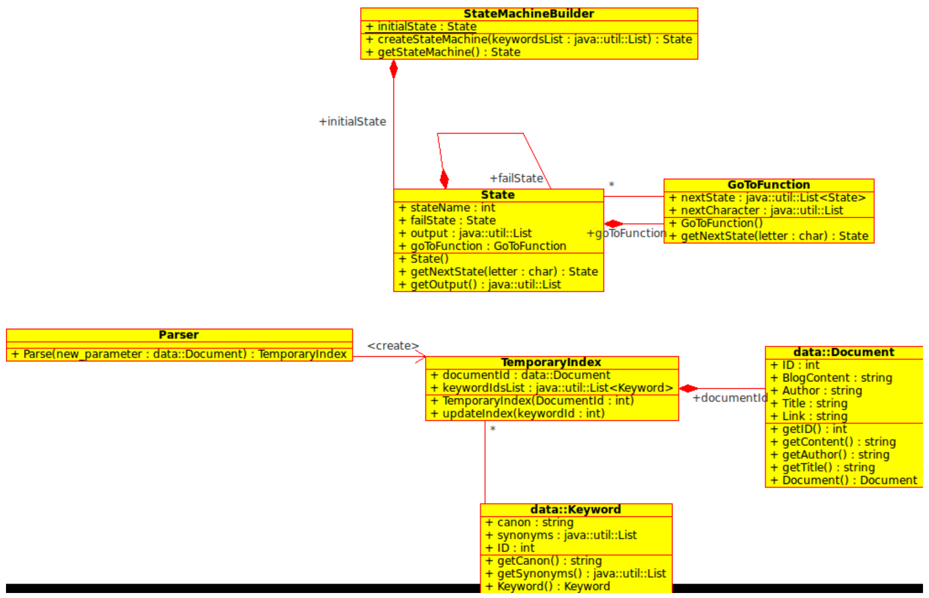
StateMachineBuilder
+ initialState : State
+ createStateMachine(keywordsList : java::util::List) : State
+ getStateMachine() : State
+initialState
+failState
*
State
+ stateName : int
+ failState : State
+ output : java::util::List
+ goToFunction : GoToFunction
+ State()
+ getNextState(letter : char) : State
+ getOutput() : java::util::List
+goToFunction
GoToFunction
+ nextState : java::util::List<State>
+ nextCharacter : java::util::List
+ GoToFunction()
+ getNextState(letter : char) : State
Parser
+ Parse(new_parameter : data::Document) : TemporaryIndex
<create>
TemporaryIndex
+ documentId : data::Document
+ keywordIdsList : java::util::List<Keyword>
+ TemporaryIndex(DocumentId : int)
+ updateIndex(keywordId : int)
+documentId
data::Document
+ ID : int
+ BlogContent : string
+ Author : string
+ Title : string
+ Link : string
+ getID() : int
+ getContent() : string
+ getAuthor() : string
+ getTitle() : string
+ Document() : Document
*
data::Keyword
+ canon : string
+ synonyms : java::util::List
+ ID : int
+ getCanon() : string
+ getSynonyms() : java::util::List
+ Keyword() : Keyword

Figure 13: State machine class diagram

5.2.2 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 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`.

