HOW DOES A MAP-REDUCE ALGORITHM BEHAVE IN AN INFORMATION-RETRIEVAL ENVIRONMENT

Author:
Vincent Gijsen

Supervisor:
prof.dr.ir. Th.P. van der Weide

date:
July 6, 2012
Abstract

In this thesis we are examining how we can incorporate the map-reduce paradigm in an information-retrieval environment. By combining map-reduce with standard techniques used in this field, we are constructing a search-engine. Unlike most search engines, we incorporated a natural-language-parser into the map-reduce framework to distribute calculations. By combining these components with a column-oriented database we try to supply the user with means to search easily.
## Contents

1 Introduction .................................................. 1  

2 Meaning .......................................................... 2  
   2.1 What is the meaning ........................................ 2  
   2.2 Representation of meaning ................................. 2  
   2.3 Our definition of meaning ................................. 4  

3 Extraction of meaning ......................................... 5  
   3.1 Introduction ............................................... 5  
   3.2 Overview methods ......................................... 5  
   3.2.1 Full parser ............................................ 5  
   3.2.2 Shallow parser ....................................... 6  
   3.2.3 The tagger ............................................. 6  
   3.2.4 Spelling and misspelling .............................. 6  
   3.3 Our choice of extraction method ......................... 7  
   3.3.1 Motivation ............................................ 7  
   3.3.2 The Stanford Parser ................................ 7  
   3.4 Storing considerations ................................... 7  
   3.4.1 Requirements for storing ............................. 7  
   3.4.2 Flattening the hierarchical structure ............... 8  
   3.4.3 The data store ....................................... 8  
   3.4.4 Data enrichment ..................................... 10  

4 Question .......................................................... 16  
   4.1 What does a query look like ............................. 16  
   4.1.1 Keyword based ........................................ 16  
   4.1.2 Guided dialogue e.g. form ........................... 16  
   4.1.3 Free text .............................................. 16  
   4.2 Our choice of query formulation ......................... 17  
   4.3 Our definition of the meaning of a query ............. 17  

5 Matching .......................................................... 18  
   5.1 User requirements for matching ......................... 18  
   5.2 Components required to match ........................... 19  
   5.2.1 How to match ......................................... 19  
   5.2.2 Calculating and weights ............................... 19  
   5.2.3 Analysis on matching ................................ 21  
   5.2.4 High level description of software .................. 22  
   5.2.5 The approach ......................................... 24  

6 Distributed Computing .......................................... 25  
   6.1 What is distributed computing ........................... 25  
   6.2 Distributed in relevance with this research .......... 26  
   6.2.1 Large datasets ....................................... 26  
   6.2.2 Expensive parsing .................................... 26  
   6.2.3 Time constraints ..................................... 26
6.3 Distributed computing = cloud computing

7 The system we built
7.1 Overview
7.1.1 Black-box
7.1.2 Indexing
7.1.3 Our index
7.1.4 Processing queries
7.2 Details
7.2.1 Research lab
7.2.2 Infrastructure
7.2.3 Distributed file system
7.2.4 Software
7.2.5 Analysis of parser
7.3 Result
7.4 The implementation
7.4.1 The index generating
7.4.2 Performance tuning

8 Experiments and results
8.1 Design of the experiment
8.1.1 Input data
8.2 Benchmark
8.2.1 Indexing Wikipedia data
8.2.2 Costs
8.2.3 Room for optimization?
8.3 Distribution of costs
8.3.1 Indexing
8.3.2 Matching
8.4 Conclusion

9 Acknowledgements

10 appendix
10.1 Data import
10.2 Distributed shell-scripts
10.3 Parser
10.4 Indexer
10.5 Searcher
10.6 Performance tuning
10.6.1 Hadoop
10.6.2 HBase
Chapter 1

Introduction

This research is about search queries. When one utilizes a search engine, one usually searches by statistical means. With this research we are looking at the means to search by meaning rather than plain statistical methods. The approach of this research will be the experimental type.

We are going to investigate how search-queries which do not use strictly keywords (eg ‘cheap car’) can be analyzed for ‘better’ results. Questions which one could pose someone on the street, questions like Where can I find a good book which is not too expensive?. When this query is executed against a search engine like Google, it will yield results like flight-tickets, rental prices for cars and books about poker. This is due the seemingly statistical approach. A ‘real’ person would recognize that the word book, is used as a verb or predicate, rather than a noun. If a search engine would dissect input queries in a similar fashion, it could possibly return more favorable results. One of the self-applied constraints is that we wanted to implement this experiment using No-SQL technology, in our case Hadoop in combination with HBase.
Chapter 2

Meaning

The most well-known structure to represent knowledge obviously is natural language. Many authors assume that the structure of language reassembles the way our brains handle knowledge. However, automatically processing of natural language is not trivial. Therefore search systems use a less ambitious level, and ask the searcher for example to only enter relevant keywords. In this section we are interested to find a representation in between, covering semantics in a better way and yet efficiently to process.

In this thesis we focus on the key components that form sentences, being:
- the predicate of the sentences;
- the noun groups and their role in relation to the predicate

Adjectives lead to a further convergence towards the intended meaning of the noun in the sentence.

2.1 What is the meaning

What is the meaning of a search query? This question can easily lead to very philosophically discussions which are not the kind of topics this research is about. Queries may not be very well-formed. A query may mention the main issues of the information need, or the result of an interview session.

A document is constructed of sentences. A document may consist of sub-documents (chapters), chapters may consist of sections, etc. At the basis of this structure we find the smallest structural units composing the document. We will refer to these smallest structures as paragraphs. The characterization of a paragraph is formed by the characterization of its sentences.

But how would we extract the meaning of a query like Where is the nearest quality car dealer? Obviously the searcher is interested in a dealer. But the sentence carries several adjectives that give a lot of context to this noun. The adjectives do not carry meaning on their own. However all together they give more semantics to the word dealer. For humans its very easy to interpret the sentence and to recognize the proper way to interpret its proper context, something that is remarkable difficult to achieve using computer programs. However Natural Language embodies lots of ambiguities in its very own existence. The only way for humans to proper interpret it is to know its context, which is usually deducted out of surrounding information supplied by other sentences or annotations. But there are still exceptions where we too, fail in recognizing the proper meaning.

2.2 Representation of meaning

From language theory it is known that the predicate is considered to be as the dominant information in a sentence. So for example, the sentence He gave her cat food, definitely is about give.

As a first step, we omit some elements of the sentences, for example particles. We will use the mechanism of index expressions to represent the resulting structure. An index expression consists of
- 1. a header term representing the predicate of the sentence,
- 2. followed by a series of modifiers, consisting of the combination of role and involved object.

For example, the structure of the sentence

\[ \text{person Smith visits country Italy} \]

is displayed in Figure 2.1 and will be denoted as the following index expression:
visit **agens** (person **being** Smith) **patiens** (country **being** Italy)

Special roles are **agens** (most commonly associated with the grammatical subject) and **patiens** (most commonly associated with the grammatical object). Other objects will have a particle that clarifies their role. This example demonstrates the structure of index expressions. A grammatical description of index expressions is: We will not go into the details for natural language processing, for example the handling of determiners. Consider the sentences:

1. This guy gives a female cat food?
2. This guy gives the woman her cat, food

Out of context, it is not possible to deduct what the author meant by the determiner *this*. Usually humans will be able to deduct this from surrounding text, but do sometimes require clarification. If the parser is able to determine the entity’s name we will include it in the index, otherwise it will be rejected.

1. Jan paints inside his house.
2. Jan walks past a house.

In both queries, the noun is “house”, however in the first sentence the noun “house” has significant meaning in its context. In contrast to the second sentence, the word house is of little meaning referring just toward a place, the actual meaning full word is “walking”. The index expression representations are:

1. paint **agens** Jan inside (house **being** his)
2. walk **agens** Jan past house
There has been a lot of research on this particular topic already, therefore we are going to use tools that do the grammar analysis up-to a certain degree of confidence, these results from tools will be used during the research.

2.3 Our definition of meaning

We like to see the concept ‘meaning’ as the following:

Figure 2.3: Graphical display of document to sentence to ‘meaning’

We slice each sentence in the document based on the punctuation. This leaves us a collection of sentences which represent a document. These sentences are processed by a parser which generates output. This output is then our ‘meaning’. By doing so we make a deliberate choice to strip out some semantics of the original document. Paragraphs and headings for example, are removed this way. We keep the sentence position in the original document, therefore leaving the possibility open to utilize this positional property.
Chapter 3

Extraction of meaning

3.1 Introduction

In order to analyze natural language we need to analyze text by automating means. The process of analyzing text and extracting some kind of information out of it is called parsing. Although researchers in the field of linguistics spend the last thirty years -as of this writing- on coping with this problem they still have not exceeded in doing this process with 100% accuracy. They most accurate parsers achieve a overall 90% correct analysis of the input. They will most likely never succeed in reaching full accuracy due to the ambiguity embodied in language itself. Ask one self, ‘why is it sometimes impossible to answer to a question without knowing it context?’ This is exactly why it is so difficult to parse text. For a program is nearly impossible to grasp concepts like ‘context’ and ‘underlying-meaning’ for example, when someone is cynical. Other problems which arise when parsing text is the usage of punctuation in sentences. A sentence starts with a capital and ending with a period, but usage of brackets, dashes or commas may not stand in syntactical relation with the surrounding material. Furthermore, incorrect spelling can lead to false assumptions.

Natural language parsing is used in many categories of research. Think of translators (Google translate), in discourse analysis and in conversion from data to human-readable text (e.g. a chat system where one asks the opposite -usually a computer program or (ro)bot) about certain topics, for example: IKEA uses such system, to let customers inquire for information.

3.2 Overview methods

We need to extract this tree form a sentence, which we just introduced. How can we go from sentence towards tree (fig 2.1)? Generally speaking there are two kind of parsers used to dissect text. There is the full-parses opposed to the shallow-parser. As the name may vaguely imply, the full parsers is more thorough where is the shallow parser more speedy.

In order to identify elements in text we need to define how these elements are crafted. By splitting a text or document into multiple pieces we can compare these elements with the elements form a query. But how should we define where to split them? We will split text to pieces based on the dot or final (.). Every single remaining sentence that is defined due to this split will be fed to the parser of choosing. This parser then extracts the ‘meaning’ of it’s input.

3.2.1 Full parser

Workings A Full parser (also know as probabilistic parser) uses hand-parsed (read human) grammar as a reference to map input onto this grammar. By probability calculations this kind of parser computes all mutations of a sentence to map as accurately onto the grammar as possible. The parser tries to build a tree, if it succeeds it’s rates this variant with a high score, if it fails, it marks this tree with a lower score. By trying many combinations it achieves a good result. However the more grammatical/spelling errors are present in a sentence, the more expensive (in terms of processing time spend and memory consumed)
it gets. Depending on the errors in the sentence the lesser sure the parser is on the correct outcome. Another problem of this kind of parser is sentences that contain ambiguities.

Pro’s
• Best possible results.

Con’s
• Expensive to calculate, and large amounts of memory required to process a sentence, see [2] & [1].

3.2.2 Shallow parser

Shallow parsers also known as Chunk parsers are quite fast due to their more simple way of working. Basically they slice pieces of text and try to match parts of them based on pre-determinant rules like position in the sentence [3]. This makes them less sensitive for misspelled words but also more prone to make mistakes.

Pro’s
• very fast.

Con’s
• more prudent to make mistakes.

Workings
Shallow parsers exists in various implementation variants. There exists no ‘one’ technique that defines how these are technically designed, however they all share a common property: their interpretation of the input need not be complete. In other words, they can output unidentified structures. These parsers work on a basic linguistic knowledge. Often a shallow parsers consists of several components, a morphological analyzer and often a tagger to eliminate dis-ambiguities [4].

3.2.3 The tagger

The tagger comes in various versions, consisting of rules to tag words. In addition, they utilize a small lexicon to compensate for exceptions to the rules. Later on, they started incorporating ‘hidden Markov models’. These HMM parsers have become the standard according to [5]. I can recommend reading those papers for more insights. Most parsers utilize a tagger in order to do the initial classification of words.

3.2.4 Spelling and misspelling

During this research we neglect misspelled words. The parser should be able to cope with misspelled words. However, if the outcome of the parser is wrong due to misspelling, we accept this. Another argument to tolerate misspelled words is that language is a dynamic thing. New words are added over time, as spelling rules tend to change. In the Netherlands at one time, there was the ‘green book’ which stated the new spelling rules. Additionally loanwords may or may not be properly recognized.

Stemming is a technique used to deduct the root or stem - of words so that different conjugations will be transformed to the stem of that word. As mentioned in the functional design. We match trees. If we do not stem the parser’s output, we could end up with the following examples. Although they are very similar, they will co-exist next to each other in the index, which is undesirable:

• (bike, blue)
• (bikes, blue)

In addition, if we were to utilize predicates in the index, this will occur without stemming:

• (Paint, house)
• (paint, House)
• (paint, house)
• (paints, house)
from a conceptual point of view, its all the same activity on the same direct object. Our stemmer (the Stanford parser’s Morphology Class) is able to stem noun plurals, pronoun case, and verb endings, and not things like comparative adjectives or derived nominal.\[6\]

3.3 Our choice of extraction method

3.3.1 Motivation

Our main concern is ease of implementation (as the focus of this research is not parsing) we’ll choose whichever parser suits this requirement best. In order to speed-up implementation the first priority is obviously that the parser should incorporate within the same programming language. As Java is our language of choice, the parser we use should preferably be a library which is written in the same language to ease implementation. A second choosing of parser is an output which is tree-based, as we like to relate to sentence structures as trees, this would also be considered a pro. Although full-parsing requires a lot of resources, doing this exercise in parallel utilizing the Hadoop framework regains a lot of time due to parallelization.

3.3.2 The Stanford Parser

The Stanford parser provides us with all the necessary tools we need for this experiment. It is Java based, so easy integration with the software. Aside, it provides us with full-parsing capabilities, as well as an output tree, which we should be able to map at a tree that fits in our database of choice. As it is a full-parsers, it should deliver high-quality output (at the cost of speed). As a bonus, it also offers stemming (more on stemming further down this document).

3.4 Storing considerations

In order to ‘store’ meaning (which we agreed on is tree-based) of a document. We need to choose a way of storing this in a format which gives us speed and efficiency. We need to somehow reach a consensus between these two parameters.

3.4.1 Requirements for storing

In order to store the data, we have constructed several points that should be considered in order to make a usable system. Between these points we should find a consensus.

