

Master Thesis

Dehyphenation of Words and Guessing Ligatures

Sumitra Magdalin Corraya

March 12, 2018

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik / Computer Science

Bearbeitungszeitraum

13.09.2017 - 13.03.2018

Gutachter

Prof. Dr. Hannah Bast

Prof. Dr. Peter Thiemann

Betreuer

Claudius Korzen

Contents

Zusammenfassung	1
Abstract	2
Declaration	3
Acknowledgments	4
1 Introduction	5
1.1 Motivation and Challenge	6
1.2 Approach	8
2 Related Work	10
3 Data Collection and Preparation	12
3.1 Count Frequency of the Words	12
3.2 Gather Data	13
4 Implementation	15
4.1 GNU Trove	16
4.2 MapDB	17
4.3 LMDB	19
4.4 Dehyphenation of Words	20
4.5 Guessing Ligatures	21
5 Experiments	24
Summary & Future Work	29
List of Abbreviations	30
List of Figures	31
List of Algorithms	32
Bibliography	33

Zusammenfassung

Die genaue Extraktion von Text aus PDF (z.B. um darin suchen zu können) kann sich als eine schwierige Aufgabe erweisen, da in einigen Fällen lange Wörter am Zeilenende in zwei mit einem Bindestrich verbundene Wörter aufgetrennt sein können. Darüberhinaus enthalten PDF-Dokumente Ligaturen (mehrere Zeichen, die in der PDF-Datei in ein einzelnes Zeichen übersetzt werden), nach denen nur schwer gesucht werden kann. In dieser Arbeit schlagen wir Lösungen für die oben genannten Probleme vor, d.h. wir entscheiden, ob ein Bindestrich (-) zwischen zwei Wörtern notwendig ist oder nicht, und schlagen eine Methode vor um Ligaturen zu erkennen in einem großen Wörterbuch. Bei beiden Lösungen bemühen wir uns um eine hohe Erkennungsgeschwindigkeit und Genauigkeit.

In PDF-Dateien können lange Wörter durch einen Bindestrich verbunden sein, wenn sie nicht in eine Zeile passen und dadurch in verschiedenen Zeilen erscheinen. Der Grund für diesen Bindestrich ist, dass die Wörter eine kombinierte Bedeutung haben oder dass es eine Beziehung zwischen den Wörtern gibt (auch bekannt als zusammengesetzte Wörter). Allerdings müssen nicht alle zusammengesetzten Wörter getrennt werden, was sich als problematisch erweisen kann. Diese Arbeit schlägt ein Verfahren vor, um zu entscheiden, wann ein Wort getrennt werden sollte. Dabei entscheiden wir über die Notwendigkeit eines Bindestrichs, indem wir das Wörterbuch verwenden. Unsere Experimente demonstrieren die hohe Effizienz und Genauigkeit unseres Ansatzes.

Bei der Erstellung von PDF-Dateien werden einige Kombinationen von Buchstaben (Ligaturen) zu einem einzelnen Zeichen übersetzt (z.B. werden die Ligaturen: ffi, ffl zu Sonderzeichen übersetzt). Dies kann beim Extrahieren des Textes aus dem PDF (z.B. für eine Suche) problematisch sein, da man dabei die einzelnen Zeichen einer Ligatur extrahieren möchte und nicht das entstandene Sonderzeichen. Diese Problematik wird durch die Tatsache, dass PDF verschiedene Schriften und Kodierungen unterstützt noch verschärft. Wir schlagen eine Methode vor, um Ligaturen aus einem Sonderzeichen zu "erraten". Dazu probieren wir verschiedene Ligaturen aus und suchen nach den entstehenden Wörtern in dem Wörterbuch. Schließlich diskutieren wir, was im Falle einer Mehrdeutigkeit bei dem eben geschilderten Nachschlagen einer Ligatur zu tun ist.

Abstract

The accurate extraction of text from PDF (e.g. in order to be able to search in it) can prove to be a difficult task as on some occasions long words at the end of the line are split into two words joined with a hyphen. Moreover, PDF contains ligatures (multiple characters, which translate to one/single character in the PDF) that can be hard to search for as well. In this thesis we propose solutions to the aforementioned problems, i.e. deciding if the hyphen (-) in the words is necessary or not, and proposing a method for guessing ligatures by looking up in a large dictionary. In both cases, we try to provide a fast response of time and good accuracy.

In PDF, long words can be split into two parts with a hyphen between them, when they do not fit in one line and as a result appear in different lines. The reason for the hyphen is to show that the words had a combined meaning or that there is a relationship between the words (also known as compound words). However, not all compound words need to be hyphenated which can prove to be problematic. This thesis proposes a procedure to decide when to dehyphenate the word. We decide upon the necessity of the hyphen by using the dictionary of words. Our approach provides efficiency and good accuracy, which we demonstrate with different experimental results.

During the making of PDF files, some combination of character (ligature) translates to a single character (for example the ligatures: "ffi", "ffl" are translated to some special characters). This can be problematic as when the text is extracted from the PDF, e.g. for searching, one wants to extract the individual characters within the word and not the special character. This issue is further aggravated by the fact that PDF has different fonts and encoding. We propose a method for guessing ligatures from such special characters. We do so, by trying different ligatures and searching the formed words in the dictionary. Finally, we also discuss what to do in case of an ambiguity when looking up the word.

Declaration

I hereby declare that this master thesis has been composed by me based on my own work, that I have not used any sources other than those specified, and that all passages which have been taken literally or meaningfully from published writings have been specified as such. Furthermore, I declare that this thesis has not been fully or partially presented for any other test.

Datum

Signature

Acknowledgments

First and above of all, I praise to God, the Almighty, for the strengths and his blessing in completing this thesis. Second, my special appreciation goes to Prof. Hannah Bast for a great opportunity to work at her Chair. Not forgotten, my appreciation to my co-supervisor, Claudius Korzen for his valuable time and knowledge regarding this topic. And his invaluable help of constructive comments and suggestions throughout the experiment and writing of this thesis.

I would like to thank Muneeb Shahid for proofreading this thesis, Moritz Freidank for the translation.

Sincere thanks to my beloved friends in Freiburg, for their kindness, moral support and refreshing time. My colleagues at Computer Science Department, for motivations and many small discussions.

My deepest gratitude to all my family in Bangladesh, especially for my mother Mrs. Gita Francisca Corraya and my father Provat Anthony Corraya, for their endless love, prayers and encouragement.

Last but not least, I am grateful my boss, Mr. Bruno Welsch, head of the IMTEK multimedia department, for his encouragement and patience when I had to take time off to study.

To everyone else who has helped me in any shape or from even with a simple smile on difficult days.

Thank you all for your encouragement and support!

1 Introduction

PDF is a popular computerized document format. The accurate extraction of the text from PDF document is an important, but difficult task, as sometimes long words at the end of the line are hyphenated, (where a hyphenated word is a word containing hyphen (-)). A hyphen is used when two or more words joined to form a new word (for example: ex-president, which means a person was a president).