5.2.3 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

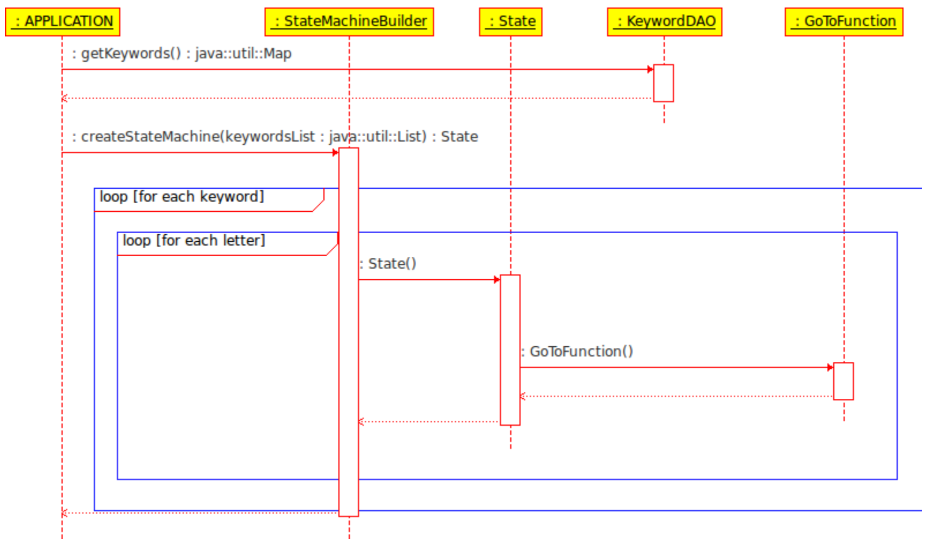

Figure 14: State machine sequence diagram

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`.

6 Experiment Results

The whole idea of the project belongs in being able to reduce the time to build the index. As our system is building the inverted index by parsing all the post into the state machine, we can expect that the complexity will be proportional to the whole number of characters of all the post we want to build the index for. As a result changing the number of Posts, the number of characters inside of each post should have a big impact in our time complexity. Besides, we should also have a time needed at the very beginning of the running to build the state machine. We can expect that the building of the state machine needs a time proportional to the sum of the size of the keywords.

The system proposed in this paper doesnt take care of various compression techniques for storing the index. To do that, many techniques have been explored in the literature. We will just highlight the different sizes of our indexes according to different parameters.

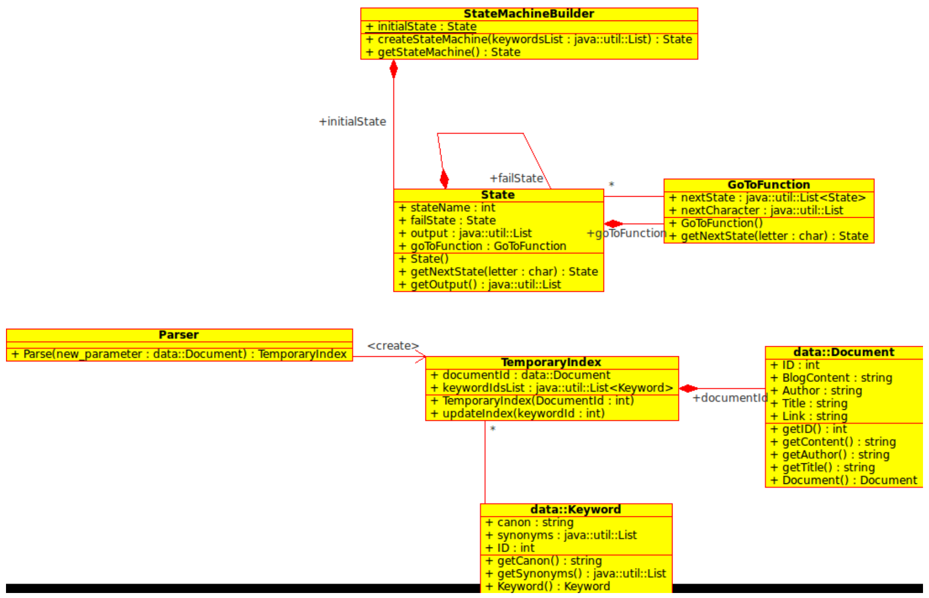
StateMachineBuilder
+ initialState : State
+ createStateMachine(keywordsList : java::util::List) : State
+ getStateMachine() : State
+initialState
+failState
*
State
+ stateName : int
+ failState : State
+ output : java::util::List
+ goToFunction : GoToFunction
+ State()
+ getNextState(letter : char) : State
+ getOutput() : java::util::List
+goToFunction
GoToFunction
+ nextState : java::util::List<State>
+ nextCharacter : java::util::List
+ GoToFunction()
+ getNextState(letter : char) : State
Parser
+ Parse(new_parameter : data::Document) : TemporaryIndex
<create>
TemporaryIndex
+ documentId : data::Document
+ keywordIdsList : java::util::List<Keyword>
+ TemporaryIndex(DocumentId : int)
+ updateIndex(keywordId : int)
+documentId
*
data::Document
+ ID : int
+ BlogContent : string
+ Author : string
+ Title : string
+ Link : string
+ getID() : int
+ getContent() : string
+ getAuthor() : string
+ getTitle() : string
+ Document() : Document
data::Keyword
+ canon : string
+ synonyms : java::util::List
+ ID : int
+ getCanon() : string
+ getSynonyms() : java::util::List
+ Keyword() : Keyword