- speed of inserting
- speed of retrieval
- costs of storing

\textbf{Speed of inserting} Speed is important. We should not need waiting long for just a single record to be fetched or stored. In order to cope with this requirement, we need to utilize a clever way to store records so that they can be fetched fast. In order to get speed, a distributed file-system is not the proper way to go, as it is designed for batch operations (e.g., read large files sequentially). Therefore we need a database which is fast for retrieving data, but also incorporates scalability for large data-sets.

\textbf{Speed of retrieval} Speed of insertion being important, speed of retrieval is even more important. Searching for data should be fast, no one likes to wait for a long time to retrieve their info.
Costs of storing  Storage (these days) is quite cheap. In order to increase speed, one could consider sacrificing more data to increase speed. Redundancy is often considered bad-practice in a relational database. However in large databases, one has to consider increasing redundancy in favor of speed. However if one overdoes this, it can potentially slow down a system. This is where Compression can increase throughput even more. One can apply compression to reduce size, this saves disk-space, but costs more CPU. As disk read/write speed will stay the same, but the amount of data represented per volume-unit is larger, rendering compression a real option. Compression can be implemented in two ways. One can apply compression in their application by compressing blobs and storing those to the database. Another option is to let the Database Management system apply compression at run-time, making it transparent to the database-client. A third option is to represent certain data which short-cuts. Common data could be represented by integers, making the database more efficient. Often database engines gain better speed at integer operations than string based ones. It is the designer’s choice to utilize these methods in their own vision as well as combining them. We have chosen to use compression in HBase, for storing documents, but not using compressions during the map-reduce iterations. Based on [7] we have come to this decision. We read relatively small amount of data (although having the type of data -text-compresses is very well ), and there is almost no map to reduce output.

The actual storage of meaning  We need to store the meaning, we could opt to discard some parts of the sentence in favour of speed or ease. However we like to keep as much information about the sentence as possible. Therefore we will store every grammatical entity in the database and it’s inter-relationship with the other entities. To do so, our approach is to flatten a tree so that we can store the relationships among entities, and rebuild when deemed necessary. As mentioned previously, the only part we discard is the layout of the text, all sentences from the text are kept, but meta data like paragraphs and style are discarded.

3.4.2 Flattening the hierarchical structure

In order to meet the above requirements, we have flattened the tree structure to individual nodes, this makes storing them more easy.

Utilizing these tree structures and ‘walking’ from node to node (as the numbers indicate) gives us the possibilities to recover the tree from flatten form. The numbering strategy is derived from [8]. Utilizing this way of identifying nodes, we are able to recover structure of sentences and inter-relationships between nodes whenever necessary.

![Figure 3.1: Parse-tree with node numbering](image)

3.4.3 The data store

To RDBMS or not to RDBMS  Many believe a relational database is the preferred way to store data. And in many cases this will be true. However when one has to process large quantities of data at reasonable speed. One big expensive super-computer won’t do the job. Overtime there have been solutions to scale relational databases using techniques like sharding. But this requires some trickery like load balancers and other artifacts. Another constraint considers the relations which occur on one record that relate to another table should ideally be stored on the same machine. Storage of a graph is not very difficult in a relational database. However if this graph starts to outgrow one physical machine, things get harder to maintain. This is where databases, designed with an distributed application in mind can offer outcome, these databases are often used in the context with ‘Big-Data’. HBase is such application. However don’t perceive HBase or big-data platforms as silver-bullet. In a lot of context’s a relational database is still the preferred method of data storage/retrieval.
**HBase** is a database-application which (usually) runs on-top of the Hadoop file-system. HBase facilitates *random, real-time read/write access to the data*, where Hadoop is meant for large sequential reads. HBase is a column-oriented database, designed to store vast amounts of data and make this data real-time accessible.

HBase stores its keys lexicographically, this means the keys are bit-wise stored on. A designer is therefore challenged to choose his keys in a way he can quickly retrieve data that is associated with those keys. Another approach is to scan all the data, but this is considered very bad practice. Because of HBase its internal structure, it is very fast in retrieving a certain key. By iterating of a range of keys, we can retrieve data very fast. In essence this key structure in HBase is similar to an index used by certain RDBMS-es like MySQL. The fact that HBase is a column-oriented database has some other benefits opposed to traditional relational databases. HBase has the concepts of Tables and Column-families. A column-family is a conceptual group. In this group one is allowed to create an arbitrary amount of 'sub-columns' for example: if the column-family could be *characteristics* in the table *members* one could create *characteristics:name*, *characteristics:street* and *characteristics:house*. HBase stores these columns which share the same column-family together on disk, for easy retrieval. One is advised to limit the number of column-families to no more than three \(^9\), due to performance reasons. The user can store an arbitrary number of columns within a column-family, regardless of the other records. This feature in contrast to relational databases, makes storage of sparse-matrices very easy. Another nice property of HBase is the support for version control. Instead of overwriting a column, HBase writes a new version. If max_versions is overflowed, the oldest version will be overwritten.

The behavior of an arbitrary amount of columns within a row and version control is uncommon in relational databases. It gives us great flexibility in terms of scalability in both the width of a table as in height. HBase was designed with big-data in mind, therefore spanning a table over multiple machines is just as easy as running it on a single node.

**Table layout** In order to store our parsed data, we need to put it into HBase. Therefore we have crafted the following table layout, which displays the looks. We will utilize some of the above properties of HBase to ease storage of our data-model:

<table>
<thead>
<tr>
<th>key</th>
<th>colfam:idf</th>
<th>colfam:doc</th>
<th>colfam:tf</th>
<th>colfam:sen</th>
</tr>
</thead>
<tbody>
<tr>
<td>acomp</td>
<td>shape</td>
<td>similar</td>
<td>idf: 0.03</td>
<td>doc:1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:351</td>
</tr>
<tr>
<td>acomp</td>
<td>stress</td>
<td>due</td>
<td>idf: 0.9</td>
<td>doc:12</td>
</tr>
<tr>
<td>advcl</td>
<td>be</td>
<td>follow</td>
<td>idf: 0.01</td>
<td>doc:91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:258</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:1590</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:235</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>doc:98740</td>
</tr>
</tbody>
</table>

*the above spelling errors are intended as they are 'dumps' from an actual data-set*

Figure 3.2: HBase table layout - descriptive

The ‘key’ column is the node produced by the parser, we will store the document id within columnfamily. Due to version control build into HBase, we can updated the column columnfamily:doc with each occurrence of this element into a document. By evaluating the IDF score or the number of columnfamily:doc versions we can quickly determine the likelihood of this word in the corpus.

If we have to retrieve all grammatical components which exist in the query, we simply ask the database to return all rows with a specific key. This key will give us all columns in the ‘columnfamily’ which are the document occurrences. The value of these columns represents all positions in which the sentence occurred.
A query for row **advc|be|follow** could yield:

<table>
<thead>
<tr>
<th>key</th>
<th>columnfamily1</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
<tr>
<td>advc</td>
<td>be</td>
<td>follow</td>
</tr>
</tbody>
</table>

Figure 3.3: Results from a getRow action

Figure 3.3 (unrelated figure to fig 3.2) tells us that the leaf **advc|be|follow** occurs in document 7 at sentence id 1 and sentence id 7. The nodes in those sentences had number (2,7) and (1,2). We can derive this because every sentence out of the same document, shares the document-number as part value. Due we use the version control to have multiple documents stored under the doc descriptor. The multiple sentences which are part of a single sentence are stored again using the versioning principle.

**Weight of meaning**  In order to give an extra dimension to the meaning. We can utilize the TF-IDF algorithm (further-on explained). This algorithm assigns a score to words. By utilizing this algorithm we can discriminate between words that are frequently used and those that aren’t. If we expand this towards the relational component between words, we end up with a setting similar to the image below:

3.4.4 Data enrichment

In order to add more ‘feeling’ to the data then merely matching on overlapping keys between query and index. We will take TF-IDF into account. We will calculate for every term in the index the TF-IDF score. Till now, we looked

![Figure 3.4: Weights of the concepts](image)

**TF-IDF** is a frequently used technique to weight significance of words. For people new to this method, let us illustrate it’s generic principle with an example: If we take a text about bicycles, the word **bicycle** will most likely appear quite frequently. But the word **the** will occur quite frequently as well. However the **bicycle** will not occur likely in a text about carpets, but the word **the** will occur frequently as well. The word **bicycle** is quite characteristic for the first text but the word **the** has little meaning cause it occurs a lot, not just in one document specific but in most documents. TF-IDF is a method of grading words or terms that occur frequent, less importance over words that hardly occur. Below we illustrate the mathematics of TF-IDF [10].

$D$ be a collection of parse-trees (the sentences processed) which describe the document. Let $t \in D$ indicate that $t$ describes $D$. Than $t$ will indicate that in some sentence in document $D$ the grammatical-entity $t$ appears. This same construct applies for query $Q$.

If we apply our parse-trees to this algorithm we get:

Example:
Q = "I want a bike with proper handles".

Then

- nsubj—want—i
- root—root—want
- det—bike—a
- dobj—want—bike
- amod—handle—proper
- prep—bike—handle

are t of Q

Formal: Document descriptor D and query Q are described as

\[ D \in t \mapsto N, \]
\[ Q \in t \mapsto N \]

When we have defined how D looks like, we can define the Corpus (C). C is a collection of D’s. Therefore the corpus is defined as:

Formal:

\[ C = \{D_0, D_1, ..., D_n\} \]

<table>
<thead>
<tr>
<th>Query</th>
<th>[ Q \in t \mapsto N ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>[ D \in t \mapsto N ]</td>
</tr>
<tr>
<td>Corpus</td>
<td>{D_0, D_1, ..., D_n}</td>
</tr>
</tbody>
</table>

Figure 3.5: Definition of corpus

As a term can occur multiple times inside a (large) document, we define it as \( t \in^k D \). In other words, t can occur multiple times in D. Because of this, we like to address a document as a frequency table of t. Therefore \( t \in k \times D \) with \( k = 0 \) means t is not present in D.

By applying this algorithm, we assign each part of the parse tree a importance factor for this document.

**Normalization** is therefore required. In order to do so, we are using the de-facto standard for doing text normalization: **TF-IDF**. TF-IDF stands for **Term-Frequency - Inverse Document Frequency**.

In other words: Term frequency is about the number of counts a term occurs in the document. The Inverse Document Frequency is about the general importance of a term in the whole collection, one could say the probability a term can occur in the document. It is more easy grasp the concept of IDF with this example:

If the word ‘bike’ in the corpus is mentioned 10 times, and the word ‘the’ is mentioned 3000 times, the word ‘bike’ has significantly more meaning when used in a document than the word ‘the’. This is what IDF is about.

Formally we define IDF as:

\[
IDF(\text{parse - tree - element}) = \log \frac{|C|}{1 + |\{D : \text{parse-tree-element} \in D\}|}
\]

Where:
\[ |C| : \text{Number of documents in the Corpus}, \]
\[ 1 + |\{D : t \in D\}| : \text{Number of documents which include the parse-tree-element} \]

In order to calculate the final weight for a given term we utilize:

\[ tf-idf(\text{parse-tree-element}, d) = tf(\text{parse-tree-element}, d) \times IDF(\text{parse-tree-element}) \]

Example:
The term ‘bicycle’ occurs 7 times in a document with 100 words. Therefore TF = \( \frac{7}{100} = 0.07 \)
Our Corpus consists of 1000 documents and the term ‘bicycle’ occurs in 30 of them. Our IDF would look like log \( \frac{1000}{30} = 1.52 \).
Our final result for ‘bike’ would yield \( 0.07 \times 1.52 = 0.1064 \)

Our solution consists of a map-reduce program. This program does the heavy lifting: the parsing of the sentences. We have a diagram of this process: fig 3.6. Every mapper instance process a part of the data. A document consists of collection of sentences. We split these sentences. Every single sentences is processed by the parser. its output is writing during the map-phase into the HBase database. After parsing every sentence of a document, the mapper routine does two things:

- The mapper calculates the Term-frequency score for every term written by the parser.
- The mapper emits a ‘one’ to the context, which basically makes sure the reducer receives it with an associated key.

We need to store the TF score for every term in a document, this is easily done during the indexing phase. For the IDF score, we need to now the final number of documents. As we are processing all the documents, we utilize the map-reduce framework to count all documents we processed. This enables us to easily count the number of processed documents although we are processing thing in parallel. By defining a static ‘key’ in our mappers who emits the ‘one’ for every document processed, the reducer will receive all ‘ones’ and accumulates them to the final number. This number is than stored into HBase for later usage, the calculation of the IDF score. This process is illustrated in figure 3.6
After we’ve indexed all of the documents we would like to process for searching, there is a final step to complete before we can search for matches. We need to calculate the IDF score. As mentioned before, to calculate this score, we need to know the total number of documents. As we work with a semi-large dataset, this number isn’t always easy to come by. Therefore we counted the number of documents during indexing. The final count is also stored in HBase in a 2nd table. We have written a small program which scans over all records (the parse-tree-elements) and retrieves the number of documents it occurs in (as this is written during indexing). This enables us to calculate the IDF formula, which is as we recall:

\[
IDF(\text{record}) = \log \frac{\text{Total number of documents}}{1 + |\{D : \# \text{ documents which contain term } \in D\}|}
\]

We do this calculation for every record and end up with a tf-score per record per document and a IDF-score for every term (regardless of document). After this second stage is completed, we can search for similar items in the database. As this is a proof-of-concept we utilized a small program to iterate over all rows, written by the indexer. One could use a map-reduce for this situation as well, but as the data is relatively small after processing, it is easily done by one node, hence this approach.

**High level description** The implementation of the TF-IDF calculation is visualized in pseudo code by three pieces: 3.1, 3.2 and 3.3.

One can clearly recognize that the mapper is responsible for three things:

1. write all parsed tree components to the database
2. calculate and write the TF (term frequency) of every parse-tree-element found the in the document in the database.
3. emit for every document the number 1.

