Karlsruhe Institute of Technology

Querying and Efficiently Searching Large, Temporal Text Corpora

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Jens Willkomm

aus Fulda

Tag der mündlichen Prüfung:	23. Juli 2021
Erster Gutachter:	Prof. Dr.-Ing. Klemens Böhm
Zweiter Gutachter:	Prof. Dr.-Ing. Kai-Uwe Sattler

Acknowledgements

This dissertation is the result of many years of intensive work. Throughout this time, I have received a great deal of helpful advice and assistance from many people in various ways. I am very grateful to all who have supported me in creating this work.

To begin with, I would like to express my deepest appreciation to my doctoral supervisor Prof. Klemens Böhm. Thank you for your tireless feedback, which substantially improved the quality of my writing and thinking.

Next, I would like to extend my sincere thanks to Prof. Kai-Uwe Sattler for serving as second reviewer despite becoming president of TU Ilmenau.

Furthermore, I would like to acknowledge Martin Schäler for his assistance and mentoring. I would also like to recognize the effort that I received from Michael Schefczyk and Christoph Schmidt-Petri. You both have provided the philosophical expertise that was necessary to get started with my dissertation project.

Finally, I would like to thank my peers in my research group for many interesting discussions and for encouraging me to think beyond my research field to see the bigger picture. Special thanks go to Holger Trittenbach and Adrian Englhardt, who have rendered a great amount of assistance during my writings. Special gratitude also goes to Michael Vollmer, who has provided me with encouragement and mental support during the hard times. I thank you for your support and the great time I had in Karlsruhe.

Abstract

Natural language is constantly evolving and adapts to changes in society and technology. Studying such linguistic changes is of interest since it helps to understand the language and society of today. Due to the limited reading speed of humans and the large amounts of books, such studies are laborious, costly, and time-consuming. The digitization of existing books has resulted in large text corpora, that is, sets of words and phrases and their usage frequency in the source text. A temporal extension to text corpora takes the time into account when the source text was written. Such temporal text corpora allow one to consider the usage frequency of words and phrases over time and, therefore, provide the potential to systematically analyze changes in language in a data-driven way. Currently, historians and linguists use temporal text corpora to empirically study simple questions, like the importance of a word over time. To study more complex questions, however, the information needs exceed the usage frequency of words. It is an open question which specific information needs exist, how to formulate them as database query, and how to efficiently process queries on large temporal text corpora.

In this thesis, we study these questions and design an information system to efficiently query the relevant information on a temporal text corpus. We, initially, investigate the information needs together with domain experts as part of an interdisciplinary project and develop a way to formulate them as query. Subsequently, we provide query optimization and improve data access for these queries. Altogether, we make the following contributions. Our first contribution is a query algebra for temporal text corpora. In cooperation with philosophers, we identify the information needs to examine linguistic changes in a data-driven way. The information needs originate from the works of Reinhart Koselleck, one of the most important historians of the 20th century. To gain the relevant information from the data, we specify necessary data transformations and define these as operators. We show that one can formulate the information needs as an algebraic expression in our query algebra. Our second contribution deals with cardinality estimation of strings, especially of so-called co-occurrences, namely, words that occur next to each other in phrases. Co-occurrences are of particular interest since a word's co-occurrences indicate its meaning. We conceive a lossy compression technique that reduces the size of strings but keeps the co-occurrence information. Our technique provides accurate cardinality estimation and enables query optimization for this kind of query. Our third contribution is a data structure for time series similarity search in arbitrary time intervals. Queries on temporal text corpora usually focus on specific time intervals, like the period around a historical event. To accelerate such queries, we develop a data structure that resembles a search tree but uses time series envelopes, in other words, curves outlining the extremes of time series. Once constructed on the entire time domain, we can efficiently query all possible time intervals. In summary, this thesis presents techniques to systematically analyze temporal text corpora and allow experts to gain new knowledge about the evolution of language.

Zusammenfassung

Natürliche Sprache ist eine Art der menschlichen Kommunikation und sie unterliegt ständiger Veränderung. Sowohl Inhalt als auch Aufbau natürlicher Sprache gleicht sich Änderungen in Gesellschaft und Technik an. Beispiele für derartige Sprachveränderungen sind der Bedeutungswandel existierender Wörter oder das Entstehen neuer Wörter. Die Untersuchung von Sprachveränderungen ist von Interesse, da es unser heutiges Verständnis von Sprache und Gesellschaft erklärt. Der entsprechende Forschungsbereich ist die Begriffsgeschichte, ein Zweig der Geisteswissenschaften. Ein wesentlicher Bestandteil begriffsgeschichtlicher Untersuchungen ist, dass Geisteswissenschaftler existierende Literatur lesen, also aktuelle und auch historische Texte. Die begrenzte Lesegeschwindigkeit und die große Menge an Büchern machen derartige Untersuchungen aufwändig und erschweren umfangreiche Recherchen.

Moderne Texterkennungsverfahren ermöglichen die Digitalisierung existierender Texte zu digitalen Bibliotheken. Diese Bibliotheken liegen in aggregierter Form als Textkorpus vor. Ein Textkorpus enthält einzelne Wörter und Wortketten, sogenannten Ngrams, sowie deren Verwendungshäufigkeit in den Quellwerken. Zeitabhängige Textkorpora enthalten zusätzlich Zeitstempel, also das Veröffentlichungsjahr des Texts. Das erlaubt es die Verwendungshäufigkeit der Ngrams als Zeitreihe darzustellen.

Aufgrund ihres Umfangs und der Möglichkeit einer technischen Verarbeitung, haben Geisteswissenschaftler ein großes Interesse an der Verwendung digitaler Textkorpora. Betrachten wir beispielsweise die folgende Fragestellung: *"War Emanzipation bereits vor 100 Jahren Teil des gesellschaftlichen Diskurses?"* Die Verwendungshäufigkeit des Worts Emanzipation über die letzten 100 Jahre liefert Indizien um diese Frage zu beantworten. Geisteswissenschaftler können somit statistische Untersuchungen einfacher Fragestellungen durchführen und Erkenntnisse aus Millionen von Büchern ziehen.

Interessante Fragestellungen der Begriffsgeschichte gehen jedoch über die Betrachtung der Verwendungshäufigkeiten hinaus. Ein Beispiel für eine solche Fragestellung ist die folgende: *"Hat sich das Wort Emanzipation bereits vor 100 Jahren hauptsächlich auf die Emanzipation der Frau bezogen?"* Derart komplexe Fragestellungen erfordern die Suche nach Wörtern, die häufig zusammen mit dem Zielwort verwendet werden, oder Wörtern mit einem ähnlichen Verlauf der Verwendungshäufigkeit zu der des Zielworts. Die Untersuchung komplexer Fragestellungen ist aus den folgenden Gründen schwierig: Erstens fehlt eine Möglichkeit für technische Laien, wie Geisteswissenschaftler, ihre Anfragen an ein technisches System zu formulieren. Dabei ist es aktuell unklar welche Wortmerkmale für die Untersuchungen benötigt werden, die über die Verwendungshäufigkeit hinaus gehen. Zweitens sind Textkorpora üblicherweise sehr große Datenmengen. Das führt dazu, dass die Ausführung von Anfragen und insbesondere das Errechnen der erforderlichen Wortmerkmale viel Rechenzeit benötigen.

In dieser Arbeit entwickeln wir ein Informationssystem, dass es erlaubt komplexe Anfragen der Geisteswissenschaftler an zeitabhängigen Textkorpora zu formulieren und effizient auszuwerten. Die Thesis umfasst die folgenden drei Beiträge zur Informatik, die es ermöglichen ein solches System effizient umzusetzen.

Eine Anfragealgebra für zeitabhängige Textkorpora. Im ersten Teil des Projekts definieren wir eine Anfragealgebra für zeitabhängige Textkorpora. Diese Anfragealgebra ermöglicht es Fragestellungen aus der Begriffsgeschichte zu formulieren. Die Identifikation der Informationsbedürfnisse ist eine interdisziplinäre Aufgabe, die wir in Kooperation mit Wissenschaftlern aus dem Bereich der Philosophie erarbeiten. Unsere Untersuchungen dazu basieren auf den Arbeiten von Reinhart Koselleck, der Pionierarbeit im Bereich Begriffsgeschichte geleistet hat. Aus Kosellecks Informationsbedürfnissen leiten wir systematisch die notwendigen Operatoren für unsere Anfragealgebra ab. Unsere Anfragealgebra enthält (1) einfache Operatoren, beispielsweise zum Auswählen von Elementen anhand des Ngram-Texts, (2) zeitliche Operatoren, beispielsweise zur Suche ähnlicher Elemente anhand des Verlaufs der Verwendungshäufigkeit und (3) sprachliche Operatoren, beispielsweise zur Suche gemeinsam auftretender Wörter. Wir beweisen die Vollständigkeit unserer Anfragealgebra, indem wir zeigen, dass sich alle von Koselleck betrachteten Wortmerkmale durch unsere Operatoren ausdrücken lassen. Unsere Anfragealgebra erlaubt es Fragestellungen der Geisteswissenschaftler für die Verarbeitung mit einem technischen System zu formulieren.