Another reason is that PDF also contains ligatures, where a ligature is a typographical letter (see figure 1.1). A ligature occurs when two or more characters are joined into a single glyph (a visual representation of a character in a specific font and style) [Ber11]. For example, ligature "fi" in "Significant" is translated to a single glyph and this cannot be separated as "f" and "i" when copying text out of the PDF file (See figure 1.2).

The goals of this thesis are (1) decide, if a hyphen in a hyphenated word is mandatory or not and (2) resolving the individual characters of ligatures. In both goals we try to achieve high accuracy and fast response time.

$$\begin{array}{l} AE \rightarrow \text{Æ} \quad OE \rightarrow \text{Œ} \\ ae \rightarrow \text{æ} \quad oe \rightarrow \text{œ} \\ fi \rightarrow \text{fi} \quad ffi \rightarrow \text{ffi} \end{array}$$

Figure 1.1: Ligature with real meaning. Ligature "AE" or "ae": When "a" is followed by "e" in one word, it shows "a" and "e" two alphabets are a single glyph. Also, same for "f" and "i" or "f", "f" and "i". When "a" is followed by "e" and "o" is followed by "e" they both look like same glyph: see in the figure second line.

Mathematical formulæ are notoriously difficult to deal with for several reasons. Some of the characters might be drawn with vector operations, the vertical position varies much and is obviously significant. Outside of running text they might conceivably be stripped out, but inlined in the text they have to be dealt with. Since it is very difficult to make any sense of the extracted equations, the most important thing is not to break the segmentation of surrounding text.

Figure 1.2: Example for ligature: "Significant" an example with two alphabets "f" and "i", which are joined into a single glyph "fi". "Difficult" is another example with three alphabets "f", "f" and "i", which are joined into a single glyph "ffi". "formulae" also have ligature "ae", which is joined into a single glyph.

1.1 Motivation and Challenge

In PDF text extraction, dehyphenating words and decomposing ligatures into individual characters is mandatory as hyphenated words and/or words with ligatures can't be found using simple text search.

Consider the example that the word "digital" is hyphenated as "digi-tal," when we extract the PDF file, the word is exported into two words "digi-" and "tal" or "digi-tal" (which is with a hyphen), thus finding the original text "digital" is not easy. Other difficulties are deciding if a hyphen is mandatory or not.

One of the reason is a hyphen in some words is correct (in terms of the English language, which has so-called compound words containing mandatory a hyphen) (See figure 1.3). A compound word is defined as a word consisting of more than one word. When two or more words are attached to each other, and they make one long word, it is then referred to as a closed compound word, for example footpath, created from two nouns foot and path. Apart from closed compound words, there are open compound words, which are created in cases when the modifying adjective is used with its noun to create a new noun (e.g. dinner table), in this case space comes in between the adjective and the noun, so sometimes it can be hard to identify a compound word.

Hyphens are used in many compound words to show that the component words have a combined meaning (e.g. sugar-free) or that there is a relationship between the words that make up the compound: for example "rock-forming minerals", are minerals that form rocks. However, a hyphen does not need to be used in every type of the compound word, for example, phrasal verbs: when verbs are made up of a main verb and an adverb or preposition. "You should continue to build up your pension", here "build up" is a phrasal verb. On the other hand, if phrasal verb is made into a noun, then a hyphen can be used in compound words. For example: here was a build-up of traffic on the ring road. Also a hyphen is used in

In this paper, we show how to construct a high-quality benchmark of principally arbitrary size from parallel TeX and PDF data.

Figure 1.3: Example for mandatory (or not) a hyphen in the words: In case a compound word (e.g. high-quality) is hyphenated, we do not want to remove the hyphen if the hyphen is mandatory, but on the other hand, for not a compound word (e.g. bench-mark) we want to remove the hyphen and merge the syllables.

1. Compound adjectives: made up of a noun + an adjective (e.g. sugar-free), a noun + a participle (e.g. custom-built), or an adjective + a participle (e.g. good-looking) [Oxf].
2. Compound verbs: two nouns are combined into a verb (e.g. to ice-skate) [Oxf].
3. Compound nouns: such nouns can be written in one word (e.g. chatroom), two words (e.g. chat room) or in a hyphenated form (e.g. chat-room) [Oxf].
4. Compounds in which the base word is capitalized: e.g. pro-Freudiana [Ass]
5. Compounds in which the base word is number: e.g. post-1970 [Ass]

A hyphen can also be used where a word is to be divided at the end of a line of writing, but it should be split at a sensible position, so that the first part does not mislead the reader e.g. "hel-met" should be preferred over "he-lmet". And also the word should split before the noun, not after the noun. For example: "a long-term solution." here "long-term" is a hyphenated compound word because it is used in the sentence before a noun (e.g. solution). And in "This is not a good solution for the long term.", "long term" is an open compound word without a hyphen because it is in the sentence after the noun. Hyphenation rules allow to hyphenate 90% of English words properly [Sho16].

Without those rules a hyphens can be used when in a sentence a subject has been described. For example: "I saw a man-eating alligator" and "I saw a man eating alligator". These two sentences mean two different things. "I saw a man-eating alligator" is describing the alligator that eats men. On the other hand, "I saw a man eating alligator" is describing a man who is eating an alligator.

Apart from those rules, some compound word can be both with or without a hyphen, deciding which one is the right choice can be a challenging task (see figure 1.4(a)). The aforementioned scenarios lead to complexities when extracting text from the PDF.

Ligatures too, can be difficult to find, for example the word "define" from PDF might be converted to "de\$ne" (where we use the "\$" symbol to illustrate the ligature), as



Figure 1.4: (a) Dehyphenating the word is challenging: for example, the word "US-English" is a compound word and it is valid both with and without a hyphen (as shown in figure (a)). Both types of compound word can be found in English. (b) Ambiguities in word with ligature: "\$u\$y" is a word with ligatures. "\$" is the position for the ligature. A single word containing ligature can lead to multiple combination of different ligature resulting in multiple valid words (as shown in figure (b) as Output). In this case, defining a right word is challenging.

it contains the ligature "fi". Since there are 20 different ligatures in English, it can be quite ambiguous for a human to decipher "\$", as multiple ligature candidates are possible. For example: "fluffy" after conversion will look like "\$u\$y". "\$u\$y" is completely unrecognizable.

One reason for such cryptic outcomes is that when converting, it converts characters to glyphs using "poppler" for rendering, now when characters like "tt", "tf", "ti", "ff", "fi" are encountered, they are considered ligature. Thus, when one searches for "t t", the actual look up is done for "tt" and not "t t", resulting in no results/finds. And when it cannot find a glyph for "tt", it converts it as "\$" (the "\$" we have used as an illustration for the ligature) or Unicode or any other special character. Handling possible ambiguities is not an easy task either: two different ligatures can result in valid words, one then needs to decide upon the best one (See figure 1.4(b)).