If this mapper is run on several machines in parallel, they all write towards the database, they also create a large list of 1’s. These will be accumulated and written in the reducer function. The resulting value (in the example below documents.length(), is (of course) an example. When iterating over multiple Gigabytes one cannot generally build these kinds of lists in memory, hence the map-reduce approach. We also like to use map-reduce to run these jobs concurrent, as this increases overall throughput.

```java
1 documents = [doc1, doc2, doc3, ..., docN]
2 3 for (doc in documents) {
4 sentences = doc.split(".")
5 #create empty list for the tf-calculations
6 TfScoreList = []
7 for (sentence in sentences) {
8 parseTreeElementsList = Parser.parse(sentence)
9 #write parse-tree-elements to database
10 DatabaseConnector.writeToTable("index", parseTreeElementsList)
11 #add terms to list
12 TfScoreList.add(parseTreeElementsList)
13 }
14 }
15 for (tf in tfScoreList) {
16 count = 0
17 for (tsearch in tfScoreList) {
18 if (tsearch == tf) {
19 count++;
20 }
21 }
22 }
23 #write tf-score to database
24 DatabaseConnector.writeTFScore("index", tf, (count / tfScoreList.size()))
25 end
26 toReducerFunction << 1
27 }
```

Listing 3.1: Pseudo example mapper TF-IDF calculating

The `toReducerFunction` is pseudo language for emit this ‘1’ to the reducer framework. After all mappers are processed the documents, the reducer function will kick in, receiving a list of 1’s with grouped by the same (not illustrated below) key:

```java
1 fromReducerFunction = [1,1,1,1,1,1,1] 2 int totalDocumentCounter = 0
3 for (iterator in fromReducerFunction) {
4 totalDocumentCounter++
5 }
6 #write result to database
7 DatabaseConnector.writeToTable("stats", "totalDocument", totalDocumentCounter)
```

Listing 3.2: Pseudo example reducer tf-idf calculating

To summarize: the reducer is simply receiving a list of 1’s, every 1 stands for a document. By counting them, we are effectively counting the number of documents, needed to apply the IDF algorithm (the $C$ in the example above).
We need the final # documents in order to calculate the \( IDF \) value. After the indexing is complete, there is a final step to complete, calculate the \( IDF \) value:

```java
1 function long calculateTfIdfScore(termOccurrences, totalDocuments)
2   return Math.log(totalDocuments / (1.0 + termOccurrences))
3 end
4
5 totalDocuments = DatabaseConnector.getRecord("stats","totalDocuments")
6
7 allRecords = DatabaseConnector.getAllRecords.toList()
8
9 for (record in allRecords){
10   recordKey = record.getRowName();
11   #retrieves the list of documents wich contain this term
12   recordAmountOfTerms = record.getListOfContainingDocuments.size()
13   #calculates the idf score based on total documents and term occurrences
14   idfScore = calcIdf(ammountOfTerms, totalDocs)
15   #write back the resulting score on the record
16   DatabaseConnector.writeIDFScore("index", recordKey, idfScore)
17 }
```

Listing 3.3: Pseudo example post-operations tf-idf calculating

What this pseudo-code function does is retrieve the total number of documents. With this number present, it iterates over all found parse-tree-element, requests the number of times, they were found in the corpus by all the mappers, and calculates the final score. This score is then written back on the same record (or parse-tree-element).

Further-on this writing we will elaborate on the actual Java implementation.
Chapter 4

Question

We like to use the term query when we mean the search question posed by the user. The process of constructing a query is important in order for the search to be successful. There are several ways to let a user define the topic he or she is interested in, which we will quickly describe below:

4.1 What does a query look like

In order to search for something, one has to present the user with means to enter his request. We need to define how these questions/queries are crafted. Below we pose some of the possibilities often used to supply means of interaction.

4.1.1 Keyword based

A common approach of entering search queries is by means of keywords. Searching for a particular item or topic often is done by descriptive keywords. Search engines like Google, Bing, Yahoo! and Ilse.nl all utilize this way of query-formatting. This method is often very effective, quick and repeatable in both indexing and search. However using just keywords leaves little room for granularity. As there is no structure in keywords, search-engines probably assume equally importance of the word regardless of grammatical function. Of course they could deduct of the order of words that the first word may or may not be equally important as the last, I suppose they are making a feedback loop utilizing the search results visited by the users to further improve their algorithms aside of TF-IDF.

Example: blue steer bike

4.1.2 Guided dialogue e.g. form

Another approach of specifying a query is by means of guided dialogue. One could think of a search mechanism often used in libraries. One enters the author in a designated input box, the title and year in others. This method of specifying is of great benefit for data-models where one describes an entity. Typically backed by a relation schema. The system can simply use boolean algebra to search in a highly optimized fashion. However for the less structured searches this approach is not so trivial. Especially in full-text searches one could opt for another technique.

Example:

- author: H. G. Wells
- title: The Time Machine
- year: 1895

4.1.3 Free text

Free text is basically an input box in which a user entered this query. We have not defined if the actual input consists of merely keywords or complete sentences. However we like to agree that free text consists of grammatical correct sentences.
4.2 Our choice of query formulation

During this research we focus on the free-text formulation. As we try to give a user as much freedom in terms of formulation we opt for a best effort correct spelling and grammar of the search query. A query should resemble one or more sentences. Because of this freedom, every lettered individual should be able to compose a (few) sentence(s) regarding the topic they are interested in. By parsing this input with a NLP (Natural-Language-Parser) we try to deduct the grammatical pieces the query consists of. The parsing will commence in the same fashion we parse build the index.

4.3 Our definition of the meaning of a query

We like to see the concept ‘Meaning of the query’ as the following:

```
Query
.parser

Line of text          'meaning'
```

Figure 4.1: From query to meaning

The query usually consists of one sentence, however some users may want to express their search parameters by entering more sentences. This will work as we design it to be able to.
Chapter 5

Matching

5.1 User requirements for matching

Our matching system should behave as following: We like to have a responsive system when searching for a match between documents and query. Therefore the system should respond within a few seconds. The user may be able to present alternative weights to certain grammatical components to influence the match outcome.

In order to search documents which match our query, the first simple way of matching is: search for the similar pairs. The sentence *Jonas buys paint* is split into three parts, as illustrated in the figure 5.1. For each piece we search the index if it occurs. After we have executed three corresponding queries (database search-instructions that is), we should get a list of documents which contain the parse-tree-elements. The document with the most occurrences is most likely the best match. But in order to build this conceptual representation into an working system we will explorer the necessities in the coming sections.
5.2 Components required to match

In order to search for something, we need three components,
- the index
- the query
- the match algorithm

The first two (index and query) we have already elaborated on. The ‘match algorithm’ is one we have not. In order to give results with a query, we need some way of defining which articles have relevance with the query. This ‘match’ or algorithm is composed of a few steps.

5.2.1 How to match

This is the first step. Search all documents which have one or more grammatical entities in-common with the query.

Our primary way of matching is to look for similar keys between all documents and the query. As we look at the sentence used in an earlier example it is dissected to:
- dobj_paint_buy
- nsubj_jona_buy
- root_buy_root

We could deduce which documents have a match and which have the most sentences per key. By introducing the TF-IDF score in this mix, the keys get assigned different weights and possibly influence the outcome.

5.2.2 Calculating and weights

In order to calculate scores and a final score we need to devise a weighting for each grammatical piece we discriminate. Some may receive a higher weight than other. eg. root_buy_root is far more common than nsubj_jona_buy, therefore one could argue that the later should receive a higher weight factor to take in account the inferior meaning for a item. However one could also argue that the predicate (root), embodies more meaning in a sentence.

The initial weight score will be given by the TF-IDF score. Perhaps we need to establish (by empirical means) which type of elements will give an additional weighting score. One could argue the root (or predicate), root_buy has more significant ‘meaning’ in real-world than an adjective amod_red.

We will need to define an output scheme which represents the documents and their scores. The resulting vector will describe the match between query and document. Below (figure 5.2) we have illustrated how we calculate the match using a diagram.

<table>
<thead>
<tr>
<th>docid</th>
<th>dobj_paint_buy</th>
<th>nsubj_jona_buy</th>
<th>root_buy_root</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>22</td>
<td>0.8</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>234</td>
<td>0.41</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td>654</td>
<td>0.21</td>
<td>0.34</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Figure 5.2: Illustration on the match-calculation

Fig 5.2 resembles the ‘matching entities’ in the diagram above. Due to the TF-IDF score of the grammatical elements, every document has a slightly different score.
If we look at the conceptual matrix representation of all documents, terms and TF-IDF scores, it looks like figure 5.3. `termDocumentsMatrix` is the conceptual representation of every term for every document. We have listed an example, as one will recognize, this is a typical sparse matrix in which all but a few columns per row are zero. `termQueryVector` is in essence the same matrix/vector as `termDocumentsMatrix` being it with just one row (or document if you will):

\[
\text{termDocumentsMatrix} = \begin{pmatrix} 0.1 & 0.0 & 0.0 & 0.3 & 0.6 \\ 0.0 & 0.0 & 0.0 & 0.9 & 0.1 \\ 0.0 & 0.1 & 0.1 & 0.0 & 0.8 & 0.0 \end{pmatrix}
\]

\[
\text{termQueryVector} = \begin{pmatrix} 0.2 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.5 \\ 0.3 \end{pmatrix}
\]

Figure 5.3

We could (also conceptually wise) reduce all columns which are 0.0 in the `queryVector` and remove the corresponding columns in the `termDocumentsMatrix` (figure 5.3) to speed up the lookup process. However, this behaviour does not actually occur. It is implicitly done in the algorithm. The end result would look like the figure below 5.4

\[
\text{documentMatrix} = \begin{pmatrix} 0.1 & 0.3 & 0.6 \\ 0.0 & 0.9 & 0.1 \\ 0.0 & 0.8 & 0.0 \end{pmatrix}
\]

\[
\text{queryMatrix} = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.3 \end{pmatrix}
\]

Figure 5.4
We can calculate the final scores by multiplying the \textit{documentMatrix} with the \textit{queryMatrix}. In order to present the user with some tuning options, he can tune the various coefficients in the query matrix to infer the outcome. The final calculation would look figure 5.5 where \( x, y \) and \( z \) denote the user configurable weights for each of the grammatical entities:

\[
\text{finalScoreMatrix} = \begin{pmatrix}
0.1 & 0.3 & 0.6 \\
0.0 & 0.9 & 0.1 \\
0.0 & 0.8 & 0.0
\end{pmatrix} \ast \begin{pmatrix}
0.2 \\
0.5 \\
0.3
\end{pmatrix} \ast \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

Figure 5.5: Weight introduction

The final matrix (if the user chooses: \( x = y = z = 1 \)) yields: Figure 5.6 indicates that the 2\textsuperscript{nd} document

\[
\begin{pmatrix}
0.35 \\
0.48 \\
0.4
\end{pmatrix}
\]

Figure 5.6: The final output

has the best match. However, if the user had opted other weighting factors than \( x = y = z = 1 \), and \( y < z \) would be true, chances are that the 3\textsuperscript{rd} would have had a higher score than the 2\textsuperscript{nd}.

5.2.3 Analysis on matching

\textbf{Initial matching.} We should ask ourselves how to access the data. We could argue using query to find document, or process document in order to gain query. Our initial approach which entails using the terms extracted from the query is a very fast way to access records. However if the record like root is root is encountered we may try to retrieve a list of size 10\textsuperscript{7} document-descriptors. These kind of numbers may make the search program overflow it’s allocated memory. In order to mitigate this possibility we need to do something extra.

\textbf{The 2\textsuperscript{nd} approach} is to utilize map-reduce for search. For low document counts this would be severe overkill and introduce massive overhead. But for really large collections it offers flexibility. This would require an alternative database layout, one in which there is a lookup table which points for every term in the query to a document. Like a ‘link-table’ in SQL, different mappers spread on multiple nodes would retrieve each such document, process it’s TF-IDF scores and return the documents they processed to the reducer. But before they would do so, they should run a \textit{combine operation}[11]. A combiner is in essence a reducer that is run on top of the mapper, before it posts it’s results. This can minimize the number results significantly and optimize the sorting process, as in the final stage, the framework would have to sort a more compact list, leading to reduction of memory usage of the reducer(s) and overall network utilization. We can use the combiner due to the fact that multiplication (the operation we are doing) is associative. In general, combiners can and should be used if the \textit{function} implemented is both \textit{commutative} and \textit{associative}[12].

Example of the combiner in place: if the mapper emits Key1:value1, Key2:value2, Key1:value3 and the reducer counts (as always) key-wise, the mapper will emit Key1:(value1 + value3), Key2:value2 when using the combiner. This reduces network overhead, since less data is transmitted. It also speeds up sorting as every mapper sorts the list prior to the overall sort for the reducers.

\textit{Example of a commutative operation which is not associative} is like the function:

\[
f(x, y) = \frac{x + y}{2}
\]
It is commutative as swapping the input values doesn’t affect the outcome: \( f(1, 2) = f(2, 1) \)
but it is not associative: \( f(1, f(2, 4)) = 2 \) and \( f(f(1, 2), 4) = 2.75 \)

This general principle may gain more context if we would choose the document as key, and a value as
TF-IDF score for a specific term. Prior to emit multiple occurrences of the document (for each parse-
tree-element) with different scores, the mapper merges these to just one, with a partially calculated score.
The reducer may receive #-mappers similar keys instead of multiples per mapper, reducing the memory
footprint significantly.

The 3rd approach is to redesign the first one. Instead of opting for a sole record based on the
grammatical occurrence, we redesign the row-key design. Where we first worked with the \( \text{dobj}_\text{paint}_\text{buy} \)
based key, we now append the document-number. HBase gives us the possibility to scan via partial keys.
Using this technique, we just get a record per get action and can decide to reject a row-key scan if the
number of affected rows exceeds a certain amount, otherwise we can iterate over this list in a lazy way,
preventing memory overflows. But this has the downside that the structure of keys cannot be hashed to
achieve a normal distributed structure.

General principle In any case, the usage of the IDF score as a filter would be smart. Terms which
occur frequently like ‘the’ and ‘is’ could be rejected by defining a minimum IDF score in order to be used
in the matching scheme. If the search query just describes these words, one could always use these words,
but as they have so little descriptive meaning one could argue that those results would be doubtful in
terms of ordering.

5.2.4 High level description of software

The pseudo code equivalent in memory could be implemented inspired by this pseudo code:

```java
1 /*
2 * TF-IDF matching pseudo code
3 */
4 def calcIdf(termOccurrences, totalDocuments)
5   return Math.log(totalDocuments / (1.0 + termOccurrences))
6 end
7
8 # the indexed parseTreeElements every record contains the tf score
9 docs = Hash.new()
10 docs["dobj_paint_buy"] = new Record()
11 docs["nsubj_jona_buy"] = new Record()
12 docs["root_buy_root"] = new Record()
13
14 # the indexed parseTreeElements every record contains the tf score
15 query = ["root_buy_root", "nsubj_mike_buy"]
16
17 # the score storage
18 queryScore = []
19 for (queryPart in query)
20   # the idf score times the tf score (quick, assumes every term 1 occurrence)
21   queryScore[queryPart] = calcIdf(1, query.length) * (1/query.length)
22
23 # calculate the idf and tf for EVERY document in the docs hash (equals whole corpus)
24 for (document in docs)
25   # the idf score times the tf score (quick’n dirty)
26   document.addField("idfScore")
27   document.set("idfScore", calcIdf(doc.getDocs, docs.length))
28
29 # building result set by getting the elements from the query out of the hashmap:
30 for (q in query)
31   record = docs.get q
32   documentsInRecord = record.getDocs
33   idf = record.get("idfScore")
34   # for each document, listed within the record
```
for (r in record.getDocs())
    tf = r.getTf
# calculate the score of the tfidf from the
# record times the tfidf from the query.
score = tf * idf * queryScore[index]
puts "Document #{r} has a match-rating of #{score}"
}
}
query.each do |searchFor, index|
}
end

In a relational database we would use SQL to execute something like this to calculate the TF scores:

Please note that this example is taken from [13]. The output would consists of words and their frequencies. The only thing left to do would be update statement in which the idf score gets calculated.