Figure 15: Parser class diagram

Finally we will do a time comparison between our way of building the inverted index and the naive way. All the time complexities tests are done on a single computer and it is important to understand that all these processes can be parallelized. A future work would be to install the system on multiple computers and explore the gain in time by running the code simultaneously on these computers.

All tests were run on a single Intel Core 2 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. The database used was MySQL x.x, running on ....

6.1 Number of Keywords

To be able to study the impact of different number of keywords, we built a keywords database with 10.000 most popular English words. These 10.000 words come from: http://www.mit.edu/ ecprice/wordlist.10000 All the experiments in this part are done with 50.000 posts randomly downloaded from internet. The idea is to get some heterogeneous length and type of post such that it is realistic. The experiment shown here consist on building the index for different number of keywords from the 10.000 keywords database and explore the time and space

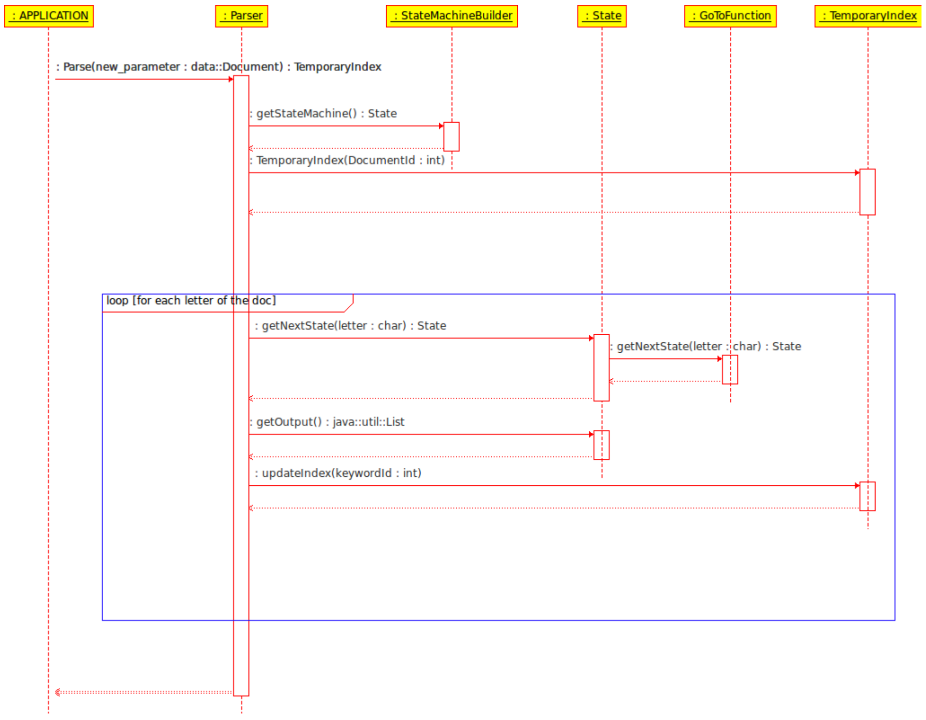
: APPLICATION
: Parser
: StateMachineBuilder
: State
: GoToFunction
: TemporaryIndex
: Parse(new_parameter : data::Document) : TemporaryIndex
: getStateMachine() : State
: TemporaryIndex(DocumentId : int)
loop [for each letter of the doc]
: getNextState(letter : char) : State
: getNextState(letter : char) : State
: getOutput() : java::util::List
: updateIndex(keywordId : int)