Effiziente Kardinalitätsschätzung für gemeinsam auftretende Wörter. Im zweiten Teil des Projekts entwickeln wir eine Methode zur Kardinalitätsschätzung von Suchmustern auf Ngram-Datenbeständen. Häufig starten Anfragen zur Begriffsgeschichte mit der Suche nach gemeinsam auftretenden Wörtern, sogenannten Kookkurrenzen, die mittels Suchmuster selektiert werden. Ein Beispiel für solche Anfragen ist: *"Welche Kookkurrenzen existieren für das Wort Emanzipation im Jahr 1975, die es 1875 noch nicht gab?"* Zur Optimierung dieser Anfrage muss abgeschätzt werden wie viele Ngrams im Jahr 1975 das Wort Emanzipation enthalten und wie viele es im Jahr 1875 sind. Anfrageoptimierer tolerieren geringe Schätzfehler, weshalb Schätzer üblicherweise ihren Speicherverbrauch auf Kosten der Genauigkeit reduzieren. Eine Möglichkeit der Kardinalitätsschätzung bieten Suffix-Bäume. Um den Speicherbedarf eines Suffix-Baums zu reduzieren, wird dieser üblicherweise in der Tiefe limitiert. Insbesondere bei Ngrams kann eine Tiefenlimitierung dazu führen, dass ganze Wörter abgeschnitten werden. Wir entwickeln einen Ansatz, der Buchstaben mit geringem Informationsgehalt aus den Ngrams entfernt. Das reduziert den Speicherbedarf des Baums und ermöglicht eine effiziente Kardinalitätsschätzung von Suchmustern auf Ngram-Datenbeständen.

Effiziente Ähnlichkeitssuche auf Zeitreihen in beliebigen Zeitintervallen. Im dritten Teil des Projekts erarbeiten wir eine Datenstruktur zur Ähnlichkeitssuche auf Zeitreihen in beliebigen Zeitintervallen. Begriffsgeschichtliche Untersuchungen erfordern die Suche nach Wörtern mit einem ähnlichen Verlauf der Verwendungshäufigkeit in einem bestimmten Zeitintervall. Ein Beispiel für solche Anfragen ist die folgende: *"Welche 5 Wörter haben*

den ähnlichsten Verlauf der Verwendungshäufigkeit zu dem Wort Krieg zwischen 1914 und 1945?" Derartige Anfragen werden auch als k-nächste-Nachbarn-Suche bezeichnet. Um zu vermeiden, dass das gegebene Element mit jedem Element des Datenbestands verglichen werden muss, wird der Datenbestand zur Ähnlichkeitssuche üblicherweise indiziert. Bei Anfragen zur Begriffsgeschichte wird das Zeitintervall erst mit einer Anfrage definiert. Existierende Ansätze benötigen allerdings ein vorab festgelegtes Zeitintervall um die Datenstrukturen aufbauen zu können. Wir entwickeln einen Suchbaum, dessen Knoten Hüllkurven für Zeitreihen enthalten. Unsere Datenstruktur ermöglicht es Anfragen zur Ähnlichkeitssuche auf Zeitreihen in einem beliebigen Intervall effizient durchzuführen.

Insgesamt bilden unsere drei Beiträge die technische Grundlage, um komplexe Fragestellungen der Begriffsgeschichte in einer Anfragealgebra zu formulieren und effizient auf großen zeitabhängigen Textkorpora auszuwerten. Ein darauf aufbauendes Informationssystem ermöglicht es systematische Untersuchungen auf Millionen von Büchern durchzuführen und neue Erkenntnisse über die Entwicklung von Sprache zu gewinnen.

Contents

Acknowledgements . . . . . **i**

Abstract . . . . . **iii**

Zusammenfassung . . . . . **v**

List of Figures . . . . . **xiii**

List of Tables . . . . . **xv**

1 Introduction **1**
1.1 Challenges 3
1.2 Contributions 5
1.3 Thesis Outline 7

2 Fundamentals **9**
2.1 Conceptual History 9
2.1.1 An Introduction to Conceptual History 9
2.1.2 The Importance of Conceptual History 10
2.1.3 Text as the Primary Source 10
2.1.4 Types of Concepts 11
2.1.5 Important Terms in Conceptual History 11
2.2 Temporal Text Corpus 11
2.2.1 Text Corpus 12
2.2.2 Word Chains 12
2.2.3 Temporal Dimension 14
2.2.4 The Google Books Ngram Corpus 15
2.3 Information Systems and Query Processing 16
2.3.1 Query Language and Query Algebra 17
2.3.2 Query Optimization and Cardinality Estimation 18
2.3.3 Data Access 19
2.4 String Data Structures 20
2.4.1 Strings and Alphabets 20
2.4.2 Trie 21
2.4.3 Compressed Trie 22
2.4.4 Suffix Tree 22
2.4.5 Suffix Array 22
2.5 Information Theory 23
2.5.1 Entropy 23

2.5.2 Joint Entropy . . . 24
2.5.3 Conditional Entropy . . . 25

3 Related Work . . . 27
3.1 Query Algebras . . . 27
3.2 String Cardinality Estimation . . . 28
3.2.1 String Algorithms and Lossless Suffix Tree Compression . . . 28
3.2.2 Suffix Tree Pruning . . . 29
3.3 Time Series Similarity Search . . . 30
3.3.1 Types of Time Series kNN Search . . . 30
3.3.2 Techniques to Accelerate Similarity Search . . . 31
3.3.3 Mathematical Definition of Interval-focused Similarity Search . . . 31
3.3.4 Dynamic Time Warping Distance . . . 32
3.3.5 Lower Bound for a Group of Time Series . . . 32
3.3.6 Cascading Lower Bounds . . . 34

4 A Query Algebra for Temporal Text Corpora . . . 37
4.1 Koselleck's Information Types . . . 38
4.2 From Concept Types to Operators . . . 40
4.2.1 Step 1: From Concept Types to Information Types . . . 40
4.2.2 Step 2: From Information Types to Data Characteristics . . . 41
4.2.3 Step 3: Finding Data Characteristics with Operators . . . 42
4.3 Data Model . . . 42
4.3.1 Ngram . . . 43
4.3.2 Shorthand Notations . . . 44
4.3.3 Sentiment and Category . . . 44
4.4 Query Operators . . . 45
4.4.1 Topic Grouping Operator . . . 46
4.4.2 Surrounding Words Operator . . . 46
4.4.3 Sentiment Operator . . . 48
4.4.4 Text Search Operator . . . 49
4.4.5 Part of Speech Filtering . . . 49
4.4.6 Time Series-based Selection . . . 49
4.4.7 kNN Operator . . . 50
4.4.8 Subsequence Operator . . . 51
4.4.9 Absolute Value Operator . . . 52
4.4.10 Count Operator . . . 52
4.4.11 Sumup Operator . . . 53
4.4.12 Set Operators . . . 53
4.4.13 Algebraic Expressions . . . 53
4.5 Proof of Concept . . . 54
4.5.1 Hypotheses Testing . . . 54
4.5.2 Hypotheses Engineering . . . 55
4.6 Summary . . . 56

5 Accurate Cardinality Estimation of Co-occurring Words . . . **57**
5.1 The Thin Suffix Tree . . . 59
5.1.1 Our Vertical Pruning Approach . . . 59
5.1.2 Character-removing Map Functions . . . 59
5.1.3 More Complex Map Functions . . . 60
5.1.4 A General Character-removing Map Function . . . 62
5.1.5 Cases of Approximation Errors . . . 62
5.2 Error Correction . . . 63
5.2.1 Counting the Branch Conflations . . . 64
5.2.2 Counting Fewer Input Strings . . . 64
5.3 Functions Insert and Query . . . 65
5.3.1 Map Function . . . 65
5.3.2 Insert Function . . . 66
5.3.3 Query Function . . . 66
5.4 Experimental Evaluation . . . 66
5.4.1 Objectives . . . 67
5.4.2 Setup . . . 67
5.4.3 Experiments . . . 68
5.5 Summary . . . 73

6 Efficient Interval-focused Time Series Similarity Search . . . **75**
6.1 The Time Series Envelopes Index Tree . . . 77
6.1.1 Tree Structure . . . 77
6.1.2 Interval-focused Query . . . 79
6.1.3 Insert Operation . . . 81
6.1.4 Generalizing Other Approaches . . . 82
6.2 Experimental Evaluation . . . 83
6.2.1 Experimental Setup . . . 83
6.2.2 Experiment 1: Parameter Influence . . . 84
6.2.3 Experiment 2: kNN Query Performance . . . 85
6.2.4 Experiment 3: Index Build Times . . . 88
6.2.5 Results . . . 89
6.3 Summary . . . 89

7 Conclusions . . . **91**
7.1 Outlook . . . 92

Bibliography . . . **95**

List of Figures

1.1 The usage frequency of the word “emancipation” when co-occurring with the words “women” and “catholic” from 1801 to 2000. 2
1.2 The challenges to build an information system that answers queries regarding conceptual history based on a temporal text corpus. 3
1.3 Our contributions as part of an information system to formulate, optimize, and execute queries on temporal text corpora. 5

2.1 The connections between concepts and the social and political reality. . . 10
2.2 The 1-grams, 2-grams, and 3-grams of the sentence “This is an example sentence”. 12
2.3 The creation of a temporal text corpus to analyze the evolution of language. 14
2.4 An overview of query processing in a database management system. . . 17
2.5 The path from an information need to an algebraic expression. 18
2.6 Query optimization consists of plan generation and cost estimation. . . . 18
2.7 Each index provides an individual method to access data. 20
2.8 A trie for the words `kitten`, `sitten`, and `sittin`. 21
2.9 A compressed trie for the words `kitten`, `sitten`, and `sittin`. 22
2.10 A suffix tree for the words `kitten`, `sitten`, and `sittin`. 22
2.11 The individual entropy of two independent random variables. 24
2.12 The relationship between individual entropy and joint entropy. 24
2.13 The relationship between individual entropy and conditional entropy. . . 25