CREATE TABLE Foo (  
  keycol INT NOT NULL PRIMARY KEY,  
  description NTEXT);  

INSERT INTO Foo VALUES (1, 'wrong contact. referred to sales');  
INSERT INTO Foo VALUES (2, 'referred to techsupport');  
WITH Num1 (n) AS ( SELECT 1 UNION ALL SELECT 1),  
Num2 (n) AS ( SELECT 1 FROM Num1 AS X, Num1 AS Y),  
Num3 (n) AS ( SELECT 1 FROM Num2 AS X, Num2 AS Y),  
Num4 (n) AS ( SELECT 1 FROM Num3 AS X, Num3 AS Y),  
Nums (n) AS (SELECT ROW_NUMBER() OVER (ORDER BY n) FROM Num4),  
Words (word) AS ( 
  SELECT SUBSTRING (' ' + descr + ' ', n + 1,  
    CHARINDEX(' ', ' ' + descr + ' ', n + 1) - n - 1)  
  FROM Nums  
  JOIN (SELECT CAST (description AS NVARCHAR(MAX)) FROM Foo) AS F(descr)  
  ON SUBSTRING(' ' + descr + ' ', n, 1) = ' '  
  AND n < LEN(' ' + descr + ' ')  
)  
SELECT word, COUNT(*) AS cnt  
FROM Words  
GROUP BY word;
/*  
word  cnt  
------------  
contact  1  
referred  2  
sales  1  
technical  1  
to  2  
wrong  1  
*/
5.2.5 The approach

In the chapter "The system we built" we elaborate on our implementation of the search mechanism.

The search is quite similar to the indexing part. The user defines his/her input, this input is split (if it consists of multiple sentences), and fed to the parser. The parser extracts the grammatical components. After all components are extracted, we calculate the TF and IDF score for the query. Then we search the database for similar parse-tree-elements and retrieve their TF and IDF score.

The sorting of the scores goes as following, we multiply the TF and IDF score of the documents and the query. This results in a final score per parse-tree-element per document. We then add the scores which share the same document by addition. The result is then sorted from high to low and presented to the user. We have illustrated this in figure 5.7.

Figure 5.7: Flow chart from query to match results
Chapter 6

Distributed Computing

6.1 What is distributed computing

Distributed computing is not to be confused with SMP (Symmetric Multi Processing). Where SMP is the concept of multiple microprocessors (CPU’s as you will) co-operating to complete a task/tasks, distributed computing is about processing a task on multiple computers. These computers are normally inter-connected by conventional network apparatus. One could incorporate SMP within a distributed computing cluster, but not visa-versa. Such a distributed computer group or commonly referred as cluster, can compute immense tasks that would otherwise be impossible to compute within a reasonable time span. A single computer with X CPU’s would be able to execute some tasks based on 1 petabyte (1.000 Gigabyte) of input. However if this computation needs to be executed every 24 hours and the results of the previous run(s) are needed for the next one. It would be infeasible. This is where distributed computing steps in. One important note is that in order to be able to utilize distributed computing, the task needs to be parallelizable. Eg trying to calculate Fibonacci sequence in a parallel environment is simply impossible as one needs to use the previous result to compute the next. In general, one could state that recursive tasks are not well suited for distributed computing. However often in modern large scale computations, a computation has to be done for each element of the data set. All of these ‘mini-tasks’ can be done in parallel. Especially profiles and websites, look at Facebook and Google.

Let $A$ and $B$ be collections, 
Let $f(A) = B$, the $f()$ has the properties : $A = A_1 \cup \ldots \cup A_k$ a random partition of $A$
Let: $\pi$ permutation of $\{1, \ldots, N\}$
Then: $f(A)$ can also be calculated as:

$$f(A) = f(A_{\pi(1)}) \oplus f(A_{\pi(2)}) \oplus \ldots \oplus f(A_{\pi(N)})$$

Where $\oplus$ an operator is which is both commutative as associative.

![Figure 6.1: principle of map-reduce](image)

**Basic principle:** In this example the function $f()$ is applied for every ‘sample’ in the collection. This paradigm is also called a *MAP function*.

Another component in parallel computing is the *reduce* or often *accumulate, fold, compress* function. This function has the sole purpose to recombine a group of data into a ‘smaller’ one. This may seem abstract, but I will illuminate this with an example:

Let us say, we have a (large) list of words, and for every word we want to know how often it occurs in the given collection. We split this large list into pieces, so every task has it’s own subset of words. First we apply a map function to the input text. Our map function has every separate word in the input list as input and will emit/output a tuple of data, the word, and it’s occurrence:

$$\forall \text{word} \in \text{word – list, } f(n) \to (\text{word, } 1)$$
After all words of the word list are processed, we have a huge collection of (word,1) pairs. If we recombine these as \( \text{key} = \text{word} \) and \( \text{value} = (1,1,1,1,1,\text{etc}) \) and feed this to a reduce function:

\[
f(\{(\text{key},\text{values})\}) \rightarrow \{(\text{key},\text{somevalue})\}
\]

the result will be a single value per key.

6.2 Distributed in relevance with this research

One of the objectives of this research is to incorporate distributed technology as one of its key components. Of course there are more reasons why to utilize concepts like these into search engines. Distributed computing enables us to harness the power of virtually indefinitely large clusters to process tremendous amounts of data. For example, Google (although they are very secretive about the absolute numbers) has an estimated number of 900,000 servers in service (as of 2011).

With their vast amount of data, they still succeed in querying for a random thing and return results in as little as 390 milliseconds (the query was randomly: tube blue locker grasshopper dell). Without these types of design its simply impossible to store such amounts of data on a single computer, let alone serve large quantities of concurrent users within small time-frames.

During this research, we will work in contrast to Google, with very small dataset, sets that will fit on a single modern computer. However we will utilize the cluster to research possibilities in terms of parallel computation, scale-enlargement. During a later stage we will use bigger datasets.

6.2.1 Large datasets

Distributed environments are often used to store data. In the old days, it was quite common to buy (expensive) RAID controllers in order to achieve higher throughput and redundancy. Another way was to buy expensive external storage systems like a SAN (Storage Attached Network), to store even more data. However with today’s data-demands these options are no longer sufficient. Therefore expensive controllers and dedicated storage devices are replaced for cheap multi-disk servers without redundancy. Instead of redundancy within the computer, there is a shift towards redundancy within the network. This shift in addition to data-safety, has an additional benefit: the redundancy of availability. If my server was to die in the ‘old-days’ the server had to be re-installed and the raid-array (collection of disks, was managed by a controller) had to be migrated to the new machine. With this new way of storing data, if an entire datacenter becomes unavailable (due to a power outage for example), the data is still available at another geographical-location. Another big benefit is that due to the multiple locations where the data is stored, client can be served more quickly, based on their location in respect to the location of the datacenter. Last but not least, an advantage is that due to spread of the data redundantly over multiple machines, computation of data can be completed more quickly as multiple nodes can process parts of the data already stored locally. Eg a standard replication factor of Hadoop (distributed file-system), is three. Meaning that 3 physical machines have a copy of the data and can process it simultaneously, or stream the parts to other nodes for processing. This enables us to use larger datasets and higher throughput as disk-access is usually the biggest bottleneck. One can easily see that with a replication factor of 3, the potential read-speed of a data-block is 3 times the read speed of a single node, but do note a good network infrastructure is paramount.

6.2.2 Expensive parsing

One of the key tasks of a distributed environment is to spread the load (or calculations) out over multiple machines. One can imagine that parsing documents on 1 computer is slower than on 10. As we will design our application to utilize as many machines as possible during the generation of the index, we hope to see a decrease in time for processing the data.

6.2.3 Time constrains

Another reason for distributed computing is the amount of time spend on computing versus data size. Some calculations are simply too complex to process within a reasonable time-frame.
### 6.3 Distributed computing $\equiv$ cloud computing

Cloud-computing is popularly known as services in internet (best know, are the store services). Many will be familiar with Dropbox (a service that synchronizes your files to ‘the cloud’) and Google’s (Google doc’s, Picassa, Gmail). These are typical examples of cloud computing. Less known services like ‘Amazon’s EC2’ and Azure-platform enable users to run their programs on large clusters. These services (at least EC2), enable users to run distributed file-systems and task. They feature features like dynamically increase of resources and more allocation of CPU-cycles. They even have GPU (Graphical Processing Unit) acceleration devices available to deal with coordinate translations and matrix operations, these are often used to bulk process expensive calculation like generation rainbow-tables. These services are also commonly used to deploy distributed file-systems. During this research we might look at the possibilities running our applications on such cloud-providers.
Chapter 7

The system we built

7.1 Overview

7.1.1 Black-box

What should the system do? We want to build a ‘complete’ system which enables a user to search for a specific document. The user has as a ‘tool’ the possibility to input a query in the form of proper formatted sentence(s). In order to supply this system or black-box, we need some conceptual building blocks which form this system.

7.1.2 Indexing

Most search engines rely on 2 components, an index and a search function. In this section we are going to describe what an index is and how it looks according to our point of view. And index is a very broad concept, but it’s core function is to offer some kind of lookup scheme to find something. One can compare this to a phone-book. In order to search an arbitrary number, one can find someone quickly by first jumping to the proper city and then the last-name. Lastly one iterates trough the last names that are similar to the one you are actually looking for and voilà, there is your answer to your query. In order to do such a quick lookup, the ‘data’, in this case the name, city and phone-number are stored in a structured form (sorted on their properties) so one can quickly find someone, after this pre-processing, the lookup is easy and fast.

7.1.3 Our index

Our index will consist of a similar conceptual index, however it’s a bit more complex than your average phone-book. Every document (the conceptual person describe above) consists of sentences, layout, words etc. All these properties are descriptive for the document. If we would have a document about ‘dogs’ and another one about ‘cats’, we could compose an index which enumerates a list of animals and the documents in which they occur. Strictly speaking this would be an index. One selects the animal of interest and looks at the documents linked to this animal. However, our system should be able to somehow describe all various topics, and therefore a statistical approach would seem appropriate. One could take a an inverted-index which tells you which keywords are in which document and then calculate which document resembles the most keywords with the search query. As mentioned before, we would like to utilize a Natural Language Parser to identify grammar. Using these grammatical concepts, we hope to achieve a more user-friendly result. By carefully weighting different combinations of grammatical concepts (eg noun, adjective) we opt for finer results. Our index will comprise of trees which describe sentences which describe documents. Components to match these sub-collections with the documents indexed.
7.1.4 Processing queries

The process of query interpretation is very similar of that of the index-generation, except this time we would have to index just the query, a few sentences at most. We feed this ‘input’ into the parser, extract the tree describing the query. Using these components we proceed to the matching part. The big difference is that we now need to parse a few sentences and thus complete within a few seconds. Which is still relative long but nothing compared to the index-generation.
7.2 Details

This chapter is about the technical design. Here we will discuss our implementation of the Software, front-end (user query) and back-end (indexing).

7.2.1 Research lab

The cluster consists of 9 desktop systems with homogeneous specifications:
- Intel Pentium 4, 2.8Ghz Single core + hyper-threading
- 1GB DDR2 memory
- 80GB 7200 rpm hard-disk
- 100Mbit Full-duplex Ethernet

7.2.2 Infrastructure

The nodes are managed by 1 Master node. The inter-connectivity between them is with help of 3 switches, therefore a close resemblance to a star-network configuration.

![Network Layout Diagram](image_url)

Figure 7.1: The network layout of our setup
7.2.3 Distributed file system

The distributed file-system we choose as mentioned prior is Hadoop. Hadoop’s file-system is comprised of large files/blocks, optimized for sequential reads. Once written, it cannot be altered, only (since recently) appended. These blocks have a default size of 128 MB. This choice of design has a great impact on performance. Due to these properties (write once, read many), Hadoop can very efficiently distribute task amongst it’s nodes. The concept allows for easy management of blocks. However the drawback is that one cannot ‘open a file and edit it’s content. Hadoop is centered around the concept of batch processing and streamed data access. This is where HBase steps in. As Hadoop and HBase integrate nicely together, it i a natural choice to utilize them on-top of each other. The Map-Reduce framework enables users to implement their algorithm in a very compact fashion. The framework abstracts the un-pleasantries of data-distribution, data-synchronization and data-management. Additionally it enables users to start a job relative easily and synchronizes all participating nodes to do their part of the job, and access of data. As all data are spread over the distributed file-system, the framework tells the who of the nodes should run the job on their piece of the data, or if they don’t have any locally stored data, retrieve a piece of the data from another node. and process that.

7.2.4 Software

For easy administration, we have chosen for Ubuntu (11.1, 32-Bit, server edition.) At first we wanted to utilize the Cloudera free edition [15] in order to do easy cluster maintenance. However, Cloudera offers 64-Bit (management)-binaries only, therefore we had to do a manual installation of the software packages. We wrote some scripts to ease distribution of software among nodes to get the cluster going.