Figure 16: Parser sequence diagram

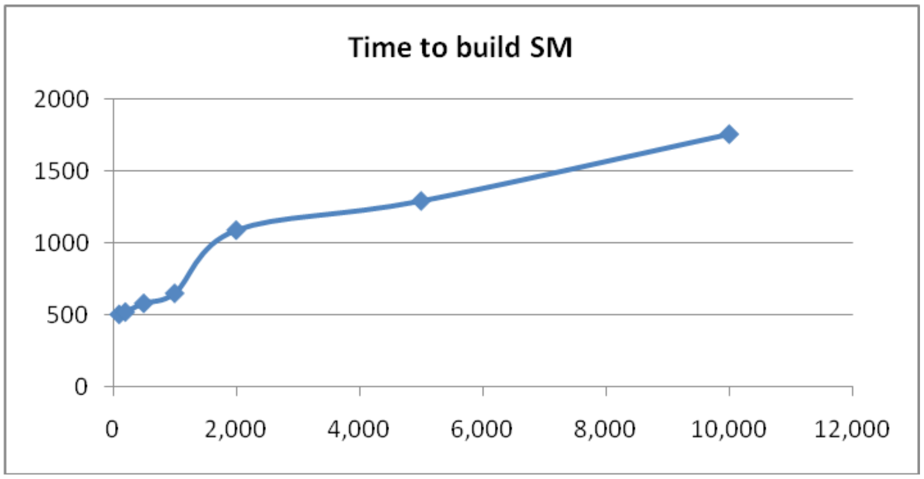

Figure 17: Time to build Aho-Corasick state machine as a function of the number of keywords. Time taken is shown in miliseconds on the y-axis and the number of keywords on the x-axis

complexity for the building of the state machine and the building of the index once the state machine is built.

6.1.1 Time Complexity

- **State Machine Construction:** 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 17 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.

- **Index Construction:** 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 18. 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 18, the time to build the index increases

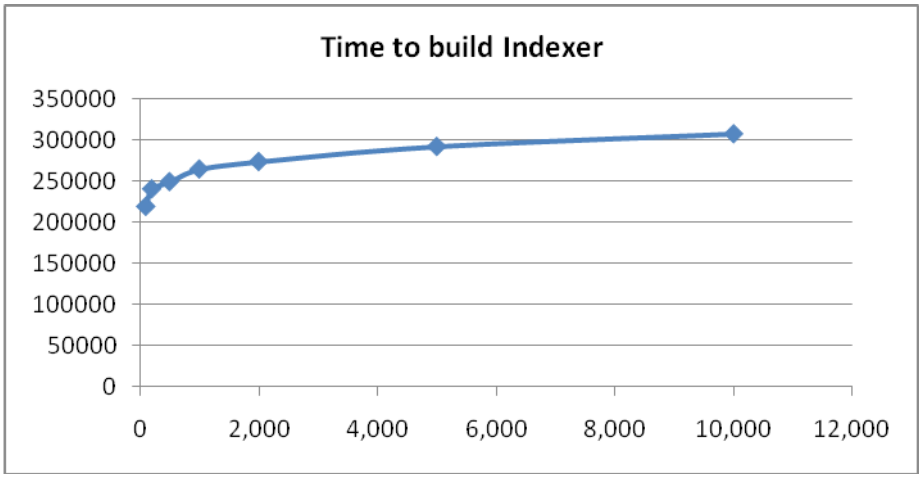

Figure 18: Time to build the inverted index as a function of the number of keywords. Time taken is shown in miliseconds on the y-axis and the number of keywords on the x-axis

logarithmically with the number of keywords. This is likely due to the fac that as the number of keywords increases, so does the size of the index and the number of database interacts required to store and update the index.

6.1.2 Space Complexity

As we would have expected it, the space used by the index in the database is proportional to the number of words we are using. It also means that our words have the same probability at the beginning or at the end of our 12,000 words dictionnary. This result is reflected in figure 19.

6.2 Size of Corpus

To be able to study the impact of different number of Posts, we built a posts database with 400.000 posts randomly downloaded from internet. Then the keywords used to build the index are simply the first 5.000 from the previous keywords database.

The experiment shown here consist on building the index for different number of Posts and explore the time and space complexity for the building of the state machine and the building of the index once the state machine is built.