3.1 A set of time series enclosed by an envelope. 33

4.1 The relationship between concept types, information types, data characteristics, and operators. 40
4.2 Testing Hypotheses H1. 54
4.3 Testing Hypotheses H2. 55

5.1 The impact of character-removing map functions on a suffix tree. 61
5.2 Each node of the thin suffix tree includes a suffix count and, additionally to calculate the correction factor, a Bloom filter. 63
5.3 TST’s memory usage for various map functions and approximation levels. 69
5.4 TST’s q-error for various map functions and approximation levels. 71
5.5 The estimation accuracy as function of the tree size. 72
5.6 The query run time of the TST. 73

6.1 DTW’s non-linear alignments between the usage frequency of the words “reparations” and “battleship” in the time interval $[1910, 1960]$. 76

6.2 An example of a Time Series Envelopes Index Tree with two leaf nodes and a single inner node. 78
6.3 An LBG computation that partially includes the PAA segments at start and end of interval $[a, b]$. 80
6.4 The impact of TSEIT's node size on the number of required DTW computations. 85
6.5 The query performance measurements depending on the number of elements and different corpora. 87
6.6 Build times for the English 2-gram corpus. 89

List of Tables

2.1 An example of an annotated text corpus. 12
2.2 An example of an annotated text corpus of 2-grams. 13
2.3 An example of a temporal text corpus. 14
2.4 The languages and number of ngrams per language that are included in the Google Books Ngram Corpus. 15
2.5 Part-of-speech annotations of the Google Books Ngram Corpus. 16
2.6 The index and the suffix array for the string `kitten`. 23

4.1 Example sets for $gram_n$. 43
4.2 An example of sentiment information. 44
4.3 An example of category information. 45
4.4 Example result set of the topic grouping operator. 46
4.5 Example result set of the surroundingwords operator. 48
4.6 Example result set of the sentiment operator. 48
4.7 Example result set of the textsearch operator. 49
4.8 Example result set of the pfilter operator. 50
4.9 Example result set of the time series-based selection. 50
4.10 Example result set of the kNN operator. 51
4.11 Example result set of the subsequence operator. 51
4.12 Example result set of the absolute operator. 52
4.13 Example result of operator count. 53
4.14 Example result of operator sumup. 53

1 Introduction

Natural language is an evolving and dynamic way of human communication. The content we communicate and the structure we use are constantly adapting to changes in society and technology [Mic+10]. Such linguistic changes can be diverse, like changes in word meanings or the creation of new words [FB16; HLJ16a; HLJ16b; Pra+16; Eng+19]. Studying these linguistic changes is of interest to explain the nowadays understanding of language and society [BCK04; Kos06; RGG07; Bla12; Kol12; Ols12; MS16]. The respective research field is called conceptual history, a branch of humanities. Philosophers, historians, and sociologists who deal with conceptual history are called *conceptual historians*. Example 1.1 depicts questions that help conceptual historians to define and improve the cultural, conceptual, and linguistic understanding of words.

Example 1.1 *Take the word "emancipation". When someone, nowadays, talks about emancipation, a majority of people first think of "emancipation of women". So in our daily language use today, emancipation primarily refers to women. In the past, however, emancipation also referred to other concepts, like Catholics. Generally, emancipation describes liberation from a state of dependence. Its usage has changed over time to its present reference on women. To better understand the relationship between emancipation and women, conceptual historians ask questions of the following kind:*

1. *Since when has the word emancipation primarily referred to women?*
2. *What words have been associated with emancipation previously?*
3. *When did its relation change to the nowadays focus on women?*

Studying conceptual history is difficult since conceptual historians have to answer their questions by reading. The core of conceptual history research is developing and testing hypotheses. Basically, one can summarize a usual research procedure to the following three points [Kos04]: First, a conceptual historian comes across an unfamiliar word or a particular application of a word. Second, she hypothesizes one or more possible reasons that may cause this change. For example, a conceptual historian reads a text about the Catholic Church in the United Kingdom that was written in the early 19th century. A typical hypothesis from such a text would be: emancipation referred to Catholics in the early 19th century. Third, she tests her hypothesis by manual and example-driven literature research that includes reading texts from various points in time. In this procedure, the first two steps form the core of the intellectual work of a conceptual historian. The reading speed in the third step, however, is obviously a limiting factor here. This makes research difficult and limits broad hypothesis testing.

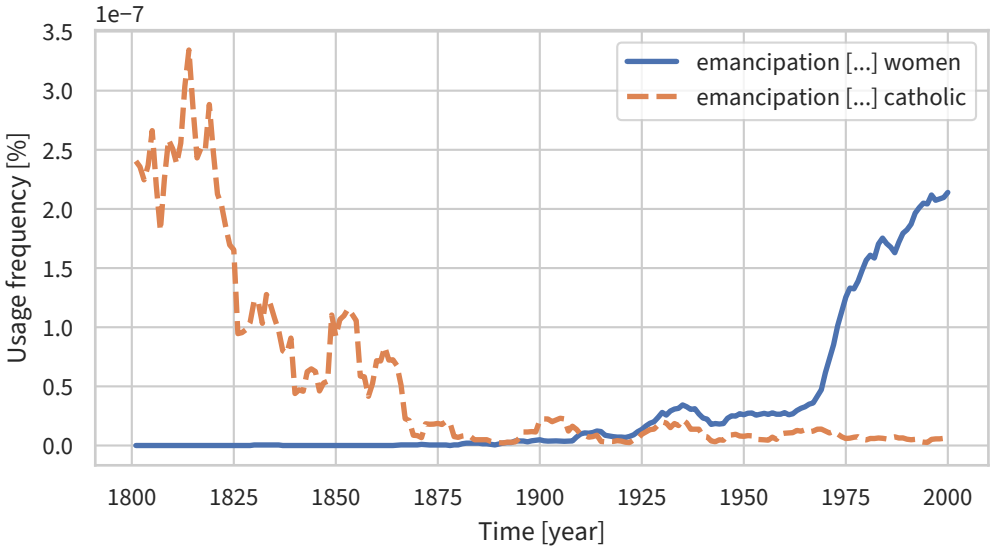


Figure 1.1: The usage frequency of the word “emancipation” when co-occurring with the words “women” and “catholic” from 1801 to 2000.[1]

With the digitization of libraries and of the humanities, conceptual historians strive to break away from their example-driven studies to become a data-driven discipline [Her99; Mor13]. Many libraries have already digitalized their book inventory to provide digital libraries. Projects, like Google Books, gather digital libraries and create large and comprehensive temporal text corpora. A temporal text corpus is a set of single words and word chains, so-called ngrams, as well as their usage frequency over time as time series [Mic+10; Lin+12]. Making use of temporal text corpora would—for the first time ever—allow statistical investigations of important hypotheses on a large scale, i. e., on millions of books.

Let us return to our questions introduced in Example 1.1. To answer the questions about emancipation, one can look at the usage frequency time series of the word “emancipation” when co-occurring either with the word “women” or “catholic” in, say, the 19th and 20th century. Figure 1.1 shows both usage frequency time series for this time interval. This plot provides indicators to answer the questions as follows:

1. Emancipation has been related to women since the beginning of the 20th century.
2. Emancipation was previously used to describe the emancipation of Catholics.
3. The relation of emancipation changed around 1920.

Interestingly, one comes to similar results by manual literature research [Kos02]. Such manual research, however, requires considerably more time and effort. Achieving a similar result with significantly less effort illustrates that temporal text corpora contain interesting information about the historical semantics of words and their potential to support conceptual historians in their research.

[1] This graph is also available at `https://books.google.com/ngrams/graph?content=emancipation%3D%3Ewomen%2Cemancipation%3D%3Ecatholic&year_start=1801&year_end=2000&corpus=15&smoothing=3`.

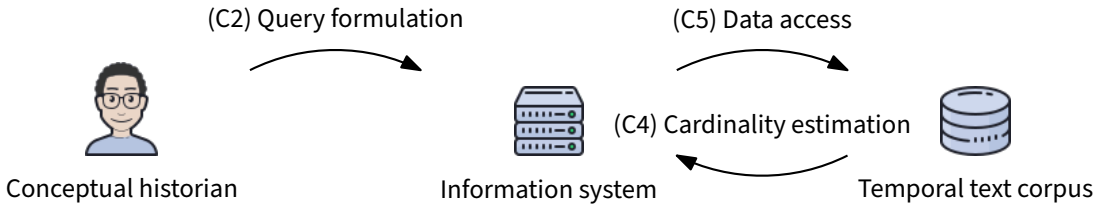


Figure 1.2: The challenges to build an information system that answers queries regarding conceptual history based on a temporal text corpus.[3]

So far, we have illustrated how a conceptual historian formulates questions regarding a hypothesis and how to answer such a question with a temporal text corpus. A natural solution to answer questions based on a database is to use an information system. One reason for this is that questions, or queries, are usually written in a declarative manner. Declarative means that the user specifies *what* information she requests, and the system creates a plan *how* to receive this information. A popular service to query a temporal text corpus is the Google Ngram Viewer.[2] The Google Ngram Viewer provides an easy way to query the usage frequency time series of a given ngram. It, however, is limited to this functionality and lacks support for corpus analysis, like querying related words or finding similar ngrams. This means that one, for example, cannot query ngrams with a similar usage frequency to the word "emancipation". As a result, the Google Ngram Viewer is insufficient to answer questions that require more than a simple time series plot.