1.2 Approach

In the first part of the thesis, we tackle the problem of dehyphenation: we first decide if the hyphenated word is mandatory or not. Our approach to solve this problem is to look up the given hyphenated word in a huge dictionary of words. While looking up the given hyphenated word in the dictionary it is possible that the word is found both with and without the hyphen (for example: sub-tube or subtube), in this case we look up the frequency (refers to the number of times that a word appears in the given text) of the words.

In the second part of the thesis, we discuss guessing ligatures (mean to determine the individual characters in ligatures), we do this in a "trial and error" style. In principle, there is only a finite number of possible ligatures. We could try one after the other, and check, if the resulting word is valid by looking it up in the dictionary. However, ambiguities can be found in a word with ligature. In this case, to decide the right

word with ligature, our idea is to look for the frequency of the words, we choose the word with the highest frequency. For example: for the word "\$u\$y" ambiguities mean: different ligature can be placed into the position "\$" and word with ligature could be "stuffy" and "fluffy". If the word "fluffy" has higher frequency than the word "stuffy", then "fluffy" is the output.

In both part of our thesis we also want to provide efficiency and good accuracy by using our approaches, which we demonstrate with different experiment results.

2 Related Work

Text Extraction is a process for generating meaningful text (do not contain special character for example: se\$-Sat) from electronic documents such as PDF. For more clarification the aim of text extraction is to be able to characterize and explain the text how human understand language. The complexity of the human/natural language (for example: English) can make it very difficult to access the information in that text. One of the reason is English has few grammatical rules (for example compound words with a hyphen or without a hyphen) and the amount of text that is available in electronic form is truly staggering, and is increasing every day. Considering those reasons working with human/natural language data is not solved; natural/human language is primarily hard because it is messy [Bro17]. However, there are some areas (e.g., Machine learning) interested in working with text extraction. Machine Learning is an area of Artificial Intelligence that is a set of statistical techniques for problem solving. Based on state of the art machine learning techniques there are some tools and open-source systems, such as GROBID, CERMINE[TSF⁺15].

GROBID performs reliable bibliographic data extractions from scholar articles combined with multi-level term extractions [Lop09]. It is a tool for analyzing technical and scientific documents, focusing on automatic bibliographical data extraction and structure recognition [PL10]. It is able to handle ligatures and hyphenated words.

CERMINE is a comprehensive open-source system for extracting structured meta-data from scientific articles. It is well known for good performance of the key process steps and the entire meta-data extraction process, with the overall average score of 77.5 % [TSF⁺15]. CERMINE can extract meta-data, full text and parsed references from a PDF file. It is also able to handle hyphenated words.

Machine learning-based approaches are far more popular. However, machine learning approaches are usually expensive in terms of run-time of the extraction processes [Fri10].

There are some papers published based on Text Extraction. In those paper's publishers proposes a different kind of method or system: PDFtotext [dav11], LA-PDFText [CRB], PDFExtract [ORBR], Icecite [BK17], OCR++ [Muk16].

PDFtotext is a generally used approach to extract text from PDF files. It reads the PDF file, and writes a text file. It treats the entire document as one string, outputs Unicode(UTF-8) data [dav11]. So ligatures represented as a single character in Unicode which will appear strangely in output.

Afterward LA-PDFText (system to facilitate accurate extraction of text from PDF files of research articles for use in text mining applications [CRB]) introduced and compared with PDF2Text system (in 91% of the cases LA-PDFText outperforms PDF2Text ($p < 0.001$)). In the experiment by using Needleman-Wunsch algorithm found that the text extracted via LA-PDFText have higher accuracy, but still contain classification errors.

PDFExtract is a tool and library that can extract various areas of text from a PDF. It handles ligature's and hyphenated words. However, only makes available a limited range of logical layout analysis functionality because of its very comprehensive geometric layout analysis.

Icecite automatically extracts, with accuracy over 95%, bibliographic meta-data [BK]. It handles all the hyphenated words as normal hyphenated words, but remove them mistakenly, if also it is important for compound words.

OCR++ is an open-source framework designed for a variety of information extraction tasks with accuracy (around 50% improvement) and processing time (around 52% improvement) from scholarly articles [Muk16], completely written in python. The framework takes a PDF article as input, first convert the PDF to an XML format then processes the XML file to extract useful information. Even though the XML file consists of rich meta-data, OCR++ suffers from common error: end-of-line hyphenation problem.

3 Data Collection and Preparation

Data collection is an extremely important part of this thesis. Without the right data collection (data which are meaningful in terms of Oxford dictionary (is the accepted authority on the English language, providing an unsurpassed guide to the meaning from across the English-speaking world [Oxf18].)) it is difficult to take decisions that our approach is right or wrong. Inaccurate data (e.g. "aaaabro", "10pc") collection may lead to the wrong result, or which is invalid and affect the expectation of the result, in the other word it will be impossible to work on the approach of the thesis. In this chapter, we are going to discuss the techniques or process of data collection for our thesis.

There are different kinds of techniques for data collection. We are going to collect data from diverse sources of documents. One diverse source we used where has around 40,515,568 words (without special character, i.e. "-" and numbers, i.e. 1,2). These words are extracted from PDF files of the digital library <https://arXiv.org/> (arXiv is a repository of electronics pre-prints approved for publication after moderation that consists of scientific papers in the fields of computer science, which can be accessed online). Another source we used is ClueWeb (The ClueWeb is a data-set, was created to support research on information retrieval and related human language technologies. It consists of billions of web pages in different languages), that contain around 34,702,294 words (without special character, i.e. "-" and numbers, i.e. 1,2).

3.1 Count Frequency of the Words

Our approach is based on the frequency of the words for solving the complicate situation and ambiguities of our thesis. Our data collection has to account for that and will contain words with their frequency.

As we are collecting our words with their frequency from the sources of documents which contains lots of text files. We take a bunch of text files as input. After getting the files we read every file once. While reading every single file we read line by line and convert every single line into array of words and counts the words occurrences (how many times they occur in the file). This counting is the frequency of the words. However, because of all the text files were extracted by using PDFtotext, it is possible that the text files are full of special characters and symbols, in particular formulas and mathematical symbols. So the words can read with a special character or digit (0-9). In example, words can be found like "1480e-02". And also as this

file have hyphenated words, one reason is PDF files break one single word into two words and put a hyphens in the first part of the word and then, put the second part of the word in another line. So inappropriate words can be found, which is not a proper compound or a hyphenated word because we are counting the words line by line. For example, the proper word is "q-maps" but while reading files, it can be found "q-" or "-maps". In those cases, we are not accepting this kind of words, before counting the frequency of that specific word we are checking is it a correct format of a word or a proper word without digit and a special character (See Algorithm 1).

Algorithm 1: The procedure for counting frequency of the words.