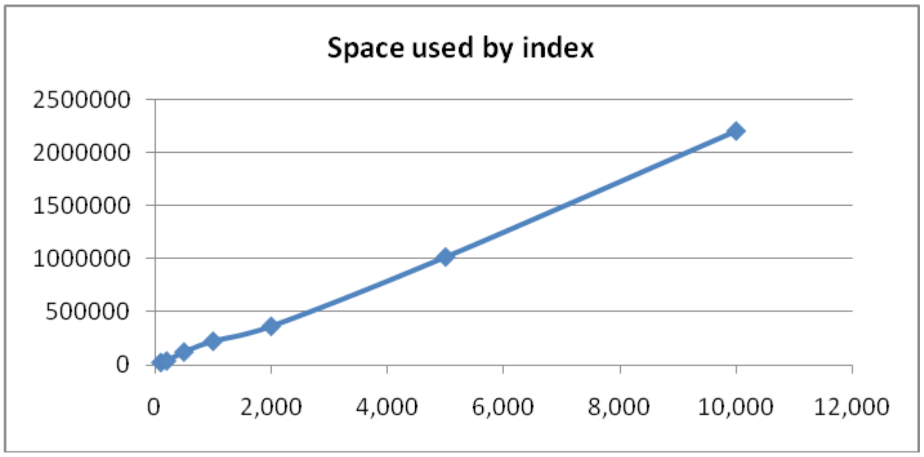

Figure 19: Size of index as a function of the number of keywords. The number of database rows used to store the index is shown on the y-axis and the number of keywords on the x-axis

We can expect that the time to build the State machine wont change with the number of posts and the time to build the index will increase almost linearly with the number of posts.

6.2.1 Time Complexity

As we were expecting, the time to build the index with the state machine strategy is linear with respect to the number of post, as depicted in figure 20. However, to be able to perform this experiment we had to run the indexer program with batches of 10,000 posts due to Java heap space problems. As a result, the state machine is rebuilt every 10,000 posts.

6.2.2 Space Complexity

As expected, the space used to sptore the index in the database is proportional to the size of the corpus. This result is shown in figure 21.

6.3 Document Size

As we have seen before, the time to build the index is almost proportional with the number of post. Theoretically this time should be proportional to the total length of all posts. Lets focus on this aspect and study the impact of the length

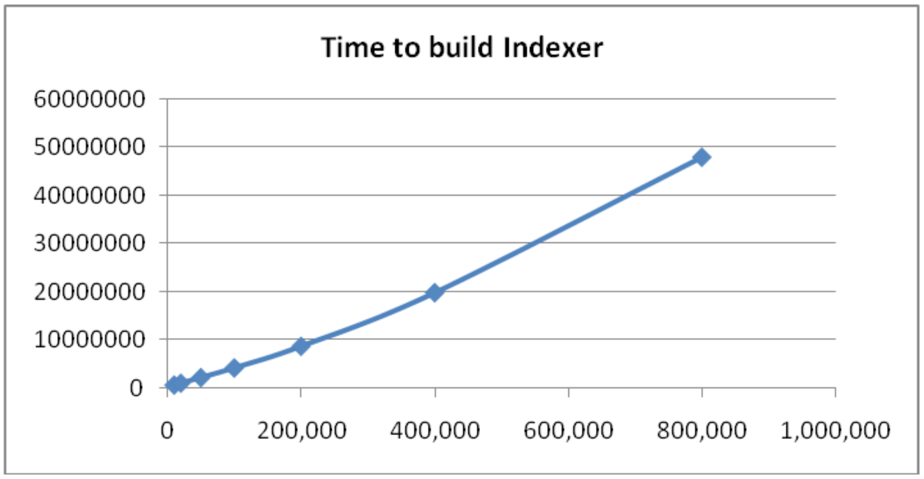

Figure 20: Time to build the inverted index as a function of the size of the corpus. Time taken is shown in miliseconds on the y-axis and the number of documents in the collection on the x-axis