This brings us to the question of which information is relevant to conceptual historians or, in other words, what are the *information needs* of conceptual history. The only information need that is necessary to answer our example questions is the usage frequency. Our questions, however, are reasonably easy to answer. In real-world hypotheses, there are more complex information needs. For example, one information need is to find and analyze surrounding words, so-called co-occurrences [Her99; Fri06; Fri11]. This is of interest since a change in a word's co-occurrences indicates a change in its meaning [HLJ16b; Eng+19]. This makes the analysis of co-occurrences an essential information need of conceptual history. Altogether, considering the usage frequency of ngrams is necessary but not sufficient. Existing inquiry systems lack the support to query complex information needs that are required to examine conceptual history hypotheses.

Our objective is to design an information system to examine complex hypotheses of conceptual history. To this end, we study the information needs, how to formulate these as queries, and how to efficiently execute them on large temporal text corpora.

1.1 Challenges

To query temporal text corpora, one needs to consider two sides: the user side and the data side. On the user side, one needs to translate the user's information needs to an executable analysis. On the data side, one has to handle large amounts of data. Searching

[2] The Google Ngram Viewer is part of Google Books and is available at `https://books.google.com/ngrams`.
[3] Icons by `https://icons8.com`.

and analyzing large amounts of data is a difficult task. Figure 1.2 illustrates both sides and its connections. Beyond this, there are more specific challenges in the following.

(C1) Information Needs in Conceptual History. Currently, it is unclear which information is relevant to conceptual historians. The main part when studying conceptual history is the intellectual work of the conceptual historian. Thus, each conceptual historian potentially has their own methodology. This makes it difficult to identify objective information needs in this field. Literature reflects this issue. The best-known books of conceptual history [BCK04; RGG07] primarily present results rather than the methodology or analysis steps that the authors take. The lack of a consistent source or a complete list makes it difficult to identify the information needs in conceptual history.

(C2) Query Formulation. The interface between a user, who has information needs, and a technical system, that generates answers, is the query language. To receive the desired information, a user writes her question as a query in a particular query language or query algebra. A query language that is suitable for this project must allow formulating the information needs of conceptual history. In literature, there are several query languages. One very popular query language is the Structured Query Language (SQL) [Cod70; Mai83; AHV95]. SQL basically supports structured data, i. e., data and its relations. In literature, there are several query languages for more specific purposes. On the one hand, there exist temporal query languages [Sno87; Sno95; LS03] to query time-referenced data. However, existing temporal query languages lack support to formulate necessary word features, e. g., co-occurrences or textual evidence—a key aspect in conceptual history. On the other hand, there exist query languages for text corpora [Con13; Jak+10; Zel+09]. Text corpus query languages provide support to query grammatical or lexical patterns. However, existing query languages for text corpora cannot handle temporal information. A missing query language for a temporal text corpus makes it complicated to formulate the information needs of conceptual history in an easy way.

(C3) Efficiency. To support real-world scenarios, the targeted information system requires to work efficiently. This is for the following two reasons. First, we aim at using large text corpora. Each text corpus is a sample of written language. The larger the sample, the more convincing are inferences about language based on that sample. For example, the Google Books Ngram Corpus with more than 8 million books has a size of more than 2 terabytes of data. Second, to enable users to explore a corpus, the information system requires short query run times. Exploring a corpus means that a user needs the result of one query to formulate a next query, i. e., the second query is based on the result of the first one. In this scenario, a user executes queries interactively, i. e., actively waits for the query result. To sum up, one needs to process a large amount of data and, at the same time, achieve short query run times. To realize this, each component of the information system needs to work efficiently.

Information systems usually make query processing efficient by optimizing the query and accelerating data access. Both, however, are difficult to apply to temporal text corpora. We describe why this is challenging in both cases in the following.

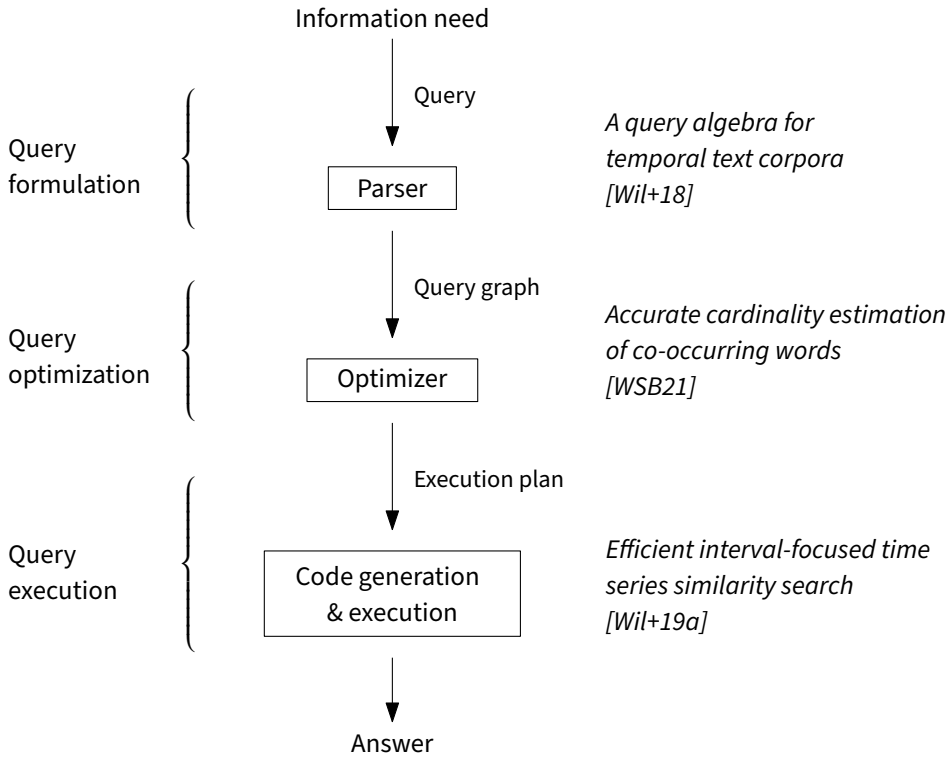


Figure 1.3: Our contributions as part of an information system to formulate, optimize, and execute queries on temporal text corpora.

(C4) Cardinality Estimation. Modern cost-based query optimizers search for the most efficient plan to execute a query by considering cost estimations, like the expected number of elements. In conceptual history, one essential kind of query is the co-occurrence query, i. e., a query for words that occur alongside each other. Such queries require keeping the words of an ngram connected. Existing cardinality estimation approaches ignore this connection between words. The resulting inaccurate cardinality estimation for co-occurrence queries makes query optimization on temporal text corpora inefficient.

(C5) Data Access. Database indices provide an efficient way to access specific elements in a large data set. When conceptual historians search for ngram with a similar usage frequency time series, they mostly focus on a specific time interval, e. g., to focus the time around a historical event or to compare two time intervals with each other. Existing indices cannot focus on arbitrary intervals when searching for similar time series. The lack of a suitable index for similarity search in arbitrary intervals makes the data access inefficient.

1.2 Contributions

In this dissertation project, we develop methods and approaches to overcome the challenges and enable users to efficiently query temporal text corpora. Thus, our information system

supports conceptual historians to analyze large amounts of books that are impossible to read within a lifetime to gain new insights into the use and change of language. To realize such an information system, we make several contributions that extend the current state-of-the-art in computer science. Figure 1.3 shows the links between our contributions to formulate, optimize, and execute queries. We describe our contributions in more detail in the following.

A Query Algebra for Temporal Text Corpora. Our first contribution is a query algebra for temporal text corpora. We design our query algebra to formulate information needs of conceptual history. Finding relevant information needs is an interdisciplinary research task. For this contribution, we cooperate with scientists from the field of philosophy. We study the information needs based on the works of Reinhart Koselleck. Koselleck is one of the most important historians of the 20th century and contributed pioneering work to conceptual history. To design our query language, we see two subtasks: one for the philosophers and one for the computer scientists. The philosophers' subtask is to identify and structure relevant information needs in conceptual history. Our subtask, as computer scientists, is to find transformations from the corpus data to the desired information and, hence, define a set of algebraic operators. As a result, our algebra includes (1) simple operators, e. g., to select ngrams by its text attribute, (2) temporal operators, e. g., to search for similar ngrams by its frequency pattern, and (3) linguistic operators, e. g., to find words that frequently co-occur within ngrams. As the basis for our operators, we define a data model to structure a temporal text corpus with a high spatial data locality. To prove the completeness of our query algebra, we show that our operators satisfy all information needs that Koselleck used in his works. This contribution addresses Challenges *(C1) Information Needs in Conceptual History* and *(C2) Query Formulation.*

Accurate Cardinality Estimation of Co-occurring Words. Our second contribution is an accurate cardinality estimation approach for co-occurrence queries. The estimate of the expected number of elements and the size of intermediate results substantially affects the quality of the query execution plan. To provide accurate estimates for co-occurrence queries, we present a novel entropy-based pruning approach for suffix trees that keeps the words of an ngram together and in order. The basic idea is to remove specific characters to shorten the strings. More precisely, we define a set of characters that we remove from all the strings in the data set. Our approach is inspired by the fact that natural language and its words have redundancy. For example, one can usually clearly read a word with a missing character. To provide a proper way to specify the set of characters to remove, we consult the information content of the characters in natural language. For this purpose, we use the alphabet's empirical entropy and conditional entropy. As a result, our pruning approach removes the most common characters from the strings. Removing specific characters from the strings preserves the full functionality of the suffix tree, especially to search for arbitrary string patterns. We show that our pruning approach keeps almost full accuracy while reducing the memory requirement by half. Thus, our pruning approach for suffix trees enables efficient cardinality estimation on ngram data sets. This contribution addresses Challenges *(C3) Efficiency* and *(C4) Cardinality Estimation.*

Efficient Interval-focused Time Series Similarity Search. Our third contribution is an efficient access method for time series similarity search in arbitrary time intervals. We develop a search tree with time series envelopes in its nodes. A time series envelope is an upper and lower band that encloses either a single time series or a group of time series. Thus, a time series envelope supports estimating a lower bound distance to a group of time series. This enables us to construct a search tree. To reduce the tree traversal cost, our tree starts with a coarse-grained envelope at the root node and becomes fully accurate at the leaf nodes. Our search tree approach avoids scanning through the entire database but prunes most time series with its hierarchical tree structure. In addition, we use envelopes also to determine a lower bound in time intervals. This facilitates building the search tree once on the full time domain and specifies the time interval at query time. For time series similarity search in arbitrary intervals, we show that our search tree has low tree traversal costs and only needs a small number of expensive similarity computations. Our search tree enables similarity search on time series in arbitrary intervals and works significantly more efficiently than competing methods. This contribution addresses Challenges *(C3) Efficiency* and *(C5) Data Access*.