Input: A list of files
Output: Words and frequency of the words

```
1 for read every single files in folder do
2   while read every single lines in file do
3     words[] = split the line;
4     for count in words[] do
5       key = words[counter];
6       if key contain only alphabets or only alphabets with a hyphen)
          hashMap.get(key) == null then
7         | hashMap.put(key, 1)
8       else
9         | value = hashMap.get(key)
10        | value++;
11        | hashMap.put(key,value)
12      end
13    end
14  end
15 end
```

3.2 Gather Data

After reading a number of text files, all the data we gathered into one file (final combined text file), which will make the files easier to handle, the data faster to read through and more efficient to search. While combining all the data files we also concentrate that in the final combined text file, we do not have the same words several times. For this reason, we first put the words with the frequency of our final combined text file in the database, words as key and frequency as value. Before putting the data into the database, we also checked whether the final text file has data or not. And then we continue reading the other text files while comparing the words to see if they already in the database or not. If the words already exist in the database, then we do not put it into the database and in our main text file (where

we are gathering all the words with frequency). And if the word does not exist in the database, then we do on the other way around, meaning we put the word and the frequency in the database and also in the final combined text file (See Algorithm 2).

We collected 220,870 KB text file, which contains 16,820,541 words with frequency. Out of 16,820,541 words 11,795,033 words do not consist a hyphen and 4,987,677 words consist with hyphen. In the text file also have 4,546,311 words with ligatures.

Algorithm 2: The procedure for combining all the text files.

Input: A list of text files and final combined text file

Output: A text file

```
1 while read final combined text file do
2   | If the file has data put them in the database.
3 end
4 for read list of text files do
5   | Do the procedure for counting frequency of the words;
6   | if if the database do not have word then
7     | Put the word and frequency of the word in the database;
8     | Put the word and frequency of the word in final combined text file;
9   | end
10 end
```

4 Implementation

In this chapter, we are going to discuss how we implemented our solutions for the two different problems (dehyphenation of words and guessing ligatures) from pdf extraction in our thesis.

After making the collection of words in the text file, the first task for our thesis experiment is making an efficient database. The database is an organized collection of data, which means database store information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). For this thesis, we want to treat the data in our databases as objects. An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. For example, in our daily lives we use human languages to communicate with others. One example of a human language is English. The English language contains words. Every single word has different meanings. Each word is a small object that fits together with other small objects in predefined ways to create other larger objects. Objects are key to understanding object-oriented technology. An object-oriented database is organized around objects rather than actions, and data rather than logic. That's why we are using an object-oriented programming language: Java. Java Database Connectivity can access any kind of tabular data, especially data stored in a Relational Database. Java Database Connectivity helps manage these three programming activities: connect to a database, send queries and update statements to the database, retrieve and process the results received from the database in answer to your query. Java Database Connectivity API is a Java API. An API is a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service. An API may be for a web-based system, operating system, database system, computer hardware or software library.

So the first question comes to our mind why not to keep using well-known JDK collections? The answer is performance and memory consumption (See the comparison between Java and GNU Trove library in chapter5: experiments results). We decide to use a third-party library to increase our performance.

In computer science, a library is a collection of non-volatile resources used by computer programs, often to develop software. These may include configuration data, documentation, help data, message templates, pre-written code, classes, values or type specifications. For our thesis, we are trying three different third-party

libraries (GNU Trove, LMDB, MapDB) which support Java. A third-party software (which is using a third-party library) component is a reusable software component developed to be either freely distributed or sold by an entity other than the original vendor of the development platform. The reason behind using third-party libraries was to accelerate the development process.

4.1 GNU Trove

GNU Trove is a third-party library that contains a set of primitive collection and supports the use of custom hashing strategies. One of the reasons for choosing GNU Trove is better performance and memory consumption than JDK collection [Vor14]. GNU Trove does not use any `Java.lang.Number` sub-classes internally, so you do not have to pay for boxing/unboxing each time you want to pass/query a primitive value to/from the collection. Besides, you do not have to waste memory on the boxed numbers and reference to them. For example, if you want to store an Integer in JDK map, you need 4 bytes for a reference (or 8 bytes on huge heaps) and 16 bytes for an Integer instance. GNU Trove, on the other hand, uses just 4 bytes to store an int [Vor14]. For each key-value pair GNU Trove does not create `Map.Entry` unlike `Java.util.HashMap`. The GNU Trove maps/sets use open addressing instead of the chaining approach taken by the JDK hash tables. This eliminates the need to create `Map.Entry` wrapper objects for every item in a table and so reduces the O (big-oh) in the performance of the hash table algorithm. The size of the tables used in GNU Trove's maps/sets is always a prime number, improving the probability of an optimal distribution of entries across the table, and so reducing the likelihood of performance-degrading collisions[jim17]. GNU Trove's maps and sets support the use of custom hashing strategies, which allow to tuning collections based on characteristics of the input data. This feature also allows you to define hash functions when it is not feasible to override `Object.hashCode()`. For example, the `java.lang.String` class is final, and its implementation of `hashCode()` takes $O(n)$ time to complete. GNU Trove does not have any dependencies, so for using this third-party library we need to import only one package for Java.

GNU Trove provides "free" (as in "free speech" and "free beer"), fast, lightweight implementations of the `Java.util` Collections API; also provide the same collections support for primitive types, whenever possible [jim17]. If there is a large array list/set/map with keys or values that could be a primitive type, it is worth replacing it with GNU Trove collection. If there are some maps from a primitive type of a primitive type, it is especially worth to replace them. Unfortunately from inspection, it looks like GNU Trove has been just a library of collections for primitive types-it's not like it is meant to be adding a lot of functionality over the normal collections in the JDK [Ske09]. On each startup GNU Trove load data into memory, performs calculations and exits. For loading data into memory, GNU Trove is taking extra runtime, which is making difficulties to get fast response time. Our approach is to

solve the two problems (as explained in the introduction) with fast response time. So, we want to save our time by not loading data into memory every startup. If we save our data as a dictionary in the database on the first startup, then afterward for next start-ups we do not need to load all the data for making database. And MapDB and LMDB allow doing that.

4.2 MapDB

MapDB is specifically designed for the Java developer. MapDB provides a reliable, full-featured and "tune-able" database engine using the Java Collections API. In MapDB there is no change in syntax from typical Java coding, other than a brief initialization syntax and transaction management. A developer can literally transform memory-limited maps into a high-speed persistent store in seconds [Isaun]. It allows developers to control exactly what type of internal structure is needed for a given database, and what the actual data structure looks like from the top-level Collections API. There are three tiers in MapDB: Collections API, Engine (where the records in a database (including internal structure, concurrency control, transaction's semantics) are controlled), Volume (is the physical storage layer (e.g. on-disk or in-memory)) [Isaay].