The software packages used during this experiment are:

- Hadoop 1.0.0 (the distributed file-system and map-reduce framework)
- HBase 0.92.0, r1231986 (the column-oriented database on top of Hadoop)
- Ssh/Scp (remote access/file-copy and maintenance-scripts of Hadoop)
- ntpd for time-synchronization (important for Hadoop/HBase)
- Tomcat 7.0 (Webserver)

They are distributed in the following order across the available hardware (fig7.3):

7.2.5 Analysis of parser

Our input sentence is Jonas is buying paint in order to paint his garden house. This sentence is fed to the parser. We have create a little test-Class to experiment with its output, stemming features and other functions. It is listed in the appendix under 10.4.
<table>
<thead>
<tr>
<th>Node</th>
<th>Hadoop processes</th>
<th>HBase processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-master</td>
<td>pri-namenode, job-tracker</td>
<td>zookeeper, master server</td>
</tr>
<tr>
<td>ir-slave1</td>
<td>sec-namenode, tasktracker</td>
<td>region-server</td>
</tr>
<tr>
<td>ir-slave2</td>
<td>data-node, tasktracker</td>
<td>region-server</td>
</tr>
<tr>
<td>ir-slave3</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
<tr>
<td>ir-slave4</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
<tr>
<td>ir-slave5</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
<tr>
<td>ir-slave6</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
<tr>
<td>ir-slave7</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
<tr>
<td>ir-slave8</td>
<td>data-node, tasktracker</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: Daemon/processes distribution

**Parsers Output**

- `gov:buying-3 rel:nsubj rel:Jonas-1`
- `gov:buying-3 rel:aux rel:is-2`
- `gov:ROOT-0 rel:root rel:buying-3`
- `gov:buying-3 rel:dobj rel:paint-4`
- `gov:paint-8 rel:dep rel:order-6`
- `gov:paint-8 rel:aux rel:to-7`
- `gov:buying-3 rel:purpcl rel:paint-8`
- `gov:house-11 rel:poss rel:his-9`
- `gov:house-11 rel:nn rel:garden-10`
- `gov:paint-8 rel:dobj rel:house-11`

Let us first order the data more intuitively:

- `gov:ROOT-0 rel:root rel:buying-3`

Our structure:

```
buying ( modifiers )
```

the root buying we get:

- `gov:buying-3 rel:nsubj rel:Jonas-1`
- `gov:buying-3 rel:aux rel:is-2`
- `gov:buying-3 rel:dobj rel:paint-4`
- `gov:buying-3 rel:purpcl rel:paint-8`

Our structure looks thus as:

```
buying nsubj Jonas aux is dobj paint purpcl (paint)
```

The second paint informs us:

- `gov:paint-8 rel:dep rel:order-6`
- `gov:paint-8 rel:aux rel:to-7`
- `gov:paint-8 rel:dobj rel:house-11`

Our final structure looks like:

```
buying nsubj Jonas aux is dobj paint purpcl (paint mark in dep order aux to dobj house)
```

Lastly we heave the branches dangling underneath house:

- `gov:house-11 rel:poss rel:his-9`
- `gov:house-11 rel:nn rel:garden-10`
7.3 Result

*buying nsubj Jonas aux is dobj paint purpcl (paint mark in dep order aux to dobj (house poss his nn garden))*

This looks graphically as:

![Diagram of the sentence structure](image)

Figure 7.4: Jonas is buying paint in order to paint his garden house

7.4 The implementation

When we look at the implementation phase, we need to implement a few pieces in order to make it work.

7.4.1 The index generating

In order to generate the index, we need some input or documents. We anticipate that the parsing will consume the most of the CPU cycles available. We started with the import of the documents in to the database. We decided to do this initial step in order to store some meta-data with the original documents, as well as means to easily retrieve the document with our index. Furthermore the processing of XML is not trivial in Hadoop, however line-based processing of documents is. Therefore we had to pre-process the data in a line-based format, to ease implementation.

**The importer** consists of a small Java application that reads our data-set, and stores each document with some META-data like document-name/file-name. The tools will be a single threaded application that reads the documents, create the initial file-structure on Hadoop in order to index it. One can think of this application as a tool to convert an arbitrary format to a homogeneous one. For example, the cran-dataset has another layout than the dump of Wikipedia, (which is a large 35 GB file with an XML structure). As mentioned previously in the functional design, we need to parse each document for grammatical properties. As this is a very CPU-intensive task, it's clearly something we want (and can) do in parallel.

**To extract the grammatical entities** we utilize the Map-reduce framework to distribute this work among available nodes hence the pre-processing of the data in a workable format. Each node will receive a list of records, process it’s grammar and writes the record to the HBase database, it also accumulates the number of documents. The reduce class will then aggregate all processed documents and writes a final document-count value. This value is needed to implement the TF-IDF algorithm.
The items need normalization or stemming due to the by-now obvious reasons. We do this step after we have extracted the grammatical entities, prior to storing the entities in the database.

For the matching we need to combine the weights of each of the elements with the TF-IDF score of the query. Multiple occurrences in documents lead to higher score/relevance on that particular ‘subject’

The interface for query needs to be built with easy-of-use in mind. The interface remains a proof of concept, however for easy verification and benchmarking it is paramount to have an easy way of evaluating the system and tuning parameters, like weights. Therefore we choose to make a Servlet which runs in the Tomcat Application server [10].

7.4.2 Performance tuning

Due to the very poor hardware specifications, we are forced to deviate from the default configurations. For example, Hadoop reserves by default 1 GB of RAM, this is more the is available after booting just the Operating System. HBase is usually run in conjunction with 4/8 GB at minimum. See the appendix for the appropriate changes: [10.6]
Chapter 8

Experiments and results

In this chapter we will evaluate our finding concerning the implementation phase of the research. We opt to describe the progress, and performance implications of certain design decisions.

8.1 Design of the experiment

As parsing of text utilizing a Natural-Language-Parser, is expensive we are benchmarking various combinations of parallel nodes and datasets.

8.1.1 Input data

In order to import a 35 GB text-file, a simple DOM-tree parser will not work. Therefore we wrote a import-tool. This tool utilizes a event-driven parser (Sax parser) and tries to strip out all the Wiki-codes, strange characters and other irregularities. See Appendix:The Wikipedia XML-importer for the actual source code.
8.2 Benchmark

We benchmark the system with help of a subset of the Wikipedia data.

8.2.1 Indexing Wikipedia data

Below are the benchmarks, of the parser in a configuration with 1, 2, 4 and 8 nodes running parallel on a subset of the Wikipedia data. The set consists the following properties:

- Number of documents: 16,036
- Number of sentences: 157,029
- Total size of all documents: 40.92 MB

In this test we have a subset of all Wikipedia data loaded onto the distributed file-system. Using a map-reduce job, we process each of the documents, extract the grammatical elements and count the number of documents and sentences processed. In this test we don’t utilize the matching part. We have split the extracted Wikipedia documents in chunks of 5 MB. In the first test we concatenated 8 blocks of 5 MB together and let 1 mapper process it entirely. During the second test, we concatenated those same 8 blocks in two pieces of each 4 block and let it process by 2 mappers. The 3rd we take those same 8 blocks, concatenate them into 4 pieces of each 2 blocks, and the last run its 8 blocks on 8 mappers. We utilize the time, specified by the framework, this includes set-up time and actual calculations.

![Exponential scale](image1.png)  
![Linear scale](image2.png)

**Figure 8.1: Exponential scale**  
**Figure 8.2: Linear scale**

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>1 mapper, 1 reducer</th>
<th>2 mappers 1 reducer</th>
<th>4 mappers 1 reducer</th>
<th>8 mappers 1 reducer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes to complete</td>
<td>541 minutes</td>
<td>273 minutes</td>
<td>143 minutes</td>
<td>84 minutes</td>
</tr>
</tbody>
</table>

**Figure 8.3**

As one can clearly see in the **Exponential scale**, the results of double the number of nodes results in half processing time, a near linear scale enlargement.

8.2.2 Costs

Currently our cluster consists of 8 slaves and one master. On each slave is an instance of Hadoop installed. HBase is installed on 3 machines, the master node and two slave nodes. Zookeeper (which coordinates the region servers) is also present on the master node. Below we have depicted the load distribution. As one can see just one region server handles all request, this is due to the nature of regions in HBase, this behavior can be changed (more on this in paragraph **Overall HBase performance**).
HBASE load overview during indexing with 8 nodes

<table>
<thead>
<tr>
<th>Server-Name</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-master.cs.ru.nl</td>
<td>requests p/s=1, # OnlineRegions=4, usedHeapMB=99, maxHeapMB=499</td>
</tr>
<tr>
<td>ir-slave1.cs.ru.nl</td>
<td>requests p/s=0, # OnlineRegions=2, usedHeapMB=99, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave2.cs.ru.nl</td>
<td>requests p/s=316, # OnlineRegions=4, usedHeapMB=158, maxHeapMB=499</td>
</tr>
<tr>
<td>Total: servers: 3</td>
<td>requestsPerSecond=317, numberOfOnlineRegions=10</td>
</tr>
</tbody>
</table>

Figure 8.4: Illustration of load on the HBASE cluster

8.2.3 Room for optimization?

Overall HBase performance  
HBase uses regions to group a set of keys. A region can only be served by a single Region-server. For those looking at the graph 8.4 might have noticed that all but one server are idling while one is processing a lot of requests per/second. This behaviour is very typical for small data sets. HBase will split a region automatically once it reached a certain file-size. Normally this is set to 256 MB. As we are working with such low-end hardware, we would be wise to lower this number considerably. By manual intervention we can split the region in to multiple parts (by adding appropriate commands to the indexing code). This will result in a better distribution of load, which increases overall performance. Below in 8.5 the result is displayed when using more regions (in this case there were 80).

HBASE load overview during indexing with 8 nodes, multiple regions

<table>
<thead>
<tr>
<th>Server-Name</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-master.cs.ru.nl</td>
<td>requests p/s=190, # OnlineRegions=28, usedHeapMB=173, maxHeapMB=499</td>
</tr>
<tr>
<td>ir-slave1.cs.ru.nl</td>
<td>requests p/s=22, # OnlineRegions=28, usedHeapMB=99, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave2.cs.ru.nl</td>
<td>requests p/s=122, # OnlineRegions=24, usedHeapMB=109, maxHeapMB=499</td>
</tr>
<tr>
<td>Total: servers: 3</td>
<td>requestsPerSecond=332, numberOfOnlineRegions=80</td>
</tr>
</tbody>
</table>

Figure 8.5: HBase load - multiple regions

If we are looking at the load between regions, we see some interesting behavior. When HBase decides to split a region, it basically splits the old region in the middle, the 1
th or left part keeps the old name, the 2
nd region starts with the splits regions based on right part of the split. What we see is that several regions have a high number of requests, where others have lower or almost no access at all. We see that region r4 (between abbrev|boojiboy|utc and abbrev|bowen|utc) is accessed very frequently, while region r1 has almost no access. This behavior occurs due to the non-normal distributed format of our keys. Obviously are there more keys which start with abbrev|1|utc and end with abbrev|boojiboy|utc, than there are keys which are between “” and abbrev|1ne|utc. In order mitigate these differences (which could lead to certain server having a higher load due to a busy region), we could hash the keys with a (non cryptographically) hash function that has a near normal-distribution characteristic, low collision rate and cheap computation.

HBASE region overview

<table>
<thead>
<tr>
<th>region</th>
<th>server</th>
<th>startKey</th>
<th>stopKey</th>
<th>request</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>ir-slave1</td>
<td>abbrev</td>
<td>1ne</td>
<td>utc</td>
</tr>
<tr>
<td>r2.abbrev</td>
<td>1ne</td>
<td></td>
<td>ir-slave1</td>
<td>abbrev</td>
</tr>
<tr>
<td>r3.abbrev</td>
<td>1</td>
<td></td>
<td>ir-slave1</td>
<td>abbrev</td>
</tr>
<tr>
<td>r4.abbrev</td>
<td>boojiboy</td>
<td></td>
<td>ir-slave1</td>
<td>abbrev</td>
</tr>
<tr>
<td>r5.abbrev</td>
<td>bowen</td>
<td>ir-slave1</td>
<td>abbrev</td>
<td>bowen</td>
</tr>
<tr>
<td>r6.abbrev</td>
<td>chrislg1970</td>
<td>ir-slave1</td>
<td>abbrev</td>
<td>chrislg1970</td>
</tr>
</tbody>
</table>

Figure 8.6: regions in a table

As one can see in figure 8.6 some regions are loaded very heavily while others are idling. As HBase balances just on the number of regions per server, rather than the load amongst servers, this can and does give us problems. When we are running a large job eventually a server collapses under the load. Zookeeper (a coordinating service part of the HBase ecosystem) will than assign another servers to take on this load, as one can imagine this instance will go down even faster as it has to serve both it’s own regions and the newly assigned. This will eventually lead to total cluster failure.
**Key re-structure** Although we know that our hardware is all but perfect, we try to squeeze a little more out of it. To do so, we changed our key structure. Rather than inserting `abbrev[1ne]` into the key, we use a key with a hash digested version of our key: `-1049442533-` which we suffix with digested version of document-id. The resulting load on the servers is displayed in figure 8.7:

<table>
<thead>
<tr>
<th>ServerName</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-master</td>
<td>request p/s=37, OnlineRegions=6, usedHeapMB=109, maxHeapMB=499</td>
</tr>
<tr>
<td>ir-slave1</td>
<td>request p/s=66, OnlineRegions=7, usedHeapMB=105, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave2</td>
<td>request p/s=58, OnlineRegions=6, usedHeapMB=37, maxHeapMB=499</td>
</tr>
<tr>
<td>ir-slave3</td>
<td>request p/s=13, OnlineRegions=7, usedHeapMB=104, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave4</td>
<td>request p/s=16, OnlineRegions=7, usedHeapMB=84, maxHeapMB=499</td>
</tr>
<tr>
<td>ir-slave5</td>
<td>request p/s=84, OnlineRegions=6, usedHeapMB=95, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave6</td>
<td>request p/s=56, OnlineRegions=7, usedHeapMB=101, maxHeapMB=505</td>
</tr>
<tr>
<td>ir-slave7</td>
<td>request p/s=17, OnlineRegions=6, usedHeapMB=60, maxHeapMB=505</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>servers: 8</td>
</tr>
</tbody>
</table>

Figure 8.7: Normal key distribution load overview

What this basically does is scatter all possible hashed representations of the keys over all regions. This makes sure that not all terms starting with an ‘a’ end up at the same server in the same region, but does ensure all terms which are have the same start-bits (after hashing) are put together in the sorted list. See the figure 8.8.

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>713507990</td>
</tr>
<tr>
<td>aa</td>
<td>1381508319</td>
</tr>
<tr>
<td>aaa</td>
<td>-306368007</td>
</tr>
<tr>
<td>a1</td>
<td>1536319132</td>
</tr>
<tr>
<td>bike</td>
<td>662162579</td>
</tr>
<tr>
<td>bikes</td>
<td>496685120</td>
</tr>
<tr>
<td>123</td>
<td>1005521553</td>
</tr>
<tr>
<td>1234</td>
<td>1880127200</td>
</tr>
</tbody>
</table>

Figure 8.8: Example of the Murmur hash function

As you probably recall HBase stores it’s key’s in lexicographically order, so spreading all terms over all regions roughly equally (as the Murmur’s output is near linear). Your final key structure looks like `-1000998945-1022125105`

**The differences** between the two key structures are depicted in figure 8.9 and 8.10. The total number of records in our table is 2,682,779 records.
When we compare with the occurrences in the original:

![Figure 8.9: hashed keys, start char](image)

Although one can see the hash-function does not distributes as a perfect normal distribution, it has a better spread over the whole line.