1.3 Thesis Outline

This thesis consists of six chapters. Chapter 2 describes the fundamentals, including the basics of conceptual history, details on the Google Books Ngram Corpus, and how query processing generally works. Chapter 3 features related work to each of our contributions. In Chapter 4, we present CHQL, a query algebra for temporal text corpora. This chapter starts with how we identify the information needs in conceptual history. Subsequently, we define CHQL's data model and algebraic operators. In Chapter 5, we focus on the cardinality estimation of co-occurring words in an ngram corpus. This chapter introduces the Thin Suffix Tree (TST), an entropy-based pruning method for suffix trees. In Chapter 6, we turn towards time series similarity search in arbitrary intervals. This chapter proposes the Time Series Envelopes Index Tree (TSEIT), a search tree with time series envelopes as tree nodes. We complete this thesis in Chapter 7 with the conclusions and an outlook on future research questions.

2 Fundamentals

In this chapter, we describe the fundamentals of this thesis in five sections. First, we give a brief introduction to conceptual history, the application case of this thesis. Second, we describe details on temporal text corpora. Third, we write on the fundamentals of query processing. Fourth, we outline basic string data structures. Last, we depict a short overview of information theory.

2.1 Conceptual History

This section gives an overview of conceptual history. First, we give a brief introduction to the topic and, second, point out its importance to research history and the evolution of language. In the third part, we describe some particularities of conceptual history. Fourth, we introduce the idea of concept types, and last, we describe essential terms in this field.

2.1.1 An Introduction to Conceptual History

Conceptual history is a method to study history based on textual sources. The basic idea is to examine concepts that are central to political and social life. Such examinations include looking at a larger semantical, ideological, and rhetorical setting that fills a concept with meaning. One significant aspect of conceptual history is to take temporal change into account. In other words, conceptual historians analyze the relevance of particular concepts and track their changes over time [Wil+19b]. For example, take the concept of *socialism*: It might mostly express generic hopes at some moment and mostly specific fears at some other [Wil+19b]. In general, conceptual history differentiates between a word and its concepts, i. e., the ideas that are expressed by this word. For example, conceptual historians draw distinctions between the word socialism and the concept of socialism that also includes social, economic, and political aspects. Furthermore, words and concepts are interrelated since words form a concept and concepts give words their semantics. The interrelation between concepts and words is not fixed and may change over time.

One of the most important researchers in this field is Reinhart Koselleck. Koselleck made major contributions to conceptual history [Kos02; Kos04; Kos06]. Furthermore, he was the lead editor of the multi-volume compendium "Geschichtliche Grundbegriffe" [BCK04]. This 8-volume compendium with over 9,000 pages examines 122 German concepts on their respective historical semantics. Geschichtliche Grundbegriffe presents many examples of how fundamental concepts—viewed from a social perspective—change over time. Thus, this compendium is by far the most successful attempt to reconstruct our today's understanding of society and history.

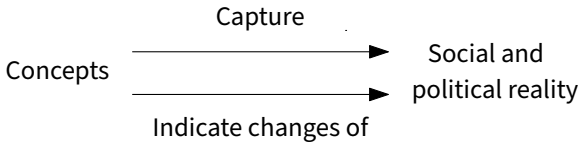


Figure 2.1: The connections between concepts and the social and political reality.

Koselleck splits conceptual history research into three tasks.[1] At the first task, one identifies the concepts that characterize history. At the second task, one hypothesizes the relation to social and political discourses and conflicts of that period. At the third task, one critically evaluates these concepts and hypotheses.

2.1.2 The Importance of Conceptual History

To understand the history and historical situations, one needs to use the proper vocabulary of that particular point in time. Therefore, conceptual historians see history as *conceptualization* of the past by the people who wrote the historical text, e. g., historians, politicians, or citizens. As a consequence, in order to understand historical texts, it is crucial to examine the historical concepts that have been used to describe the past. Figure 2.1 illustrates the connection between concepts and social and political reality.

Language is the way we report on the past. Since language is constantly changing, considering the used historical concepts matters in a similar way as the historical developments within this period. This means, understanding language is of significant importance to understand history. From this perspective, conceptual history becomes a key to all historical studies [Vog12].

Koselleck defines a separate space within the philosophy of history, a field that deals with meanings of concepts in history by analyzing historical facts in text. It takes the evolution of language into account to distinguish between a concept's semantic when writing history and a concept's semantic when discovering history. As a result, Koselleck provides an innovative way to examine historical representations and knowledge.

2.1.3 Text as the Primary Source

At first glance, conceptual history seems closely related to social history since both disciplines analyze changes in society and social life. However, both disciplines significantly differ in their practices. On the one hand, social history deals with circumstances and movements that influenced history but which are not necessarily named or referred to within the texts. For example, social history might introduce new economic theorems to study individual events or political actions. In this case, the text serves as a reference point only. On the other hand, conceptual history derives from philosophical and linguistic

[1] The details on conceptual history largely originate from the following sources: (1) Reinhart Koselleck [Kos04]. 2004. "Futures Past: On the Semantics of Historical Time", (2) Daniel Little. 2016. "What is conceptual history?", available at `https://understandingsociety.blogspot.com/2016/10/what-is-conceptual-history.html`, and (3) Kai Vogelsang [Vog12]. 2012. "Conceptual History: A Short Introduction"

disciplines, like the philosophical history of terminology, historical philology, and semasiology. Findings in conceptual history are directly based on text. This means, textual examinations form the semantics of historical concepts. In conclusion, historical concepts can be evaluated through text while they, at the same time, are based on text [Kos04].

Interestingly, conceptual history refers to history in two ways. Firstly, Koselleck studied the logic and semantics of concepts that describe historical events and processes. Secondly, he was interested in the historical evolution of some specific concepts over time. Hence, Koselleck (1) derives the methodology of conceptual history from text and (2) also examines the history of specific concepts based on text. Through this interaction, Koselleck aims to find the meaning that was associated with key historical concepts in different historical periods.

2.1.4 Types of Concepts

The knowledge about the logic and semantics of a concept enables conceptual historians to group similar concepts to a concept type. Concepts may belong to a particular concept type at a particular moment, depending on their particular semantics at this moment. Two important concept types are *parallel concepts* and *counter concepts*. Examining a concept mostly also requires analyzing its parallel and counter concepts to fully understand the semantics of a concept. Details on characteristics of concept types and the number of existing concept types are still a subject of research in the field of conceptual history.

2.1.5 Important Terms in Conceptual History

To summarize the introduction of conceptual history, we describe the meaning of important terms in conceptual history.

Concept A concept is a word with a wide range of social and political meanings. This wide range makes it ambiguous, which creates space for interpretations [And03].

Concept Type A concept type outlines a group of concepts with similar characteristics. One objective of conceptual history is to determine which concept belongs to a particular concept type during a particular period. For example, a *parallel concept* is a concept type that contains concepts with a similar role in a particular discourse, such as “love” and “peace”.

2.2 Temporal Text Corpus

This section gives details on a temporal text corpus. First, we describe a text corpus and how it is built. In the second part, we extend our definition of a text corpus from storing only single words to one that stores word chains. In the third part, we define a temporal text corpus as a text corpus with a temporal dimension. Last, we show the Google Books Ngram Corpus as the largest real-world example of a temporal text corpus.

Word	POS	Frequency
...		
mine	<all>	395,162
mine	<noun>	195,455
mine	<verb>	5,333
mine	<adjective>	56,561
...		

Table 2.1: An example of an annotated text corpus.

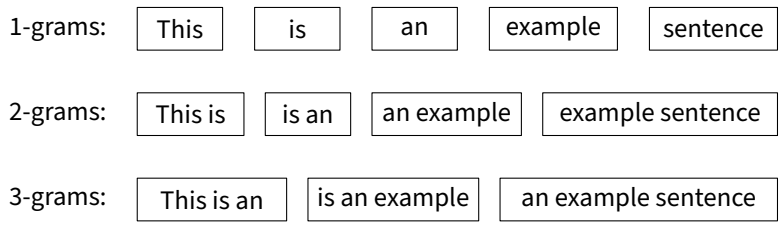


Figure 2.2: The 1-grams, 2-grams, and 3-grams of the sentence "This is an example sentence".