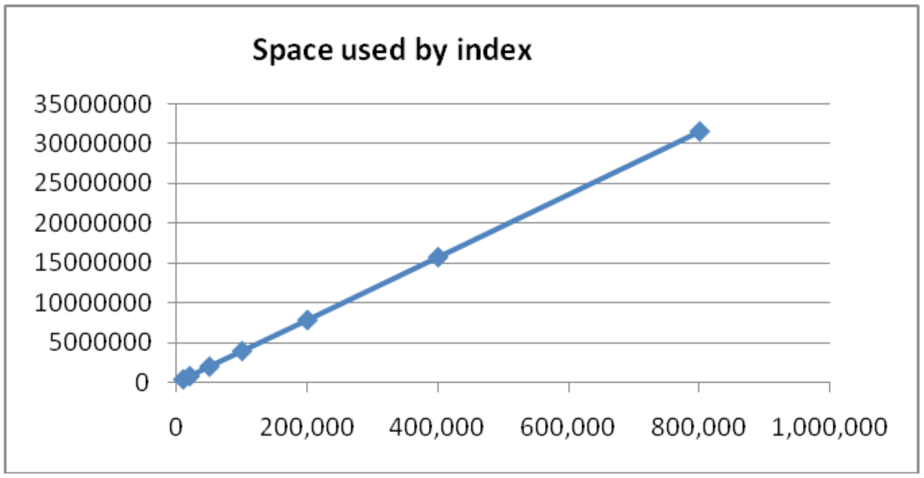

Figure 21: Size of index as a function of the size of the corpus. The number of database rows used to store the index is shown on the y-axis and the number of documents in the collection on the x-axis

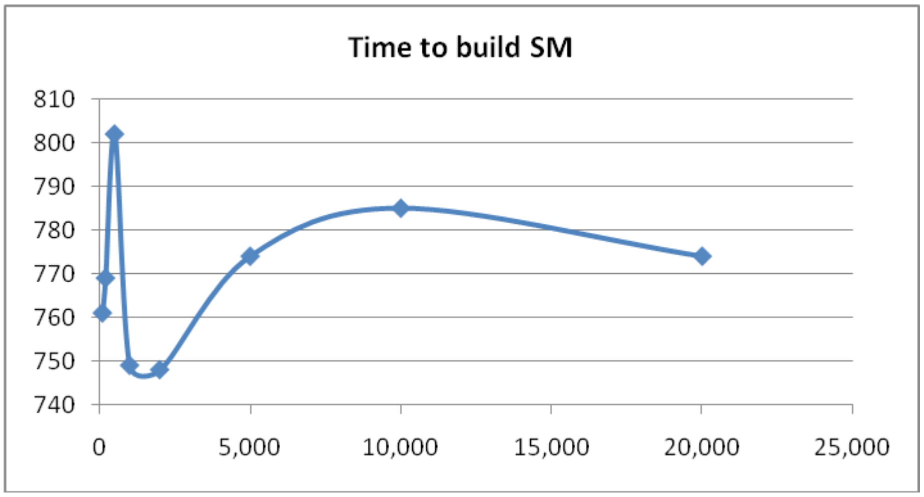

Figure 22: Time to build Aho-Corasick state machine as a function of the average document size. Time taken is shown in miliseconds on the y-axis and the average document size in characters on the x-axis

of the post on the time to build the state machine. To do that we built a 5.000 posts database with posts that are all composed of 20.000 characters. Then, we ran our system on different length of substring of these posts.

6.3.1 Time Complexity

- **State Machine Construction:** As expected, state machine construction time does not depend on the length of the documents. Nonetheless it is interesting to compare figures 22 and 23, showing the construction times for the state machine and inverted index relative to the average document size. The comparison shows that state machine construction time is several orders of magnitude smaller than index construction and negligible by comnparison. Our state machine is built in 770 ms on average for 1000 keywords with a standard deviation of 18ms.

- **Index Construction:** As expected the time to build the indexer is almost linear with respect to the average document size. This relationship is shown in figure 23.