**Retrieval performance**  In order to make HBase faster in term of records lookup we can apply an Bloom-filter to a table. What this basically does is create buckets in which it stores hashes. Like an hash function utilizes buckets to do quick retrieval of values, in a similar fashion HBase utilizes Bloom-filter. A Bloom-filter is basically a large bit-array. This array is populated with all the hashes of members.

The only thing the system has to do is generate the hash-value of a certain search-term. If it does an 'AND' operation between the bucket and the just calculated hash, and the remaining value is the same as the calculated hash **chances** are that this hash is actually a member. If the value of the remaining hash is unequal, the hash is **definitely not** a member. So Bloom-filter are a very efficient way to determine memberships. However they may produce false-positives. One of the constraints for a well designed Bloom-filter, is the use of hash-functions with low collision ratios and an evenly distributed output. Another important property is the initialization of the Bloom-filter. The filter must be create with an expected number of items in it. For every member inserted, the Bloom-filter is rewritten as \( \text{Bloom-filter} = \text{Bloom-filter and newHash(member)} \). A general caution with respect to bloom-filters: if one overflows the number of members, the false-positive ratio will quickly increase, rendering the filter less and less usable, till eventually every membership test will result in a false positive due to the saturation of the bit-array (all bits are 1).

Enabling Bloom-filter in HBase decreases lookup times, but as in the real-world, nothing is free. For every key, 4 bytes are needed additionally for the Bloom-filter.

**These alterations** provide us with a 10% increase in documents before the system collapses due to the load. As we mentioned earlier, the HBASE utilizes the Hadoop file-system to store it files. As we also
mentioned Hadoop is build for streaming purposes and sequential reads/writes. If would want to alter a byte inside so called data-block, we would have to read the block to memory, alter the byte, and write the whole block back to the file-system. This is exactly why HBase keeps these files into memory. If it has to flush memory to disk, it has to write those blocks back to the file-system and remove the previous versions. During this indexing we have a lot of access to a wide range of regions. This forces HBase to keep everything in memory in order to server request quickly and thus the Heap is completely filled after about 22,000 documents or 178,000 sentences. HBase is notorious for it’s large memory footprints. We run our HBase nodes unfortunately with just 500 MB (there is nor more), a typical configuration on modern hardware would assign around 16 24 GB or RAM, that’s 32-48 times the amount we have.

Our improvement comes with a downside. As all our keys are hashed and suffixed with a hash as well. We cannot easily iterate over the entire collection. Instead we calculate the IDF score on retrieval of the records, as we than know which key we search, we use the startRow- and stopRow- scan attributed to retrieve all rows with the same prefix hash.
8.3 Distribution of costs

In this section we try to chart the different stages of indexing in a multi-node environment.

8.3.1 Indexing

This data is based on a 4-mapper run, processing 16,036 documents which a total of 157,029 sentences. Total runtime: 2hrs, 19mins, 42sec. The distribution of costs during indexing are depicted below:

Figure 8.11: Indexing resources (in terms of time) usage

Please note that the set-up- and clean-up phases together actually take 0.003412852803659 % of time. As the number of mappers increase so will the percentage of the mapper(s) share will decrease. The reducer is run in single (as there is effectively just one key emitted).

When we look at the CPU time spend we see the results depicted in the table below:

<table>
<thead>
<tr>
<th>module</th>
<th>time ms</th>
<th>time hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>time spend: Mapper</td>
<td>29,911,910</td>
<td>8.3</td>
</tr>
<tr>
<td>time spend: Reducer</td>
<td>7,480</td>
<td>0.0019</td>
</tr>
<tr>
<td>time spend: total</td>
<td>7,480</td>
<td>8.3</td>
</tr>
</tbody>
</table>

Figure 8.12: CPU time spend

It may appear strange at first, looking at table 8.12 when we just said that the total runtime was around two hours, but it makes sense if we remind ourselves that we used 4 nodes to do these calculations simultaneously.

Memory usage is displayed below:

<table>
<thead>
<tr>
<th>module</th>
<th>Bytes</th>
<th>GigaBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Memory: Mapper</td>
<td>1,763,631,104</td>
<td>1.64</td>
</tr>
<tr>
<td>Total Memory: Reducer</td>
<td>15,859,712</td>
<td>0.014</td>
</tr>
<tr>
<td>Total Memory: Total</td>
<td>1,779,490,816</td>
<td>8.3</td>
</tr>
</tbody>
</table>

Figure 8.13: Memory usage
Bandwidth usage is listed below. We have installed on one of our nodes a small package named Bandwithd\[10\]. This package utilizes the frequently used libpcap\[11\] package to capture the data send across the network. What we have done is started a fresh version of the bandwidth tool, just before firing of the indexing job. Again we have used our benchmarking data set, with an total of 8 mappers. We did two test, one in which we had the database enabled, and one in which we did not. This way we can ’see’ what traffic is send from all nodes towards all database-node-instances and what is plain map-reduce network traffic.

<table>
<thead>
<tr>
<th>Node</th>
<th>Total</th>
<th>Total Send</th>
<th>Total Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-slave2</td>
<td>655.6M</td>
<td>269.7M</td>
<td>385.9M</td>
</tr>
<tr>
<td>ir-slave3</td>
<td>297.3M</td>
<td>201.1M</td>
<td>96.3M</td>
</tr>
<tr>
<td>ir-slave7</td>
<td>138.9M</td>
<td>50.0M</td>
<td>88.8M</td>
</tr>
<tr>
<td>ir-slave4</td>
<td>125.9M</td>
<td>84.6M</td>
<td>41.2M</td>
</tr>
<tr>
<td>ir-master</td>
<td>34.5M</td>
<td>11.5M</td>
<td>22.9M</td>
</tr>
<tr>
<td>ir-slave1</td>
<td>17.9M</td>
<td>9.7M</td>
<td>8.2M</td>
</tr>
<tr>
<td>ir-slave5</td>
<td>15.6M</td>
<td>8.7M</td>
<td>7.0M</td>
</tr>
<tr>
<td>ir-slave6</td>
<td>12.5M</td>
<td>10.5M</td>
<td>2.0M</td>
</tr>
<tr>
<td>ir-slave8</td>
<td>9.6M</td>
<td>8.6M</td>
<td>987.7K</td>
</tr>
</tbody>
</table>

Figure 8.14: Total network Usage with HBASE enabled

As one can estimate when looking at figure \[8.14\] is a total 1.295G data exchange. This was measured at 1 node, which runs an instance of HBase, a tasktracker (the process which runs it’s part the job) and a data node. The total amount of data exchanged between nodes would be approximately 11.655 MB (when we multiply the usage of one node \(\times\) the number of nodes + master-node). This is quite a lot of information considered we have processed in input file of around 40 Megabytes. Please note that the key format we used in this test, was not one that had a normally distributed character.

<table>
<thead>
<tr>
<th>Ip</th>
<th>Total</th>
<th>Total Send</th>
<th>Total Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir-slave2</td>
<td>6.7M</td>
<td>2.7M</td>
<td>4.0M</td>
</tr>
<tr>
<td>ir-slave3</td>
<td>2.8M</td>
<td>2.7M</td>
<td>88.1K</td>
</tr>
<tr>
<td>ir-master</td>
<td>2.5M</td>
<td>675.2K</td>
<td>1.8M</td>
</tr>
<tr>
<td>ir-slave5</td>
<td>237.7K</td>
<td>115.4K</td>
<td>122.3K</td>
</tr>
<tr>
<td>ir-slave6</td>
<td>107.2K</td>
<td>35.6K</td>
<td>71.6K</td>
</tr>
<tr>
<td>ir-slave1</td>
<td>78.0K</td>
<td>50.7K</td>
<td>27.3K</td>
</tr>
<tr>
<td>ir-slave4</td>
<td>70.4K</td>
<td>57.9K</td>
<td>12.5K</td>
</tr>
</tbody>
</table>

Figure 8.15: Total network Usage with HBASE disabled

If we look at figure \[8.15\] derived when NOT using HBase, we see a very different usage characteristic. Instead of generating more than 1GB of network traffic as is displayed in figure \[8.14\] the overall usage is around 14MB. That is quite the difference. Another thing clearly visible in this table, is that nodes communicate with each other but not necessarily with all other nodes. The node on which we ran the benchmark (ir-slave2) had (nearly) no communication with node ir-slave6 whatsoever, as it was omitted from by the benchmark tool’s output.

8.3.2 Matching

The application How we have configured our searching application as following: We have a servlet (a object type in Java, used for web-applications). In Java there is some concept know as ‘application server’. An application-server (in our case Tomcat), as the name implies, it runs (or serves) applications. A servlet is such application. On could regard a servlet as a normal program, but is does have a limited lifespan, that of a website-request. Servlets have an \textit{init-method}. Objects that are instantiated during this \textit{init-method} have a global scope within the application-server’s servlet. We use this fact to instantiate an instance of our Natural-Language-Parser. As mentioned previously, the initial instantiation of the parser takes up to 2.3 seconds,
way to much time between every query (even in a test environment). By doing so, only the first request has to wait 3 seconds, after that it is just the processing time of the query. One should normally be cautious when using these methods of accessing global objects, it is not trivial that accessing a global instance is a thread-safe operation, in this case (being a prototype), it is a controlled environment.

We choose to build a very basic layout for our user-interface:

![Figure 8.16: The user-interface](image)

Via this web page the user is given the controls to search the database for sentences. In the screen shot we have searched for the sentence *the vampire walked during the night*. For testing purposes we didn’t calculate the IDF score which leads to a very high score for the term *root—root—walk and det—night—the*. Because we wanted the user the possibility to give extra input during the search, we set the *det—vampire—the* user-value to 100. This leads to a shift in scores for each found item; the document 3786760-Mainichi Film Award has a far higher score than the short documents following it.

See the appendix for the Java implementation: [10.6](#)

The results The matching results are not great due to our very limited reach of 22,000 documents and wide variety between documents. This causes results to be spread thin.

![Figure 8.17: Original keys](image)
8.4 Conclusion

Utilizing a distributed environment to index large quantities of documents is well worth the effort. However, when we look at how we implemented it for our specific situation things look different. We spread a very expensive calculation onto a cluster with very limited resources. One could argue whether the overhead introduced by using map-reduce to process 40 Megabytes of text-documents and processing them is justified. Especially when we take the storage of this output into account. As we have seen in this chapter, HBase is not suited in this low-end environment.

If we where to start a thread (or program) on every node, and let them write autonomously towards a database, this approach would gain overall performance and durability. Due to our low-end hardware and size of the cluster, it looks like that in this particular use-case it would be more efficient to run separate programs. Although Hadoop run rather well on the systems, HBase does not. On our available hardware I would not recommend HBase due to (by now) obvious reasons. As we have seen in the results section, HBase is not suitable for our low-end hardware.

With serious hardware however, one can harness the true power of HBase in collaboration with Hadoop. It performs very fast and integrates well with Hadoop (which is proven in practise by Facebook, as they utilize HBase for their messaging systems). Unfortunately we where not able to try our implementation on a serious cluster and benchmark the results.
Bibliography


[7] Randy H. Katz Yanpei Chen, Archana Ganapanhi. To compress or not to compress - compute vs. io tradeoffs for map reduce energy efficiency.


Chapter 9

Acknowledgements

The Radboud University, where this experiment took place. I would like to thank the University to supply the computers and accommodation to facilitate this research.

My supervisor, prof.dr.ir. Th.P. van der Weide, thanks you for your advice, input and guidance to keep on track. You’ve learned me a lot during my research and always took the time to deliberate on the subject, where you found the time is still a riddle to me. You made the this research a enjoyable and a learning experience. In retrospection I think I couldn’t have opted for a more-suited person to supervise the research than you.

To all my friends and fellow students at the University, Hoogeschool Arnhem en Nijmegen and colleges at work, Laurens Koot, Bart Leusink, Ashwien Rampersad, Menno van Wieringen, Edwin van der Graaf, Bard Duijs, Jan Michels Eamonn Cassidy, Giel Janssen Lok, Daan Schraaven, Robbin Janssen, Marc Bitter, Niels van der Weide, Bart Nikkelen, Niels Mauel, Lock Blonk, Marco Frenken, Raoel Roolofs, Bart Kusters for all your comments, (implicit) input, coaching and support.

Thanks to my family, Paul Gijsen, Mariet Rutten, Paula Gijsen and Lidwien Gijsen, for their everlasting support.
Chapter 10

appendix

In this section, we’ve listed the most important parts of the system, but it is to much to list all the code. Feel free to contact me for an copy of the complete code.