2.2.1 Text Corpus

A text corpus is a large collection of digital texts of a specific language [McA98]. It consists of words and their usage frequency in the source text. The primary purpose of a text corpus is to perform statistical analysis and test hypotheses on the source text or about language in general. For example, one uses a text corpus to determine the variability of a particular word or to analyze the syntactic construction of phrases [Cry92].

To make a corpus more valuable for analyses, most corpora include additional word information in the form of annotations. A popular annotation is part-of-speech tagging, i. e., each word is tagged with its part of speech (POS). Parts of speech describe the grammatical property of a word within a sentence. Examples of parts of speech are nouns, verbs, or adjectives. Table 2.1 shows an example of an annotated text corpus.

Based on this simple kind of text corpus, there are two dimensions to expand a text corpus. First, instead of storing only single words, a text corpus can additionally store word chains. Second, a temporal dimension is added in the form of timestamps when the source text has been written. This results in time-dependent frequency counts. We show both dimensions in more detail in the following.

2.2.2 Word Chains

A simple text corpus consists of single words and their usage frequency. In addition, a corpus can also store word chains. This is of interest since word chains imply information about the relation between words, e. g., what words are commonly used together, or what words modify which others.

Ngram	POS	Frequency
...		
drink coffee	<all> <all>	2,408
drink coffee	<all> <noun>	2,402
drink coffee	<verb> <all>	2,229
drink coffee	<verb> <noun>	2,226
hot coffee	<all> <all>	4,219
hot coffee	<all> <noun>	4,214
hot coffee	<adjective> <all>	4,219
hot coffee	<adjective> <noun>	4,214
...		

Table 2.2: An example of an annotated text corpus of 2-grams.

A word chain is a consecutive sequence of words. A chain of n words is called *ngram*. For example, Figure 2.2 shows the 1-grams, 2-grams, and 3-grams of the sentence "This is an example sentence". In terms of a text corpus, a 1-gram text corpus contains single words, a 2-gram text corpus contains word chains of length 2. One example of an ngram text corpus is the Microsoft Web Ngram Corpus [Wan+10]. Table 2.2 shows an example of an annotated text corpus that stores 2-grams.

Extending a corpus from single words to word chains provides significant additional value, namely the words that occur alongside each other in a specific order. This additional information is called *word co-occurrence information* and allows analyzing the word meanings. Generally, the meaning of a word is determined by its context, i. e., by the words that surround it [Har54; Fir57]. This means that ngram text corpora allow to analyze a word's meaning by looking at its surrounding words.

Analyzing surrounding words is the foundation of many modern text analysis methods. For example, there are techniques to identify the similarity between words [Li+15b; HLJ16b] and methods to perform word-sense disambiguation [NZ97]. Generally, there are two terms to describe relations between words in an ngram: co-occurrence and collocation [Kol16]. We explain both in the following.

Co-occurrences. Co-occurrences are words within an ngram that occur alongside each other in a specific order. Such a word co-occurring triggers a relation between both words. Common co-occurrences are between adjectives and nouns, like "drink" and "coffee", or nouns and nouns, like "coffee" and "tea". Co-occurrences can be found by looking at the surrounding words without considering syntactic or semantic reasons [Kol16].

Collocations. In contrast to co-occurrences that may be a random word combination, collocations are word combinations with a semantic relation. Since a semantic relation between words depends on the interpretation and application, there is no general definition of collocation. Usually, collocations are frequently recurrent combinations of words that

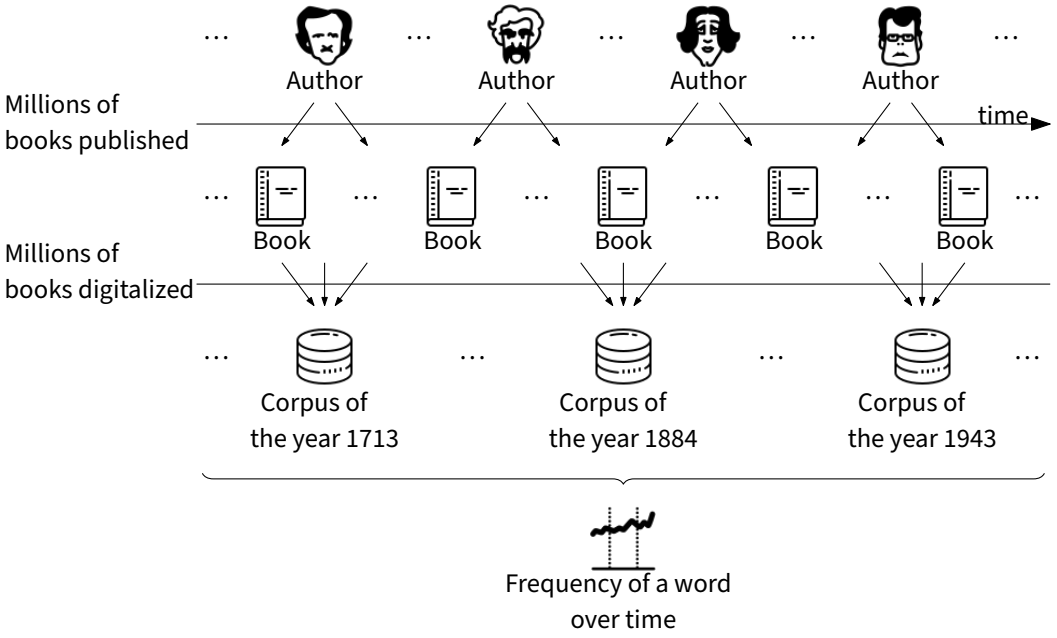


Figure 2.3: The creation of a temporal text corpus to analyze the evolution of language.[2]

Word	Year	Frequency
...		
mine	2001	392,977
mine	2002	457,423
mine	2003	519,400
mine	2004	616,434
...		

Table 2.3: An example of a temporal text corpus.

have a direct syntactic relationship [Kol16]. For example, the co-occurrence "drink coffee" is simultaneously also a collocation.

2.2.3 Temporal Dimension

Simple text corpora contain words and their usage frequency in the source texts. When the digitization process also captures and adds the times when each source text was written, the usage frequencies become time-dependent. These time-dependent usage frequencies allow inspecting changes in language. We call a text corpus with a temporal dimension

[2] This figure is based on a figure in the article [Mic+10]: Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin Andreas Nowak, and Erez Lieberman Aiden. 2010. "Quantitative Analysis of Culture Using Millions of Digitized Books." In *Science*. URL: https://doi.org/10.1126/science.1199644. Icons by https://icons8.com.

Language	Number of ngrams
English	468,491,999,592
French	102,174,681,393
Spanish	83,967,471,303
Russian	67,137,666,353
German	64,784,628,286
Italian	40,288,810,817
Chinese	26,859,461,025
Hebrew	8,172,543,728

Table 2.4: The languages and number of ngrams per language that are included in the Google Books Ngram Corpus.

temporal text corpus. Figure 2.3 illustrates the creation of a temporal text corpus. Table 2.3 shows an example of a temporal text corpus.

The time-dependent usage frequencies of a word form a time series. Mathematically, a time series is a time-ordered sequence of data points. In the above example, the time axis of a time series is discrete, and the data points are equally spaced. Recall Figure 1.1 for an example of time series from a temporal text corpus.

2.2.4 The Google Books Ngram Corpus

The most popular temporal text corpus is the Google Books Ngram Corpus. The corpus exists in three versions[3] that were released in 2009 (version 1), 2012 (version 2), and 2020 (version 3). In this thesis, we use the second version from 2012.

Version 2 of the Google Books Ngram Corpus contains data from 8,116,746 books written over the past five centuries [Lin+12]. Relatively speaking, this is 6 % of all books ever published. Due to its size, the Google Books Ngram Corpus is a suitable choice for quantitative analyses of the evolution of natural language [Mic+10]. In the following, we describe the technical details of the corpus.

Languages. The Google corpus contains over 860 billion ngrams. This corresponds to approximately 3 terabytes of information. The full corpus is grouped into 8 languages. Table 2.4 shows all languages that the corpus includes as well as the number of ngrams for each language.

Part-of-Speech Annotations. The corpus includes 12 universal POS tags that exist in a similar form in most languages. Table 2.5 lists and describes the POS tags [Lin+12].

Google Ngram Viewer. In addition to the raw corpus data, Google provides an online web frontend to query the frequency time series for a given ngram. This web frontend is

[3] All three versions of the Google Books Ngram Corpus are available at `https://storage.googleapis.com/books/ngrams/books/datasetsv3.html`.

POS tag	Description
NOUN	Nouns
VERB	Verbs
ADJ	Adjectives
ADV	Adverbs
PRON	Pronouns
DET	Determiners and articles
ADP	Prepositions and postpositions
NUM	Numerals
CONJ	Conjunctions
PRT	Particles
.	Punctuation marks
X	A catch-all for other categories such as abbreviations or foreign words

Table 2.5: Part-of-speech annotations of the Google Books Ngram Corpus.

named Google Ngram Viewer.[4] The Ngram Viewer creates plots of the frequency time series over a selected time range.

2.3 Information Systems and Query Processing

A database is a collection of data that is digitally stored and accessed from a computer system. The computer system to access a database is a database management system (DBMS). For this purpose, a DBMS interacts with users as well as the database to search and analyze the data. Existing DBMSs differ in their database model, the query language to access the database, and other features, like their internal engineering. Typically, a user writes a question in the form of a query and sends it to a DBMS. The DBMS accesses the database, generates the answer for the given query, and sends the answer back to the user. Thus, a DBMS provides an easy way for the user to retrieve information from a database. The entire procedure from receiving a query to deliver the query result is called *query processing*.