The brief history of MapDB is: Prior to MapDB, Jan Kotek (the primary MapDB developer) supported various versions of the JDBM projects. JDBM itself was a Java port of UNIX DBM and GDBM, C-language databases that support hash-based key-value stores on disk. Through this experience, Jan Kotek saw how he could greatly improve and expand the architecture, and created MapDB as a totally new implementation. His experience paid off, with MapDB offering ease-of-use, an agile approach to the database structure, transaction support, concurrency, and very impressive performance [Isaay]. MapDB is an open-source (Apache 2.0 licensed), embedded Java database engine and collection framework, it provides Maps, Sets, Lists, Queues, Bitmaps with range queries, expiration, compression, off-heap storage and streaming [Map]. It has 7 compile dependencies such as Eclipse Collections, Guava, Kotlin library and some other libraries (See Figure 4.1 for a full list of dependencies) and 4 test dependencies (See Figure 4.2).

MapDB is flexible and simple. To get better performance, it is better to use MapDB in 64-bit operating system. Collection from MapDB (for example HashMap) can be stored in memory or in a file.

```
DB db = DBMaker.memoryDB().make();
ConcurrentMap map = db.hashMap("map").make();
or
DB db = DBMaker.fileDB("file.db").make();
ConcurrentMap map = db.hashMap("map").make();
```








Category/License	Group / Artifact	
<div style="background-color: red; color: white; padding: 2px;">Core Utils</div> <div style="background-color: blue; color: white; padding: 2px;">Apache 2.0</div>		com.google.guava » guava
<div style="background-color: red; color: white; padding: 2px;">Hashing</div> <div style="background-color: blue; color: white; padding: 2px;">Apache 2.0</div>		net.jpountz.lz4 » lz4
<div style="background-color: red; color: white; padding: 2px;">Collections</div> <div style="background-color: blue; color: white; padding: 2px;">EDL 1.0 EPL 1.0</div>		org.eclipse.collections » eclipse-collections-api
<div style="background-color: blue; color: white; padding: 2px;">EDL 1.0 EPL 1.0</div>		org.eclipse.collections » eclipse-collections
<div style="background-color: blue; color: white; padding: 2px;">EDL 1.0 EPL 1.0</div>		org.eclipse.collections » eclipse-collections-forkjoin
<div style="background-color: red; color: white; padding: 2px;">JVM Languages</div> <div style="background-color: blue; color: white; padding: 2px;">Apache 2.0</div>		org.jetbrains.kotlin » kotlin-stdlib
<div style="background-color: blue; color: white; padding: 2px;">Apache 2.0</div>		org.mapdb » elsa

Figure 4.1: MAPDB compile dependencies [MDB]

To make it faster we can also use serializer, which indicates the type of key and value. Without serializer definition MapDB will use slower generic serialization.

```
DB db = DBMaker.fileDB("file.db").make();
HTreeMap<String, Long> map = db.hashMap("name_of_map")
    .keySerializer(Serializer.STRING)
    .valueSerializer(Serializer.LONG)
    .create();
```

MapDB utilizes some of the advanced Java Collections variants, such as ConcurrentNavigableMap. With this type of Map, you can go beyond simple key-value semantics, as it is also a sorted Map allows you to access data in order, and find values near a key. MapDB have different Maps system to store data, for example HTreeMap, BTreeMap and Sorted Table Map. For MapDB: HTreeMap provide HashMap and HashSet collections, BTreeMap provides TreeMap and TreeSet, SortedTableMap provides Sorted String Tables. However, SortedTableMap is read-only and does not support updates. So the solution to update is creating a new Map with Data Pump. If we compare HTreeMap with BTreeMap, then BTreeMap is better for smaller keys, such as numbers and short strings. BTreeMap also provides a solution for a multiple or composite key. We can use Tuple2Serializer(), SerializerArrayTuple() or





Category/License	Group / Artifact	
<div style="display: flex; flex-direction: column; align-items: center;"> <div style="background-color: red; color: white; padding: 2px 5px; font-weight: bold;">Testing</div> <div style="background-color: blue; color: white; padding: 2px 5px; font-weight: bold;">EPL 1.0</div> </div>		junit » junit
<div style="display: flex; flex-direction: column; align-items: center;"> <div style="background-color: red; color: white; padding: 2px 5px; font-weight: bold;">Assertion</div> <div style="background-color: blue; color: white; padding: 2px 5px; font-weight: bold;">Apache 2.0</div> </div>		org.easytesting » fest-assert
<div style="display: flex; flex-direction: column; align-items: center;"> <div style="background-color: red; color: white; padding: 2px 5px; font-weight: bold;">Reflection</div> <div style="background-color: blue; color: white; padding: 2px 5px; font-weight: bold;">Apache 2.0</div> </div>		org.easytesting » fest-reflect
<div style="display: flex; justify-content: space-around; width: 100%;"> <div style="background-color: blue; color: white; padding: 2px 5px; font-weight: bold;">EDL 1.0</div> <div style="background-color: blue; color: white; padding: 2px 5px; font-weight: bold;">EPL 1.0</div> </div>		org.eclipse.collections » eclipse-collections-testutils

Figure 4.2: MAPDB test dependencies [MDB]

SerializerArrayDelta() methods. However multiple values with a single key (which called Multimaps) does not work in MapDB. For example, Map<Key,List<Value> > can be found in Guava or in Eclipse Collections. Closing the DB: DB.close() is one of the important tasks for MapDB. Because of file corruption MapDB offers WAL. And if WAL is disabled, then MapDB detects an unclean shutdown and will refuse to open such corrupted storage. Because of this reason there is strong chance to lose all of the data. So we decide, to go on with LMDB.

4.3 LMDB

LMDB library is designed around virtual memory, MVCC and SLS concepts. It is implemented in modern Linux, UNIX, and Windows operating systems. LMDB makes the entire database file appear in memory. The entire database is exposed in a memory map, and all data fetches return data directly from the mapped memory. It gives a Java interface, which is a fast key-value storage library. When an LMDB database is initially created, it occupies space in virtual memory as is needed to accommodate the data structure [Mat16]. As more entries are starting to add then more memory is needed, so the mapped file also becomes bigger. However, when entries deleted from the database, the space it used is not returned to the operating system. Which makes the library expensive and also make the systems overall performance slowdown. The good thing is the memory it is not returned to the operating system, afterward it is used for storing new entries. LMDB create/open database as followed

```
Env env = new Env();
env.setMapSize(5 * 1024 * 1024 * 1024);
env.open(System.getProperty("user.dir"));
Database db = env.openDatabase();
```

LMDB also have dependencies, such as `HawtJNI`. `HawtJNI` produces the JNI code, which needed to implement Java native methods `LMDBJNI`. For `LMDBJNI` two packages are recommended to import: `org.fusesource.lmdbjni` and `org.fusesource.lmdbjni.Constants`. LMDB is a Btree-based database management library, modeled loosely on the BerkeleyDB API. The library is extremely simple because it requires no page caching layer of its own, and it is an extremely high performance and memory-efficient [HC15]. LMDB offers Transactions (full ACID semantics), Ordered keys (enabling very fast cursor-based iteration), Memory-mapped files (enabling optimal OS-level memory management), Zero-copy design (no serialization or memory copy overhead), Configuration-free (no need to "tune" it to your storage), Instant crash recovery (no logs, journals or other complexity), Minimal file handle consumption (just one data file; not 100,000's like some stores), Freedom from application-side data caching (memory-mapped files are more efficient) [LMD].