10.1 Data import

```java
/**
 * Wikipedia Data importer
 * Vincent Gijsen, 20-6-2012
 */
package nl.vincentgijsen.edu.util;

import java.io.IOException;
import java.util.regex.*;
import javax.xml.parsers.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.xml.sax.*;

public class ImportWiki {
    public static String infile = "";
    public static String outfile = "";
    public static int counter = 0;
    public static int fileSizeCounter = 0;
    public static final int maxFileSize = 5368709;
    private static Configuration conf ;
    private static FileSystem dfs ;
    private static Path outPath ;
    private static FSDataOutputStream outFileStream ;

    public static void main(String[] args) {
        System.out.println("main loading");
        System.out.println("Arg 1 = input file");
        System.out.println("Arg 2 = output dir");
        if (args[0] == null || args[0].isEmpty()) {
            System.out.println("Arg1 not specified..");
            System.exit(1);
        } else {
            infile = args[0];
            outfile = args[1];
        }

        for(String a : args) {
            System.out.println("arg: " + a);
        }
        try {
            conf = new Configuration();
            dfs = FileSystem.get(conf);
            outPath = new Path(outfile + "/" + counter);
```
if (dfs.exists(outPath)) {
    System.err.println("ERROR, output dir exists!");
    System.exit(1);
}

outFileStream = dfs.create(outPath);
if (null == outFileStream) {
    System.err.println("Error: writer = null;");
    System.exit(1);
}
}

import Wiki i = new ImportWiki();
System.out.println("Loader initialised");

public ImportWiki () {
    try { parseData () ;} catch ( Exception ex) {ex.printStackTrace(); System.exit(1) ;}
}

public void fileWriter(String in) {
    int length = in.length();
}

public void parseData () {
    try {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();
        DefaultHandler handler = new DefaultHandler () {
            boolean isPage = false;
            boolean isTitle = false;
            boolean isText = false;
            boolean isId = false;
            String pageN = "";
            String tileN = "";
            String text = new StringBuilder ();
            int localCnt = 0;

            public void startElement (String uri, String localName, String qName, Attributes attributes) throws SAXException {
                // System.out.println("Start Element: " + qName);
                if (qName.equals("page")) {
                    isPage = true;
                    localCnt++;
                }
                if (qName.equals("title")) { isTitle = true;}
                if (qName.equals("text")) { isText = true;}
            }

            public void endElement (String uri, String localName, String qName) throws SAXException {
                if (qName.equals("page")) {
                    isPage = false;
                    DumpRecord();
                }
                if (qName.equals("title")) { isTitle = false;}
                if (qName.equals("text")) { isText = false; }
            }

            public void characters (char ch[], int start, int length) throws SAXException {
                if (isTitle) {
                    tileN = localCnt + ";" + new String(ch, start, length).replace(";","_" forg);  
                    localCnt++;
                }
                if (isText) { text.append(sanitizer(new String(ch, start, length)));
            }
            }

        public void DumpRecord () {  

    }
```java
try {
    String record = tileN + ":" + text.toString().replaceAll("\\s*", " ") + "\n";
    // check if file approximates 64 mb
    if ((fileSizeCounter + record.length()) < maxFileSize) {
        fileSizeCounter = fileSizeCounter + record.length();
        outFileStream.writeBytes(record);
    } else {
        counter ++;
        System.out.println("Opening new File to write to: "+ outPath + "/" + counter);
        outFileStream.close();
        outPath = new Path(outFile + "/" + counter);
        outFileStream = dfs.create(outPath);
        // reset de file - size - counter
        fileSizeCounter = record.length();
        outFileStream.writeBytes(record);
    }
    if ((localCnt % 1000) == 0) {
        System.out.println(" wrote: " + tileN + " size: " + record.length());
    }
    tileN = "";
    // maak text leeg;
    text = new StringBuilder();
} catch (Exception e) {
    e.printStackTrace();
}

public String sanitizer(String in) {
    StringBuilder s = new StringBuilder();
    // REGEX to clean WikiPedia Codes
    final String r = "\{\{.*\}\} |\[\[ Image : |\[\[ File : |\[\[ File\]* |\[\[ File\]|thumb |& lt ;|/ ref | ref |& gt ;| http :.+? |& quot ;| name =.+? |\[ |\\] |! - -.* - -";
    final Pattern p = Pattern.compile(r);
    Matcher m = p.matcher(in);
    String out = m.replaceAll(" ");
    // Sanitizing all non readable chars
    for (int x = 0; x < out.length(); x++) {
        // voor de punt tm 0 -9
        char c = out.charAt(x);
        if (c >= 40 && c <= 58) {
            s.append(c);
        } else {
            // @ & A-Z
            if (c >= 63 && c <= 90) {
                s.append(out.charAt(x));
            } else {
                // a-z
                if (c >= 97 && c <= 122) {
                    s.append(out.charAt(x));
                } else {
                    if (c == 63) {
                        s.append(out.charAt(x));
                    } else {
                        s.append(" ");
                    }
                }
            }
        }
    }
    return s.toString();
}
```

Listing 10.1: Appendix: The Wikipedia XML-importer

```
// replace all double spaces with one.
return s.toString().replace("\\s+"," ");
```

Listing 10.2: Appendix: Distribute files across the cluster

```
echo "USAGE: copy input output"
for i in 1 2 3 4 5 6 7 8
do
SLAVE=ir-slave$i.cs.ru.nl
echo $SLAVE:
scp -r $1 vincent@$SLAVE:$2
done
```

Listing 10.3: Appendix: Execute commands on cluster nodes

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8
do
SLAVE=ir-slave$i.cs.ru.nl
echo Connecting to: $SLAVE
# ping -c 1 $SLAVE | grep transmi
ssh vincent@$SLAVE $1
done
```

10.3 Parser

```
/**
 * Vincent Gijsen
 * 28-5-2012
 * Master Thesis - Radboud University
 * Experimentation class to test Stanford-NLP Parser
 */
package nl.vincentgijsen.edu;
import java.util.*;
import edu.stanford.nlp.parser.lexparser.LexicalizedParser;
import edu.stanford.nlp.process.Morphology;
import edu.stanford.nlp.trees.*/;
public class TreeBuilder {
public final static String sentences[] = { "Jonas is buying paint in order to paint his garden house"};
public final static String grammar = "englishPCFG.caseless.ser.gz";
public final static String[] options = { "-maxLength", "80", 
"-retainTmpSubcategories" };
static LexicalizedParser lp = LexicalizedParser.loadModel(grammar, options);
static TreebankLanguagePack tlp = new PennTreebankLanguagePack();
static GrammaticalStructureFactory gsf = tlp.ggrammaticalStructureFactory();

// Stemmer
public static void main(String[] args) {
    List<String> miniIndex = new ArrayList<String>();
    int cnt = 0;
    long startTime = System.currentTimeMillis();
    for (String s : sentences) {
        Tree parse = lp.apply(s);
        GrammaticalStructure gs = gsf.newGrammaticalStructure(parse);
        List<TypedDependency> tdl = gs.typedDependenciesCCprocessed(true);
        for (TypedDependency t : tdl) {
            miniIndex.add(keyBuilder(t, cnt));
            cnt++;
        }
    }
    Collections.sort(miniIndex);
    for (String s : miniIndex) {
        System.out.println(s);
    }
    System.out.println("\nparse took " + (System.currentTimeMillis() - startTime) + "
milliseconds");
    System.out.println("\nStemmer Testing\n");
    String arr[] = { "hiking", "driving", "driver", "drivers" };
    stemmer(arr);
}

public static void stemmer(String input[]) {
    for (String st : input) {
        String s = morpher.stem(st);
        System.out.println("Stemming " + st + " to " + s);
    }
}

/**
* Keybuilder. Keybuilder builds keys to store in HBase. The function keeps
* the lexicographically ordering in account.
* @param TypedDependency the node of the parser
* @param id The location in the tree
* @return returns the formatted key
*/
public static String keyBuilder(TypedDependency d, int id) {
    String s = String.format("%s_%s_%d_keyId-%d", d.reln(),
        morpher.stem(d.dep()).nodeString(),
        morpher.stem(d.gov()).nodeString(), d.dep().index(), id);
    return s;
}

Listing 10.4: Appendix:The treebuilder

10.4 Indexer

package nl.vincentgijsen.edu;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class IndexerToHBase extends Configured implements Tool {

public final static String parserGrammar = "/home/vincent/scriptie/englishPCFG.caseless.ser.gz";

public static enum MATCH_COUNTER {
    Sentences, docs
};

public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

    // parser stuff
    static NaturalLanguageParser nlp = new NaturalLanguageParserImpl(parserGrammar);
    static Stemmer stm = new StemmerImpl();
    static KeyBuilder kb = new KeyBuilder();
    static List<TypedDependency> typedDependcyList = null;

    // hbase stuff
    static Configuration HBconf = null;
    static HTable table = null;
    static HTable documentTable = null;
    static IntWritable intWritable = new IntWritable(1);
    static HBaseAdmin admin;

    public Map() {
        System.err.println("Started constructor");
        HBconf = hbaseConfig();
        try {
            admin = new HBaseAdmin(HBconf);
            for (HTableDescriptor des : admin.listTables()) {
                System.err.println("Tables:: " + des.getNameAsString());
                table = new HTable(HBconf, CONS.HBaseTable);
                documentTable = new HTable(HBconf, CONS.HBaseTableDocs);
            }
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }

}
77     System.err.println("Done with constructor");
78 
79 }
80 }
81 static IntWritable one = new IntWritable(1);
82 static Text statsKey = new Text(CONS.HBaseStatTotal);
83 static MurmurHash hash = new MurmurHash();
84 static int counter = 0;
85 
86 public void map(LongWritable key, Text value, Context context)
87     throws IOException, InterruptedException {
88         try {
89             String line = value.toString();
90             // check for improper formatting.
91             if (line.split(";").length < 2)
92                 return;
93             String document = line.split(";")[0];
94             String body = line.split(";")[1];
95             // STORING the original in the database
96             Put docPut = new Put(Bytes.toBytes(document));
97             // write just id in the docs db
98             docPut.add(CONS.HBaseFam, CONS.HBaseColContent,
99                 Bytes.toBytes(body));
100             documentTable.put(docPut);
101             String sentences[] = body.split("\s\.");
102             int sentenceCounter = 0;
103             List<TypedDependency> totalList = new ArrayList<TypedDependency>();
104             List<Put> putList = new ArrayList<Put>();
105             for (String sentence : sentences) {
106                 if (sentence.length() > 10 && sentence.length() < 100) {
107                     context.getCounter(MATCH_COUNTER.Sentences)
108                         .increment(1);
109                     // The parser may/will throw errors if input is awkward.
110                     typedDependencyList = nlp.parseSentence(sentence);
111                     // add the found elements to the list of elements found
112                     // in the document, for later parsing.
113                     if (null != typedDependencyList) {
114                         totalList.addAll(typedDependencyList);
115                         for (TypedDependency td : typedDependencyList) {
116                             String keyString = kb.prep(td, sentenceCounter,
117                                 document);
118                             // prefix key with hash of key, for normal
119                             // distribution of keys
120                             Put put = new Put(Bytes.toBytes(keyString));
121                             // store document number
122                             // improves performance for non-critical inserts
123                             put.setWriteToWAL(false);
124                             put.add(CONS.HBaseFam, Bytes.toBytes(String
125                                 .format("doc", document)), document
126                                 .getBytes());
127                             putList.add(put);
128                         }
129         }
130             } else {
131                 typedDependencyList = null;
132                 sentenceCounter++;
133         }
for (TypedDependency dep : totalList) {
    String keyString = kb.prep(dep, sentenceCounter, document);
    double score = calcTf(dep, totalList);
    // Calculate the TF score
    Put putTf = new Put(Bytes.toBytes(keyString));
    putTf.add(CONS.HBaseFam, CONS.HBaseColTf, Bytes.toBytes(score));
    // store the calculated score to the map.
    putList.add(putTf);
}
// flush the map to the database
// write to HBASE the records
table.put(putList);
// populate a 'one' in the mapper-output, to count the total 
// documents
context.write(statsKey, one);
context.getCounter(MATCH_COUNTER.docs).increment(1);
context.progress();
if (counter == 0) {
    admin.split(CONS.HBaseTable);
    admin.split(CONS.HBaseTableDocs);
    counter = 1;
}
if (counter == 1000 || counter == 100) {
    if (counter == 100) {
        counter = 1;
        admin.compact(CONS.HBaseTable);
        admin.compact(CONS.HBaseTableDocs);
        // wait for 5 secs after put
        // Thread.sleep(5000);
    } else {
        admin.split(CONS.HBaseTable);
        Thread.sleep(5000);
    }
    counter ++;
} catch (Exception e) {
    System.err.println("Caught exception " + e);
    System.err.println("REsetting nlp");
    nlp = new NaturalLanguageParserImpl(parserGrammar);
}
/**
 * calcTf. Calculates the tf score
 */
public double calcTf(TypedDependency typeDepTerm,
    List<TypedDependency> terms) {
    int duplicateElements = 0;
    for (TypedDependency iterator : terms) {
        // check for gov overlap
        if (typeDepTerm.reln().getLongName().
equals(iterator.reln().getLongName())) {
            // check dep overlap
            if (typeDepTerm.dep().nodeString().
equals(iterator.dep().nodeString())) {
                if (typeDepTerm.gov().nodeString()
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    Configuration HBconf = null;
    HTable statsTable = null;

    public Reduce() {
        System.err.println("Started constructor of Reduce");
        HBconf = hbaseConfig();
        try {
            HBaseAdmin.checkHBaseAvailable(HBconf);
            statsTable = new HTable(HBconf, CONS.HBaseTableStats);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
        System.err.println("Done with constructor");
    }

    static IntWritable iterator;

    /*
     * The reduce function
     */
    @SuppressWarnings("unused")
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        long counter = 0;
        for (IntWritable iterator : values) {
            counter ++;
            // prevent timeouts due to one large key
            if ((counter % 1000) == 0) {
                context.setStatus("static");
            }
        }
        Put totalNumberOfDocumentsRecord = new Put(Bytes.toBytes(CONS.HBaseStatTotal));
        totalNumberOfDocumentsRecord.add(CONS.HBaseFam,
                                           Bytes.toBytes(CONS.HBaseStatTotal), Bytes.toBytes(counter));
        System.err.println("writing final score " + counter + " to hbase");
        System.err.println("writing to table: " + Bytes.toString(statsTable.getTableName()) + " for key: " + (key.toString()));
        statsTable.put(totalNumberOfDocumentsRecord);
    }

    /**
     * The main method. Called by the hadoop map-reduce job-runner.
     */
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new IndexerToHBase(), args);
        System.exit(res);
    }

    /* hbaseConfig(). This methods returns the configurationType used to connect
     */
public static Configuration hbaseConfig() {
  try {
    Configuration HBconf = HBaseConfiguration.create();
    HBconf.clear();
    HBconf.set("hbase.zookeeper.quorum", CONS.Hbase_Quorum);
    HBconf.set("hbase.master", CONS.Hbase_Master);
    HBconf.set("hbase.zookeeper.property.clientPort", CONS.Hbase_Port);
    HBaseAdmin.checkHBaseAvailable(HBconf);
    System.out.println("HBASE is running");
  } catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
  }
  return HBconf;
}

public int run(String[] args) throws Exception {
  Configuration HBconf = hbaseConfig();
  try {
    HBaseAdmin admin = new HBaseAdmin(HBconf);
    // create Data Table
    fixTable(admin, CONS.HBaseTable, CONS.HBaseFam);
    admin.createTable(createTable(CONS.HBaseTable, CONS.HBaseFam));
    // re-create StatsTable
    fixTable(admin, CONS.HBaseTableStats, CONS.HBaseFam);
    admin.createTable(createStatsTable(CONS.HBaseTableStats,
                                         CONS.HBaseFam));
    // re-create DocTable
    fixTable(admin, CONS.HBaseTableDocs, CONS.HBaseFam);
    admin.createTable(createDocTable(CONS.HBaseTableDocs,
                                       CONS.HBaseFam));
    // Necessarily for increment to function properly
    HTable statsTable = new HTable(HBconf, CONS.HBaseTableStats);
    Put initialCount = new Put(Bytes.toBytes(CONS.HBaseStatTotal)),
                                Bytes.toBytes(CONS.HBaseStatTotal));
    initialCount.add(CONS.HBaseFam, Bytes.toBytes(CONS.HBaseStatTotal),
                     Bytes.toBytes(0));
    statsTable.put(initialCount);
    statsTable.close();
  } catch (IOException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
  Job job = new Job();
  job.setJobName("Scriptie runner - " + args[2]);
  job.setJarByClass(IndexerToHBase.class);
  job.setMapperClass(IndexerToHBase.Map.class);
  job.setReducerClass(IndexerToHBase.Reduce.class);
  // input & output FILES
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  job.setOutputValueClass(IntWritable.class);
  job.setOutputKeyClass(Text.class);
job.setNumReduceTasks(1);
job.submit();
job.waitForCompletion(true);
return 0;
}

public void fixTable(HBaseAdmin adm, String tableName, byte[] columnFam)
throws IOException {
if (adm.tableExists(tableName)) {
    if (!adm.isTableDisabled(tableName))
        adm.disableTable(tableName);
    adm.deleteTable(tableName);
}
}