In general, query processing includes the following three steps:

1. Translate the query into an executable query plan.
2. Optimize the query plan.
3. Execute the plan to create the result.

Figure 2.4 shows the architecture for this kind of query processing. The figure illustrates the following steps. First, the query parser translates the query into a series of data manipulation operations, called query plan. Second, the optimizer rewrites the query plan

[4] The Google Ngram Viewer is available at `https://books.google.com/ngrams`.

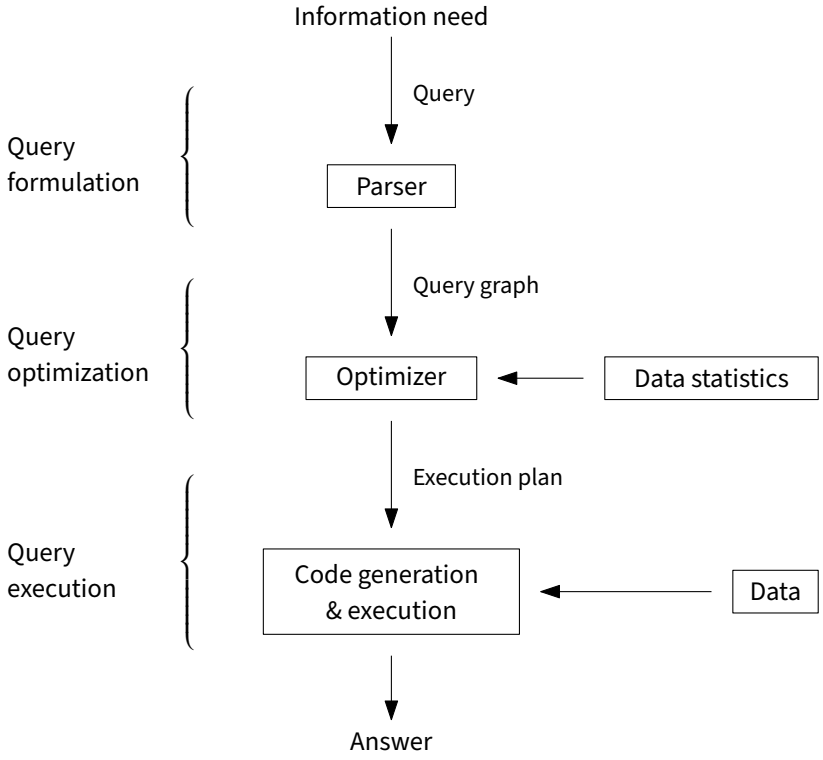


Figure 2.4: An overview of query processing in a database management system.

to provide a more efficient query processing. Third, the execution engine creates the result by executing the optimized query plan. We describe these three aspects in more detail in the following.

2.3.1 Query Language and Query Algebra

A query language specifies a way to formulate queries for a database system or an information system. In general, two kinds of query languages exist.

Declarative languages A declarative language specifies *what* information to retrieve, but not how to retrieve it. For example, a relational declarative language is the relational calculus.

Procedural languages A procedural language specifies what information to retrieve as well as *how* to evaluate it. For example, a relational procedural language is the relational algebra.

Independent of the specific query language, an information system creates an internal representation of the query. Transforming a specific query language to its internal representation is the task of the query parser. Internally a query is represented in a procedural language, typically a query algebra. Thus, the internal query representation always specifies how to evaluate the query. Figure 2.5 illustrates this transformation step.

In a query algebra, an algebraic expression represents a query. To write algebraic expressions, a query algebra consists of a set of operators and a *domain* of elements, e. g.,

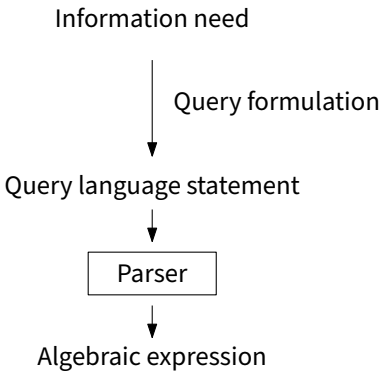


Figure 2.5: The path from an information need to an algebraic expression.

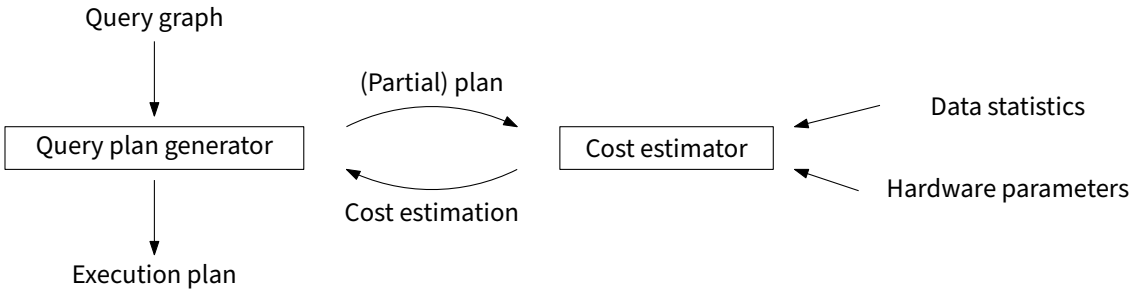


Figure 2.6: Query optimization consists of plan generation and cost estimation.

a set of words. An operator maps an element of the domain to other elements of the same domain. Each of the operators has defined *semantics*, i. e., a definition of how the result looks like when applying the operator to some input. Since both the operands and the result of an operator are of the same format, one can combine operators to express potentially complex queries. Each algebraic expression can be depicted as an operator tree that represents the equivalent query.

2.3.2 Query Optimization and Cardinality Estimation

Often, different algebraic expressions are semantically equivalent. This means, all these algebraic expressions generate the same result. The operator tree of each algebraic expression specifies the query execution plan, i. e., how to execute the individual expression. Consequently, there are various ways to generate a result that differ in their efficiency.

The initial operator tree corresponds directly to the user-formulated query without any optimization done. The query optimizer takes this initial operator tree and rewrites it into a semantically equivalent, final operator tree that is efficient to execute. The query optimizer strives to find the most efficient way to execute a given query by considering possible query plans. Figure 2.6 shows the components of a query optimizer.

To find efficient query plans, a query optimizer depends on heuristic rules for query transformation and also estimates and compares the expected costs of different query plans. Among all considered query plans, the query optimizer chooses the one with the lowest

cost estimate. A fair cost-based comparison of different query plans requires accurate cost estimates [EN16].

Estimating the cost of a query plan includes the following components [EN16].

Disk I/O cost These are the costs to read data and store intermediate results.

Computation cost These are the costs for in-memory operations.

Memory usage These are the costs in terms of main memory requirements.

Communication These are the costs to ship the result.

To estimate the cost of a query plan, a crucial input is the expected number of elements. The estimation of the expected size of a query result or subquery result is called *cardinality estimation.* An accurate estimation of the cardinality is crucial since it has an impact on multiple costs: disk I/O cost, computation cost, and memory usage. In addition, cardinality estimates are also valuable to allocate buffers of an adequate size before the query execution starts. Thus, the quality of the cardinality estimate highly influences the quality of the query optimizer.

2.3.3 Data Access

The data of a database is a set of records. In turn, a record is a sequence of a fixed number of data values. Each data value represents a member or attribute of a record. During the query processing, the record order in a database is random with respect to the search condition of the query. Searching a randomly ordered database for specific records is costly since it requires scanning the entire database.

To address this problem, one uses additional data structures, called *database indices.* A database index accelerates the retrieval of records with respect to a specific search condition. To this, an index provides an alternative access path to the data without changing the order or placement of the records in the database. Example 2.1 illustrates how an index works.

Example 2.1 *Take the index of a book as an example for an additional access path. Envision we have a book with, say, 1,000 pages and we want to find the pages that contain or describe a particular word. Without index pages, our only option is to scan through the entire book, i. e., through 1,000 pages. Only at the very end of the book, we know that we have seen every page that contains our particular word. This is an analogy to a* sequential scan *of a database. If index pages exist, we know where we have to look. The words of the index are in alphabetical order. This makes it easy for us to scan the index, find the particular word, and observe the page numbers. We can now efficiently jump to the corresponding pages and skip the rest of the book. In this way, a book index provides a method to directly access book pages when searching for particular words.*

The literature describes various types of indices. These indices use different algorithms and different data structures to suit best to a corresponding query operation. Examples of such operations are to select specific records, sort records, or find similar records. Two

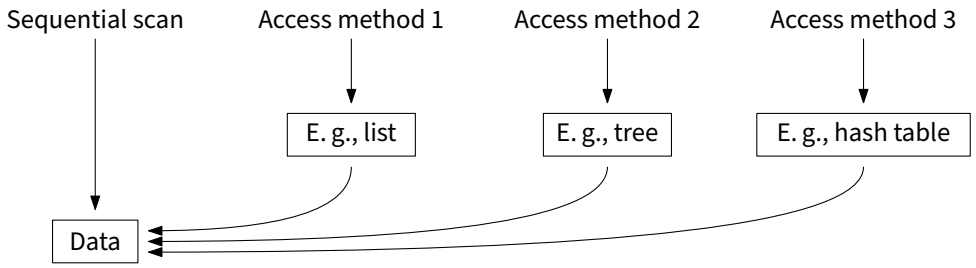


Figure 2.7: Each index provides an individual method to access data.

commonly used index structures are hash tables and tree structures, e. g., B^+ trees [EN16]. On the one hand, the speed of a hash table is constant and, thus, independent of the number of records in a database. However, hash tables only support equality comparisons as the search condition. On the other hand, the speed of a B^+ tree logarithmically depends on the number of records in a database, but such trees also support inequality comparisons and sorting. Figure 2.7 illustrates various methods to access data.