After making dictionaries using both LMDB and MAPDB, we concentrated on how to find a hyphen in a word is necessary or not and how to guess ligature.

4.4 Dehyphenation of Words

First, for every single input, we check that the word contains a hyphen (-). If the word contains hyphens, then we follow two steps. In the first step, we check if the word with a hyphen (-) is available in the dictionary (contain words and their frequency) or not. If the word is available in the dictionary, we get the frequency of these word from the dictionary. Then we save the word and the frequency of these word in a result array. And on the second step, we reduce the hyphen (-) from the input word to find the input word as a compound word without space and hyphen in the dictionary and then we search the word without a hyphen is available in the dictionary or not. If the word available, we get the frequency of these word from the dictionary. We save the word and the frequency of these word in the result array. Then, we check the frequency of the word to take our decision that if this hyphen is necessary or not. If the frequency of a hyphenated word is higher than without hyphen word, then we give output "true", which means hyphen is necessary for the input word. And if the frequency without a hyphenated word (which we say a compound word) is higher than a hyphenated word, we give output "false", which means hyphen is not mandatory for the input word. From searching if we only found a hyphenated word or compound word we do not check frequency. If only a hyphenated word is found, then an output is set to "true", otherwise "false" if found as a compound word (see Algorithm 3). The important part is if both of

them (with a hyphen and without a hyphen) are not found in the database, then we give output "false" because in this case the input word is not a compound word (for example struc-ture).

Algorithm 3: The procedure for making the decision of hyphenated words.

```
Input: A list of words
Output: Hyphen is necessary for word or not
1 for read every single inputWord in list do
2   | input word contain hyphen (-)
3   | if Input word found in DB then
4   |   | frequency = Get the frequency of the input word from DB;
5   |   | result[] = input word, frequency;
6   |   | newInput = reduce hyphen(-) from input word;
7   |   | if newInput found in DB then
8   |   |   | frequency = Get the frequency of the newInput word from DB;
9   |   |   | result[] = newInput, frequency;
10  |   |   | Check the higher frequency in result[];
11  |   |   | output: hyphen necessary if input word (with hyphen) have higher
12  |   |   | frequency or
13  |   |   | output: hyphen not necessary if input word (without hyphen) have
14  |   |   | higher frequency;
15  |   | else
16  |   |   | output: hyphen necessary;
17  |   | end
18  | else
19  |   | newInput = reduce hyphen(-) from input word;
20  |   | if newInput found in DB then
21  |   |   | output: hyphen not necessary;
22  |   |   | else
23  |   |   |   | output: hyphen not necessary;
24  |   |   |   | end
25  |   | end
26 end
```

4.5 Guessing Ligatures

Ligatures are letters that have been mashed together as one character. We are going to discuss how we decided the correct ligatures for a word. The ligatures used most often are fi and fl, accompanied by their friends ff, ffi, and ffl. However, there is also some other ligature often used, so we are using 20 kinds of different ligatures for our implementation. They are aa, ae, ao, au, av, ay, et, ff, ffi, ffl, fi, fl, oe, oo, fs, st,

ft, tz, ue, vy. As for the ligatures, pdf shows a special character, hence we use the English dollar sign (\$) for the ligatures. At the first step we try to find out in the input, there is any dollar available or not. If the input contains a dollar sign, then we put every single ligature combination one by one instead of dollars and make a word and try to find out that data is available in the database or not. If it is available, we keep the word, but a complicated case occurs when there are multiple dollar signs. For two dollars (when one word contains two dollars) this induces 2^n combinations. So for 20 ligatures, we are using 2^{20} combinations for one word (See Algorithm 4).

Algorithm 4: The procedure for taking the decision of ligatures for the words.

Input: Word with "\$"
Output: correct meaningful word

```

1 for read inputWord do
2   if input word contain special character. e.g. $ then
3     for read one by one ligatures in list do
4       updatedInput = inputWord.replacefirst(, ligaturefromlist)
5       if updatedInput have more then
6         for read one by one ligatures in list do
7           updateInput = updateInput.replace(, ligaturefromlist)
8           search updateInput in DB.
9           If found, output: updateInput
10        end
11       else
12         search updateInput in DB.
13         If found, output: updateInput
14       end
15     end
16   end
17 end

```

Sometimes for one word can have different kinds of ligature with a different meaning. For more clarification: \$u\$y is a word with special a character (\$), trying to find out the meaning of \$ can result to different types of ligatures, e.g. stuffy, fluffly, flusty, fliffy. Now a problem arises in that, all these words are valid or known to the English language. However, we expect only one word. For this we approach to check the frequency of the word. As a result, we take the word which has the highest frequency (See Algorithm 5).

Algorithm 5: The procedure for taking the decision of ligatures for the words.

Input: list of Words**Output:** Word

```
1 for read list of Words in entry[] do
2   | key1 = entry.getKey();
3   | value1 = entry.getValue();
4   | for read list of Words in newEntry[] do
5     | key2 = newEntry.getKey();
6     | value2 = newEntry.getValue();
7     | if value1 > value2 then
8       |   value1 = value2;
9       |   key1 = key2;
10    | end
11  | end
12  | break;
13  | Output: key1;
14 end
```

5 Experiments

In this chapter, we show the experimental results for finding fast response time and good accuracy by using GNU Trove, Java, MapDB and LMDB.

We created three different text files (Text file1, Text file2 and Text file3) the experiments.

Text file1: contain words and the frequencies of the words (See Chapter: Data collection and preparation for more detail).

Text file2: contain input words with hyphen and the expected output word per line.

Text file3: contain input words with "\$" (the "\$" we have used as an illustration for the ligature) and the expected output word per line.

Text file2 is structured for the experiments of hyphenated words. It contains per line input words with a hyphen (for example "after-effect") and the expected word is either "true" or "false". "True" means a hyphen is necessary for the input word and "false" means a hyphen is not necessary for the input word. For the input word, we try to find out, compound word that can be hyphenated from a website. We also try to find from different websites [[Ass],[Oxf]] which have explanation what kind of compound words cannot be hyphenated. For example, "after-effect" cannot have hyphen, so we save this word in our input text file and also the expectation of output "false"(See figure 5.1(b) for more examples). We also collect our data from arXiv's (contain 1106572 text files), where we try to find out words with hyphen. We collected in total 4,987,829 words as an input.

Text file3 is structured for the experiments of guessing ligature. It contains input words and the expected outputs per line. Each input word contains one or more ligatures, represented by a "\$". The expected output word denotes the same word, with the correct individual characters that are encoded by the ligature(s). For example, if we found "against" which have ligatures "st". for those word combinations we put \$, so the word become "again\$". In the text file, we save our input word as "again\$" and expectation of output "against" (See figure 5.1(a)). We tried to find our input word which contains at least one ligature or more from the text file1, which we gather from different sources (see section: Data collection and preparation). We collect in total 4,546,311 words as an input.