/**
 * createTable. Returns table format for the index
 *
 * @param name Name of the Table
 * @param colFam The name of the ColumnFamily to create
 * @return returns a HTableDescriptor
 */
public HTableDescriptor createTable(String name, byte[] colFam) {
    HTableDescriptor t = new HTableDescriptor(name);
    HColumnDescriptor col = new HColumnDescriptor(colFam);
    col.setMaxVersions(CONS.HBaseVersions);
    col.setBlockCacheEnabled(false);
    t.addFamily(col);
    t.setDeferredLogFlush(false);
    return t;
}

/**
 * createStatsTable. Returns tableLayout from the stats-table.
 *
 * @param name Name of the table.
 * @param colFam The name of the columnFamily to create
 * @return The HTableDescriptor to be inserted in to HbaseAdmin.
 */
public HTableDescriptor createStatsTable(String name, byte[] colFam) {
    HTableDescriptor t = new HTableDescriptor(name);
    HColumnDescriptor col = new HColumnDescriptor(colFam);
    col.setMaxVersions(CONS.HBaseVersions);
    t.addFamily(col);
    return t;
}

/**
 * createDocTable. Returns layout for the document store.
 *
 * @param name
 * @param colFam
 * @return
 */
public HTableDescriptor createDocTable(String name, byte[] colFam) {
    HTableDescriptor t = new HTableDescriptor(name);
    HColumnDescriptor col = new HColumnDescriptor(colFam);
    col.setMaxVersions(1);
    col.setCompactionCompressionType(Algorithm.GZ);
    col.setBlockCacheEnabled(false);
    col.setInMemory(false);
    t.addFamily(col);
    t.setDeferredLogFlush(false);
    return t;
}
10.5 Searcher

```java
package nl.vincentgijsen.edu;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import nl.vincentgijsen.edu.util.KeyBuilder;
import nl.vincentgijsen.edu.util.NaturalLanguageParser;
import nl.vincentgijsen.edu.util.Stemmer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.HBaseAdmin;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.io.hfile.Compression.Algorithm;
import org.apache.hadoop.hbase.regionserver.StoreFile.BloomType;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.util.ToolCmdLine;
import org.apache.hadoop.util.ToolRunner;
import edu.stanford.nlp.trees.TypedDependency;

public class IndexerToHBase extends Configured implements Tool {
    // Every node stores a local copy of the lexicon
    public final static String parserGrammar = "/home/vincent/scriptie/englishPCFG.caseless.ser.gz";

    public static enum MATCH_COUNTER {
        Sentences, docs
    }

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        // parser stuff
        static NaturalLanguageParser nlp = new NaturalLanguageParserImpl(parserGrammar);
        static Stemmer stm = new StemmerImpl();
        static KeyBuilder kb = new KeyBuilder();
        static List<TypedDependency> typedDependcyList = null;

        // hbase stuff
        static Configuration HBconf = null;
        static HTable table = null;
        static HTable documentTable = null;
        static IntWritable intWritable = new IntWritable(1);
```
static HBaseAdmin admin;

public Map()
{
    System.err.println("Started constructor");
    HBconf = hbaseConfig();
    try {
        admin = new HBaseAdmin(HBconf);
        for (HTableDescriptor des : admin.listTables()) {
            System.err.println("Tables:: " + des.getNameAsString());
        }
        table = new HTable(HBconf, CONS.HBaseTable);
documentTable = new HTable(HBconf, CONS.HBaseTableDocs);
    }
    catch (Exception e) {
        System.err.println(e.getMessage());
    }
    System.err.println("Done with constructor");
}

public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
    try {
        String line = value.toString();
        // check for improper formatting.
        if (line.split(";").length < 2)
            return;
        String document = line.split(";")[0];
        String body = line.split(";")[1];
        // STORING the original in the database
        Put docPut = new Put(Bytes.toBytes(document));
        // write just id in the docs db
        docPut.add(CONS.HBaseFam, CONS.HBaseColContent, Bytes.toBytes(body));
        documentTable.put(docPut);
        String sentences[] = body.split("\\.");
        int sentenceCounter = 0;
        List<TypedDependency> totalList = new ArrayList<TypedDependency>();
        List<Put> putlist = new ArrayList<Put>();
        for (String sentence : sentences) {
            if (sentence.length() > 10 && sentence.length() < 100) {
                context.getCounter(MATCH_COUNTER.Sentences).increment(1);
            }
            // The parser may/will throw errors if input is awkward.
            typedDependencyList = nlp.parseSentence(sentence);
            // add the found elements to the list of elements found
            // in the document, for later parsing.
            if (null != typedDependencyList) {
                totalList.addAll(typedDependencyList);
                for (TypedDependency td : typedDependencyList) {
                    String keyString = kb.prep(td, sentenceCounter, document);
                }
            }
            // prefix key with hash of key, for normal
            // distribution of keys
Put put = new Put(Bytes.toBytes(keyString));
// store document number
// improves performance for non-critical inserts
put.setWriteToWAL(false);
put.add(CONS.HBaseFam, Bytes.toBytes(String.format("doc", document)), document.getBytes());

putList.add(put);
}
}
typedDependencyList = null;
sentenceCounter++;
}

for (TypedDependency dep : totalList) {
    String keyString = kb.prep(dep, sentenceCounter, document);
    double score = calcTf(dep, totalList);
    Put putTf = new Put(Bytes.toBytes(keyString));
    putTf.add(CONS.HBaseFam, CONS.HBaseColTf, Bytes.toBytes(score));
    // store the calculated score to the map.
    putList.add(putTf);
}
// flush the map to the database
// write to HBASE the records
table.put(putList);

// populate a 'one' in the mapper-output, to count the total #
// documents
context.write(statsKey, one);
context.getCounter(MATCH_COUNTER.docs).increment(1);
context.progress();

if (counter == 0) {
    admin.split(CONS.HBaseTable);
    admin.split(CONS.HBaseTableDocs);
    counter = 1;
}

if (counter == 10000 || counter == 100) {
    if (counter == 100) {
        counter = 1;
        admin.compact(CONS.HBaseTable);
        admin.compact(CONS.HBaseTableDocs);
        // wait for 5 secs after put
        // Thread.sleep(5000);
    } else {
        admin.split(CONS.HBaseTable);
        Thread.sleep(5000);
    }
    counter++;
}
} catch (Exception e) {
    System.err.println("Caught exception " + e);
    System.err.println("REsetting nlp");
nlp = new NaturalLanguageParserImpl(parserGrammar);
}
}
public double calcTf(TypedDependency typeDepTerm, 
List<TypedDependency> terms) {
int duplicateElements = 0;

for (TypedDependency iterator : terms) {
    // check for gov overlap
    if (typeDepTerm.reln().getLongName()
        .equals(iterator.reln().getLongName())) {
        // check dep overlap
        if (typeDepTerm.dep().nodeString()
            .equals(iterator.dep().nodeString())) {
            if (typeDepTerm.gov().nodeString()
                .equals(iterator.gov().nodeString())) {
                // same node, increase counter
                duplicateElements ++;
            }
        }
    }
}
return (double) ((duplicateElements * 1.0) / terms.size());
}

public static class Reduce extends 
Reducer<Text, IntWritable, Text, IntWritable> {
    Configuration HBconf = null;
    HTable statsTable = null;

    public Reduce() {
        System.err.println("Started constructor of Reduce");
        HBconf = hbaseConfig();
        try {
            HBaseAdmin.checkHBaseAvailable(HBconf);
            statsTable = new HTable(HBconf, CONS.HBaseTableStats);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
        System.err.println("Done with constructor");
    }

    static IntWritable iterator;

    /*
    * The reduce function
    */
    @SuppressWarnings("unused")
    public void reduce(Text key, Iterable<IntWritable> values, 
        Context context) throws IOException, InterruptedException {
        long counter = 0;

        for (IntWritable iterator : values) {
            counter ++;
            // prevent timeouts due to one large key
            if ((counter % 1000) == 0) {
                context.setStatus("static");
            }
        }

        Put totalNumberOfDocumentsRecord = new Put(
            Bytes.toBytes(CONS.HBaseStatTotal));
        totalNumberOfDocumentsRecord.add(CONS.HBaseFam, 
            Bytes.toBytes(CONS.HBaseStatTotal), Bytes.toBytes(counter));
        System.err.println("writing final score " + counter + " to hbase");
        System.err.println("writing to table: "
            + Bytes.toString(statsTable.getTableName()) + " for key: "
            + (key.toString()));
        statsTable.put(totalNumberOfDocumentsRecord);
public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new IndexerToHBase(),
                           args);
    System.exit(res);
}

/**
 * hbaseConfig(). This methods returns the configurationType used to connect
 * to the HBASE instance.
 * @return returns the Configuration object.
 */

public static Configuration hbaseConfig() {
    Configuration HBconf = HBaseConfiguration.create();
    try {
        HBconf.clear();
        HBconf.set("hbase.zookeeper.quorum", CONS.Hbase_Quorum);
        HBconf.set("hbase.master", CONS.Hbase_Master);
        HBconf.set("hbase.zookeeper.property.clientPort", CONS.Hbase_Port);
        HBaseAdmin.checkHBaseAvailable(HBconf);
        System.out.println("HBASE is running");
        try (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        return HBconf;
    }
    /**
     * The Tools implementation. This methods configures the HADOOP framework.
     */
    public int run(String[] args) throws Exception {
        Configuration HBconf = hbaseConfig();
        try {
            HBaseAdmin admin = new HBaseAdmin(HBconf);
            // create Data Table
            fixTable(admin, CONS.HBaseTable, CONS.HBaseFam);
            admin.createTable(createTable(CONS.HBaseTable, CONS.HBaseFam));
            // re-create StatsTable
            fixTable(admin, CONS.HBaseTableStats, CONS.HBaseFam);
            admin.createTable(createStatsTable(CONS.HBaseTableStats,
                                                 CONS.HBaseFam));
            // re-create DocTable
            fixTable(admin, CONS.HBaseTableDocs, CONS.HBaseFam);
            admin.createTable(createDocTable(CONS.HBaseTableDocs, CONS.HBaseFam));
            // Necessarily for increment to function properly
            HTable statsTable = new HTable(HBconf, CONS.HBaseTableStats);
            initialCount.add(CONS.HBaseFam, Bytes.toBytes(CONS.HBaseStatTotal),
                             Bytes.toBytes(0));
            statsTable.put(initialCount);
            statsTable.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
Job job = new Job();
job.setJobName("Scriptie runner = " + args[2]);
job.setJarByClass(IndexerToHBase.class);
job.setMapperClass(IndexerToHBase.Map.class);
job.setReducerClass(IndexerToHBase.Reduce.class);
// input & output FILES
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setNumReduceTasks(1);
job.submit();
job.waitForCompletion(true);
return 0;
}

public void fixTable(HBaseAdmin adm, String tableName, byte[] columnFam) throws IOException {
if (adm.tableExists(tableName)) {
if (!adm.isTableDisabled(tableName)) {
adm.disableTable(tableName);
adm.deleteTable(tableName);
}
}
}

/**
 * createTable. Returns table format for the index
 * @param name Name of the Table
 * @param colFam The name of the ColumnFamily to create
 * @return returns a HTableDescriptor
 */
public HTableDescriptor createTable(String name, byte[] colFam) {
HTableDescriptor t = new HTableDescriptor(name);
HColumnDescriptor col = new HColumnDescriptor(colFam);
col.setMaxVersions(CONS.HBaseVersions);
col.setBloomFilterType(BloomType.ROW);
t.addFamily(col);
t.setDeferredLogFlush(false);
return t;
}

/**
 * createStatsTable. Returns tableLayout from the stats-table.
 * @param name Name of the table.
 * @param colFam The name of the columnFamily to create
 * @return The Htabledescriptor to be inserted in to HBaseAdmin.
 */
public HTableDescriptor createStatsTable(String name, byte[] colFam) {
HTableDescriptor t = new HTableDescriptor(name);
HColumnDescriptor col = new HColumnDescriptor(colFam);
col.setMaxVersions(CONS.HBaseVersions);
t.addFamily(col);
return t;
}

/**
 * createDocTable. Returns layout for the document store.
 * @param name
 * @param colFam
 * @return
 */
public HTableDescriptor createDocTable(String name, byte[] colFam) {
Listing 10.6: Appendix: The Servlet

10.6 Performance tuning

10.6.1 Hadoop

Listing 10.7: Appendix: Hadoop Environment Settings
Listing 10.8: Appendix: Hadoop Map-Reduce Settings

```xml
<configuration>
  <property>
    <name>mapred.system.dir</name>
    <value>/hadoop/mapred/system/</value>
  </property>
  <property>
    <name>mapred.local.dir</name>
    <value>/hadoop/mapred/temp/</value>
  </property>
</configuration>
```

Listing 10.9: Appendix: Hadoop HDFS Settings

```xml
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>/hadoop/dfs/namedir</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>/hadoop/dfs/datadir</value>
  </property>
  <property>
    <name>dfs.support.append</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.datanode.max.xcievers</name>
    <value>4096</value>
  </property>
</configuration>
```

10.6.2 HBase

```bash
export JAVA_HOME=/usr/local/jre1.7.0_03
export HBASE_HEAPSIZE=512
export HBASE_OPTS="-XX:NewSize=64m -XX:MaxNewSize=64m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC" 
export HBASE_MANAGES_ZK=true
```

Listing 10.10: Appendix: HBase Environment Settings

```xml
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://ir-master.cs.ru.nl:9000/hbase</value>
    <description>The directory shared by RegionServers.</description>
  </property>
  <property>
    <name>hbase.zookeeper.dns.nameserver</name>
    <value>131.174.224.4</value>
  </property>
  <property>
    <name>hbase.region.max.filesize</name>
    <value>32000000</value>
  </property>
  <property>
    <name>hbase.master</name>
    <value>ir-master.cs.ru.nl:60020</value>
  </property>
</configuration>
```
Listing 10.11: Appendix:HBase Core Settings