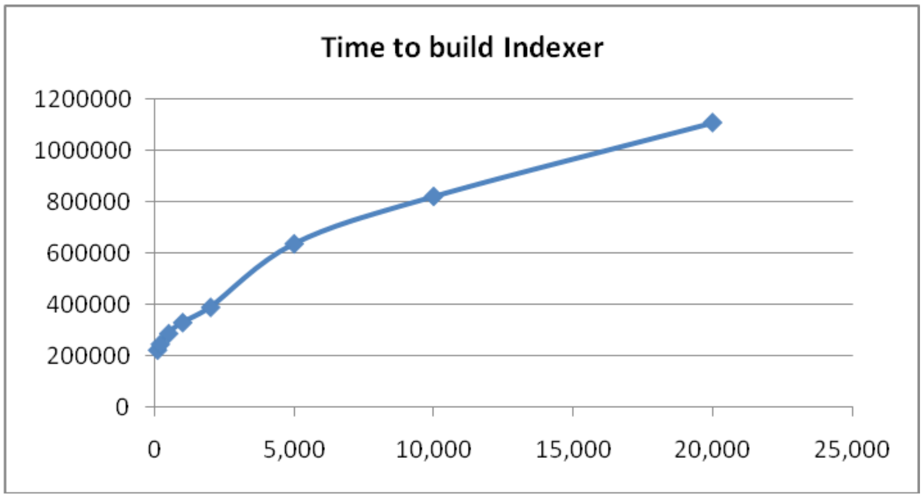

Figure 23: Time to build an inverted index as a function of the average document size. Time taken is shown in miliseconds on the y-axis and the average document size in characters on the x-axis

6.3.2 Space Complexity

Here we are studying the number of records in our index database as a function of the average document size. As expected, the larger the documents the more records created to represent the index. However, this is not linear since if a word occurs many times in the same post we just save it once in the database. This relationship is shown in figure 24.

6.4 Comparison with Naïve Index Builder

This part focuses on the improvement that our system with respect to time complexity in comparison with a naïe index builder. 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.

As we can see this algorithm has a complexity that directly depends on the number of keywords n, the number of document m, and the length of document since we have to check in each document for each keyword. Moreover, we decided to limit, as much as possible, the accesses to the database since it is really time consuming in Java. Thats why we do one access to the database for

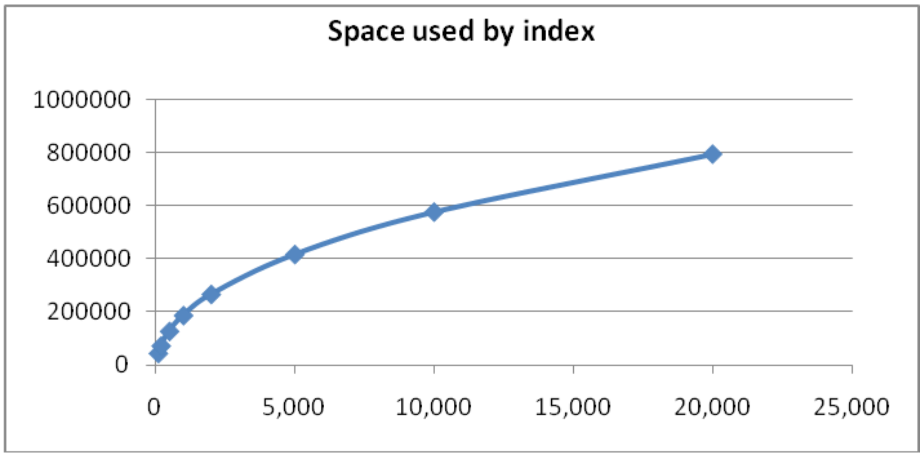

Figure 24: Size of index as a function of the average document size. The number of database rows used to store the index is shown on the y-axis and the average document size in characters on the x-axis

Algorithm 1 Build inverted index

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

Figure 25: Time to build an inverted index as a function of the average document size. Time taken is shown in miliseconds on the y-axis and the average document size in characters on the x-axis

each keywords and not for each document. In the state machine we do as much accesses to the database as the number of document (in the worst case meaning we find at least one keyword per document). In the following subsections we compare the results for this algorithm with our system results for different number of keywords and posts.