We performed six experiments for searching words to check if a hyphen is necessary or not and five experiments for searching words to find ligature. For all experiments we went through the text files (text file2 and text file3) accordingly. For each (input

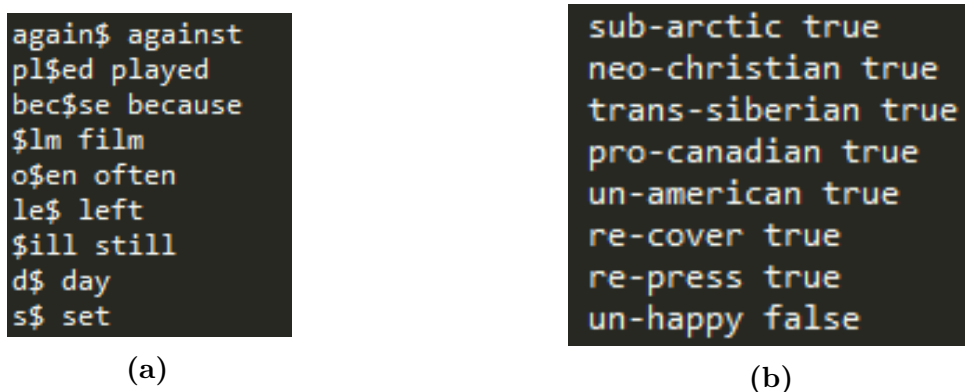


Figure 5.1: (a) Screenshot from an input text file3 for the experiment of ligature. In the screenshot first column is input data (the data with special character "\$"). We want to find out in our thesis what is the meaning of "\$". The second column is the data of our expectation (b) Screenshot from an input text2 file for experiment of the hyphenated word. The first column is the query word, for that word we want to find out is hyphen necessary or not. The second column is our expectation which only contains either "true" (Hyphen necessary) or "false" (Hyphen not necessary)

word and expected output word) pair in this file we put the input word into our algorithms (see section: Dehyphenation of words and section: guessing ligatures into chapter implementation). While getting the output from algorithms we also tried to find out the runtime (how much time every single experiment takes to give an output). And we also compare the output from algorithms with the expected output to find accuracy (is the number of correct results while comparing the result with expected output). We decide the accuracy value as follows: For every single output we got from the algorithm, we compare the result with similarity to our expectation of output. We count the number of the same result and then we divide the in-total number of the same result by the total number of searching value (see Algorithm 6).

For every single experiment Java and GNU Trove took times to load data into memory. GNU Trove took in average (out of six experiments): 42356.268 milliseconds to load data into memory from text file1 and 0.037866 milliseconds to check hyphenated data (data from text file2) into memory and it took in average (out of five experiments): 42984.998 milliseconds to load data into memory from text file1 and to find ligature (input data from text file3) 0.08082 milliseconds. On the other hand, for the same procedure Java took in average (out of six experiments): 39768.402 milliseconds to load data into memory and 0.06821 milliseconds to check hyphenated data (See table 5.1 for more detail of all experiments result). And it took in average (out of six experiments): 35257.436 milliseconds to load data into memory and to find ligature 0.07202 milliseconds (See table 5.2 for more detail of all experiments result).

However, data cannot be stored permanently in-memory. In this situation primitive collections (GNU Trove) are failing because every time we have to wait extra time to

Algorithm 6: The procedure for finding out the accuracy of the results

Input: Words (want to find out in database) and the expectation of result for the input

Output: Accuracy

```

1 while read the input words do
2   if input words found in the database then
3     result = get the word from the database.
4     if (Objects.equal(result, expectation of result)) then
5       dataMatch++;
6     end
7   end
8 end
9 accuracy = dataMatch / total no of the input

```

load data into memory. So, we need something which will store data permanently in the database and we found MapDB and LMDB from GitHub. GitHub is a website which brings together the largest community of developers to share and build better software.

MapDB needs 1,481,728 KB space for making the database (data from text file1). On the other hand, LMDB needs 531,688 KB space. To check hyphen is necessary or not MapDB has taken in average (out of six experiments) 1.39908 milliseconds and LMDB has taken in average 0.2521 milliseconds (See table 5.1 for more detail of all experiments result). For searching ligature MapDB has taken in average (out of five experiments) 0.12704 milliseconds and LMDB has taken in average 0.0187 milliseconds (See table 5.2 for more detail of all experiments result). From the experiments, we figure out LMDB gives faster runtime (only for searching word) then GNU Trove, Java and MapDB in the case for guessing ligature, but for deciding hyphenation, Java is faster (only for searching words) than others. We found 86% accuracy of the result for Java, GNU Trove, MapDB and LMDB with our approach for deciding whether a hyphen was necessary or not for the words. And 93% accuracy of the result for Java, GNU Trove, MapDB and LMDB with our approach for guessing ligature.

	Experiments no	runtime for load data in memory	runtime for searching word	accuracy
GNU Trove	experiment1	41772.59	0.0162	86%
	experiment2	42789.57	0.0396	86%
	experiment3	42417.46	0.0386	86%
	experiment4	42339.63	0.0476	86%
	experiment5	42145.56	0.0403	86%
	experiment6	42672.80	0.0449	86%
Java	experiment1	43349.937	0.0917	86%
	experiment2	46713.474	0.0864	86%
	experiment3	37957.726	0.0597	86%
	experiment4	38166.135	0.0661	86%
	experiment5	38149.48	0.0627	86%
	experiment6	34273.66	0.0427	86%
LMDB	Experiment1		0.2440	86%
	experiment2		0.2487	86%
	experiment3		0.2523	86%
	experiment4		0.2550	86%
	experiment5		0.2634	86%
	experiment6		0.2492	86%
MapDB	Experiment1		1.8935	86%
	experiment2		1.4244	86%
	experiment3		1.2437	86%
	experiment4		1.2502	86%
	experiment5		1.2708	86%
	experiment6		1.3119	86%

Table 5.1: Experimental results for dehyphenation of the words. The table shows six experimental results (for GNU Trove, Java, MapDB and LMDB) of the runtime (to load data in memory and for searching the words in memory) in millisecond and accuracy in percentage for deciding hyphenation in word. MapDB and LMDB save data into database and do not load data into memory every start of the program, that is why runtime to load data temporary in is empty.

	Experiments no	runtime for load data in memory	runtime for searching word	accuracy
GNU Trove	experiment1	41978.30	0.0773	93%
	experiment2	41929.21	0.0806	93%
	experiment3	44424.39	0.0841	93%
	experiment4	44349.46	0.0842	93%
	experiment5	42243.63	0.0779	93%
Java	experiment1	34477.15	0.0699	93%
	experiment2	34732.23	0.0696	93%
	experiment3	34600.07	0.0710	93%
	experiment4	34264.89	0.0748	93%
	experiment5	38212.84	0.0748	93%
MapDB	experiment1		0.1510	93%
	experiment2		0.1516	93%
	experiment3		0.1412	93%
	experiment4		0.0919	93%
	experiment5		0.0995	93%
LMDB	experiment1		0.0225	93%
	experiment2		0.0200	93%
	experiment3		0.0166	93%
	experiment4		0.0170	93%
	experiment5		0.0174	93%

Table 5.2: Experimental results for guessing ligature. The table shows five experimental (for GNU Trove, MapDB, LMDB and Java) results of the runtime (to load data in memory and for searching the words in memory) in milliseconds and accuracy in percentage for searching words in the database to find ligature. MapDB and LMDB do not load data into memory every start of the program, that is why runtime to load data in memory is empty.

Summary & Future Work

In this thesis, we solved two different problems. One of the simulation to decide hyphenation of the word and another simulation to guess the ligature. In both simulations, we focused on getting good response time and better quality.

In our experiments, we run the algorithms with fixed number of words. For the both simulations, we did experiments by different libraries (Lmdb and MapDB). And we found from both cases (dehyphenation of words and guessing ligatures) that Lmdb is faster (in response time) than MapDB. In the algorithms we used our approach (using frequency) for the words when ambiguity in our results occurred. With the first ambiguity to decide hyphen ("-") in the word we manage to reach 86% accuracy of the first simulation and the second ambiguity to find out expected ligature of a word, we were able to get 93% accuracy for the second simulation with our approach. Considering all these accuracy percentage, we can conclude that our approach provides an accurate solution to decide hyphenation of the words and guessing the ligature for words.

In our thesis for the first simulation deciding the hyphen in the words, we took two words as an input. For example: "go-to", two words is "go" and "to". However, there are some hyphenated word which contains more than two words. For example: "Pick-me-up", here we have three words: "pick", "me" and "up". In the future, dehyphenation of the words can be done for more than two words.

List of Abbreviations

1. The Portable Document Format (PDF)
2. Layout-Aware PDF Text Extraction (LA-PDFText)
3. The Lightning Memory-mapped Database (LMDB)
4. Multi-version Concurrency Control (MVCC)
5. Single-Level Store (SLS)
6. Write Ahead Log (WAL)
7. Generation of Bibliographic Data (GROBID)
8. application programming interface (API)
9. Java Development Kit (JDK)
10. Portable Document Format to text converter (PDFtotext)

List of Figures

Figure 1.1: Ligature with real meaning	5
Figure 1.2: Example for ligature	6
Figure 1.3: Example for mandatory (or not) hyphen in the words	7
Figure 1.4(a): Dehyphenating the word is challenging	8
Figure 1.4(b): Ambiguities in word with ligature	8
Figure 4.1: MAPDB compile dependencies	18
Figure 4.2: MAPDB test dependencies	19
Figure 5.1(a): Screen shot from an input text file3 for experiment of ligature	25
Figure 5.1(b): Screen shot from an input text file2 for experiment of the hyphenated word	25

List of Algorithms

Algorithm 1: The procedure for counting frequency of the words.	13
Algorithm 2: The procedure for combining all the text files.	14
Algorithm 3: The procedure for making the decision of hyphenated words.	21
Algorithm 4: The procedure for taking the decision of ligatures for the words.	22
Algorithm 5: The procedure for taking the decision of ligatures for the words.	23
Algorithm 6: The procedure for finding out the accuracy of the results.	26

Bibliography

- [Ass] ASSOCIATION, American P.: when do you need to use a hyphen for compound words.
- [Ber11] BERG, Oyvind R. *High precision text extraction from PDF documents*. 2011
- [BK] BAST, Hannah ; KORZEN, Claudius. *The Icecite Research Paper Management System*
- [BK17] BAST, Hannah ; KORZEN, Claudius. *A Benchmark and Evaluation for Text Extraction from PDF*. 2017
- [Bro17] BROWNLEE, Jason: What Is Natural Language Processing? (September 22, 2017)
- [CRB] CARTIC RAMAKRISHNAN, Eduard H. ; BURNS, Gully A. *Layout-aware text extraction from full-text PDF of scientific articles*
- [dav11] DAVIDG: Cleaning up pdftotext font issues. (2011)
- [Fri10] FRIEDRICH, Jöran BeelBela GippAmmar S. *SciPlore Xtract: Extracting Titles from Scientific PDF Documents by Analyzing Style Information (Font Size)*. 2010
- [HC15] HOWARD CHU, Symas C. *Lightning Memory-Mapped Database Manager (LMDB)*. 2015
- [Isaun] ISAACSON, Cory. *MapDB: The Agile Java Data Engine*. 2015-Jun
- [Isaay] ISAACSON, Cory. *Introducing MapDB: The agile Java data engine*. 2014-May
- [jim17] JIMDAVIES. *GNU Trove: High performance collections for Java*. 2017
- [LMD] Lightning Memory Database (LMDB) for Java: a low latency, transactional, sorted, embedded, key-value store <https://github.com/lmdbjava/lmdbjava>.
- [Lop09] LOPEZ, Patrice: GROBID: combining automatic bibliographic data recognition and term extraction for scholarship publications. (2009)
- [Map] MAPDB. <https://jankotek.gitbooks.io/mapdb/>.
- [Mat16] MATTHEW HARDIN. *Understanding LMDB Database File Sizes and Memory Utilization*. 2016
- [MDB] MAPDB. <https://mvnrepository.com/artifact/org.mapdb/mapdb/3.0.5>.

- [Muk16] MUKHERJEE, Mayank Singh; Barnopriyo Barua; Priyank Palod; Manvi Garg; Sidhartha Satapathy; Samuel Bushi; Kumar Ayush; Krishna Sai Rohith; Tulasi Gamidi; Pawan Goyal; A. *OCR++: A Robust Framework For Information Extraction from Scholarly Articles*. 2016
- [ORBR] OYVIND RADDUM BERG, Stephan O. ; READ, Jonathon. *Towards High-Quality Text Stream Extraction from PDF*
- [Oxf] Hyphen (-) <https://en.oxforddictionaries.com/punctuation/hyphen>.
- [Oxf18] OXFORD. *Oxford Dictionaries*. 2018
- [PL10] PATRICE LOPEZ, Laurent R.: HUMB: Automatic Key Term Extraction from Scientific Articles in GROBID. (2010)
- [Sho16] SHOR, Peter: English language and usage. (2016)
- [Ske09] SKEET, Jon: What is the most efficient Java Collections library? (2009)
- [TSF⁺15] TKACZYK, Dominika ; SZOSTEK, Pawel ; FEDORYSZAK, Mateusz ; DENDK, Piotr J. ; BOLIKOWSKI, Lukasz: CERMINE: automatic extraction of structured metadata from scientific literature. In: *IJDAR* 18 (2015), Nr. 4, S. 317--335
- [Vor14] VORONTSOV, Mikhail. *Trove library: using primitive collections for performance*. 2014