6.4.1 Complexity Analysis

For the naïve algorithm described above, the theoretical worst case time complexity relationship is shown in equation 1.

$$Time \propto keyNum * (docNum * maxLength + conn) \tag{1}$$

where $keyNum$ is the number of keywords, $docNum$ is the number of documents, $maxLength$ is the maximum document length, and $conn$ is the time taken to create a connection to the database. For the state machine algorithm the theoretical complexity relationship is given by equation 2.

$$Time \propto docNum * (maxLength + conn) \tag{2}$$

6.4.2 Document Size

For this experiment we fixed the number of keywords to 1,000 and the number of documents to 5,000 and we ran our code on a database composed of document of different length. The experiment results are shown in figure 25.

As expected, the curves for both our indexer and the naive indexer are almost linear since for the naive implementation:

$$Time \propto (keyNum * docNum) * maxLength + conn * keyNum$$

and for our Aho-Corasick state machine method:

$$Time \propto docNum * maxLength + conn * docNum$$

Figure 26: Time to build an inverted index as a function of the average document size. Time taken is shown in miliseconds on the y-axis and the average document size in characters on the logarithmic x-axis

Figure 27: Time to build an inverted index as a function of the number of keywords. Time taken is shown in miliseconds on the y-axis and the number of keywords on the x-axis

We can observe that for a small length of document naive method is faster since $keyNum$ is smaller than $docNum$. For $docLength = 1300$ our indexer is far more efficient. To see this effect more clearly, experiment results are plotted on a logarithmic scale in figure 26.

6.4.3 Number of Keywords

As expected, the two implementations exhibit a linear response to the number of keywords used. In the naïve implmementation:

$$Time \propto (keyNum * maxLength) * docNum + conn * keyNum$$

and in our Aho-Corasick state machine method:

$$Time \propto (maxLength + conn) * docNum$$

However, we would have expected the state machine method not to depend on the number of keyword. The reason why its not the case is because when we dont have any keywords in a document, we dont access the database. Knowing that the database access is very time consuming, increasing the number of keyword increases the probability of finding a keyword into each document and so increase the time complexity.

The experiment results are shown in figure 25 and on a logarithmic scale in figure 26.

6.4.4 Size of Corpus

As expected, the time complexity is linearly related to the number of documents being indexed. The experiment results are shown in figure 29 and on a

Figure 28: Time to build an inverted index as a function of the number of keywords. Time taken is shown in miliseconds on the y-axis and the number of keywords on the logarithmic x-axis

Figure 29: Time to build an inverted index as a function of the number of documents being indexed. Time taken is shown in miliseconds on the y-axis and the number of documents on the x-axis

logarithmic scale in figure 30. For the naïve implementation:

$$Time \propto (docNum * maxLenght + conn) * keyNum$$

and in our Aho-Corasick state machine method:

$$Time \propto docNum * maxLenght + conn * docNum$$

7 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. Due to time constraints, this was not done but the resulting implementation lays a strong

Figure 30: Time to build an inverted index as a function of the number of documents being indexed. Time taken is shown in miliseconds on the y-axis and the number of documents on the logarithmic x-axis

foundation for further research in this area. 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.
- 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] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern recognition.*, vol. 30, no. 7, p. 1145, 1997.

[4] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[5] R. Gupta, H. Gupta, U. Nambiar, and M. Mohania, "Efficiently querying archived data using Hadoop," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, ser. CIKM '10. New York, NY, USA: ACM, 2010, pp. 1301–1304.

[6] T. G. Kolda and D. P. O'Leary, "A semidiscrete matrix decomposition for latent semantic indexing information retrieval," *ACM Trans.Inf.Syst.*, vol. 16, no. 4, pp. 322–346, October 1998.

[7] T. A. Letsche and M. W. Berry, "Large-scale information retrieval with latent semantic indexing," *Information Sciences*, vol. 100, no. 1-4, pp. 105–137, 8 1997.

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

[9] M. Rosell, "Introduction to Information Retrieval and Text Clustering," " 2006.

[10] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

[11] H. Zha and H. D. Simon, "On updating problems in latent semantic indexing," *SIAM Journal on Scientific Computing*, vol. 21, no. 2, pp. 782–791, 1999.