2.4 String Data Structures

To efficiently process text, one needs data structures for strings. One important string data structure is the *suffix tree*. A suffix tree provides a fast implementation of many common string operations, like finding the longest common substring in two words or locate matches of a search pattern. In this section, we describe some essential string data structures. At first, we give a formal definition of a string and an alphabet. Subsequently, we describe the data structures trie and compressed trie and, lastly, describe the suffix tree and its more compact version, the suffix array.

2.4.1 Strings and Alphabets

This section gives a formal definition of an alphabet, a string, and a suffix.

Definition 2.1 (Alphabet) *An alphabet Σ is a finite non-empty set of symbols [Hig10]. The symbols of an alphabet are also called characters.*

One uses the cardinality $|\Sigma|$ to denote the size of the alphabet, i. e., the number of characters that are available in alphabet Σ.

Definition 2.2 (String) *A string w over Σ is a finite sequence of symbols that are chosen from an alphabet Σ [Hig10]. A string with no characters is called empty string and denoted with ϵ.*

One can define operations on strings. A standard operation is a concatenation of two strings that results in a single string. Given two strings p and s, we denote the concatenation by $p \cdot s$ [Hig10]. For example, `water` $\cdot$ `proof` = `waterproof` or $\epsilon \cdot$ `water` = `water`.

Definition 2.3 (Suffix) *A string s is a suffix of a string w, if there exists a string p such that $w = p \cdot s$ [Lot02]. If string p is not the empty string, then suffix s is a proper suffix.*

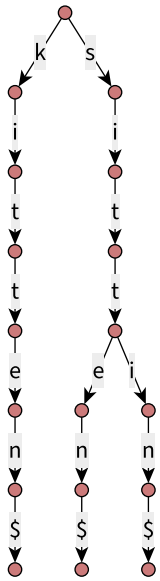


Figure 2.8: A trie for the words `kitten`, `sitten`, and `sittin`.

2.4.2 Trie

A trie is a tree data structure that stores a collection of strings over an alphabet Σ [Knu98]. Each node has between 0 and $|\Sigma|$ children. All edges in the tree are labeled with a single character. Edges that leave a particular node receive different labels. In this way, a trie stores each character of a string as a label on the path from the root node to a leaf node.

String Termination. Sharing an inner node with various strings may cause problems. If a string w_1 equals the prefix of another string w_2, string w_1 ends at an inner node rather than a leaf node. Therefore, it is impossible to decide whether both strings w_1 and w_2 are part of the trie or only string w_2.

To solve this problem, each string in a trie ends with a special symbol that is unequal to any other character of the strings. Literature usually denotes this termination symbol with symbol \$. The termination symbol ensures that no word equals a prefix of another word. For example, the termination symbol enables one to distinguish between a trie that contains the two words `system` and `systematically` from a trie that contains only the word `systematically`.

String Representation. Due to the termination symbol, for each string stored in the tree, a leaf node exists that represents this particular string. The root node represents the empty string. Inner nodes represent a prefix of a string. Therefore, strings that share a prefix also share an inner node in the trie. Figure 2.8 shows the trie for the words `kitten`, `sitten`, and `sittin`.

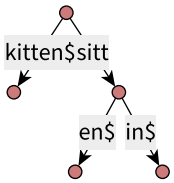


Figure 2.9: A compressed trie for the words `kitten`, `sitten`, and `sittin`.

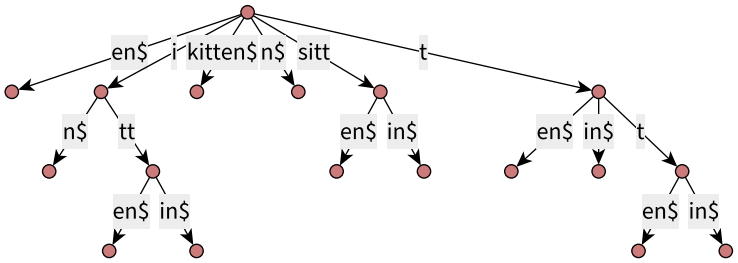


Figure 2.10: A suffix tree for the words `kitten`, `sitten`, and `sittin`.

2.4.3 Compressed Trie

A compressed trie is a space-optimized trie [Mor68]. Each node of a trie that has only a single child is merged with its parent node. The label of the removed edge is appended to the incoming edge of the merged node. Thus, this reduces the number of nodes but keeps the information of the edge labels. Figure 2.9 shows the compressed trie for the words `kitten`, `sitten`, and `sittin`.

2.4.4 Suffix Tree

A suffix tree is a compressed trie that includes a string and all its suffixes [Gus97]. For example, the string `nana` is a suffix of the string `banana`. To efficiently construct a suffix tree, we refer to Ukkonen's algorithm [Ukk95] and Farach's algorithm [Far97]. Figure 2.10 shows the suffix tree for the words `kitten`, `sitten`, and `sittin`.

2.4.5 Suffix Array

A suffix array addresses the same problems as a suffix tree by using a linear array [MM93; AN95]. Enhanced with some additional information, a suffix array provides the same functionality with the same run time complexity as a suffix tree [AKO04]. However, a suffix array requires less memory space as a suffix tree. Suffix arrays and suffix trees are closely related to each other and can be used interchangeable [NB00; AKO04].

A suffix array is a sorted array of all suffixes of a string. Instead of storing each suffix explicitly, a suffix array only stores the indices of the suffixes within this string. Table 2.6 shows the suffixes of the string `kitten`. The resulting suffix array for the string `kitten` is

$$[4, 1, 0, 5, 3, 2].$$

To find a suffix, one runs a binary search on the suffix array.

Suffix	Index
kitten	0
itten	1
tten	2
ten	3
en	4
n	5

(a) All suffixes and their corresponding index.

Sorted suffix	Suffix array
en	4
itten	1
kitten	0
n	5
ten	3
tten	2

(b) All suffixes in lexicographic order produce the suffix array.

Table 2.6: The index and the suffix array for the string `kitten`.

2.5 Information Theory

Information theory studies what information generally is and how to measure the amount of information. Hence, information theory is the basis for the coding, compression, and transmission of information. The mathematical foundation of information theory was established by Claude Elwood Shannon [Sha48]. Shannon defined a measure, called *entropy*, to quantify the information that a message carries. Shannon's entropy is a key component in information theory.

2.5.1 Entropy

Entropy measures the *information content* of a message by determining the surprisal value of the message content. We illustrate the connection between information and surprisal value in the following. If one observes an expected event, it is hardly surprising. Hence, such an event contains very little information. If, in contrast, an event happens unexpectedly, it carries much more information. This means that, for an event Ev, the information content increases when the probability $p(Ev)$ of this event decreases. Example 2.2 illustrates this.

Example 2.2 *Let us consider a coin toss. With a fair coin, the probability of observing heads is the same probability as observing tails. Since there are two possible outcomes that occur with the same probability, the actual outcome contains one bit of information. Now, we use another coin that has two heads and no tail. The outcomes of coin tosses will always be heads. Since this outcome is predictable, the actual outcome contains no information.*

Equation 2.1 defines the entropy $H(X)$ of a discrete random variable X [CT06].

$$H(X) = -\sum_{x \in X} p(x) \cdot \log p(x) \tag{2.1}$$

Literature [CT06] has the convention that $0 \cdot \log 0 = 0$. If using the logarithm to the base 2, the entropy is of unit *bits*. To explain relationships between the entropy of different

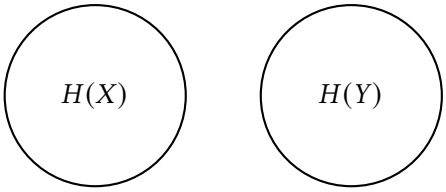


Figure 2.11: The individual entropy of two independent random variables.

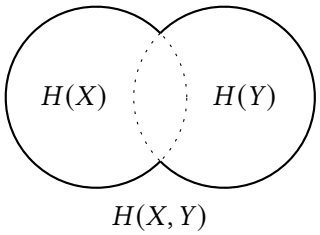


Figure 2.12: The relationship between individual entropy and joint entropy.

variables X and Y, it is common to illustrate entropy as a Venn diagram. Figure 2.11 shows the individual entropy of two variables X and Y.

In Example 2.2, the true probability distribution is known. However, there are cases where the true probability distribution is unknown. In such cases, we can use an estimate of the probability distribution to calculate the entropy. Literature calls this *empirical entropy* [CT06].

Entropy of Text. One application of empirical entropy is to analyze texts that are written in a natural language, like English. Formally, a text is a string of characters from a finite alphabet. The true distribution of the characters in English text is unknown, but one can estimate the distribution by an empirical analysis of an English text corpus.

The usage frequency of characters in natural language is unevenly distributed [SEW04]. For example, it is far more likely to observe character `e` than character `q`, and it is more likely to observe character chain `er` than chain `em`. This makes natural language text easy to predict [Sch15] and demonstrates the low entropy of natural language text.

2.5.2 Joint Entropy

Joint entropy extends entropy to a pair of discrete random variables X, Y [CT06]. Equation 2.2 defines joint entropy.

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \cdot \log p(x, y) \tag{2.2}$$

Again, it is convention that $0 \cdot \log 0 = 0$. Figure 2.12 depicts the relationship between individual entropy and joint entropy as Venn diagram.

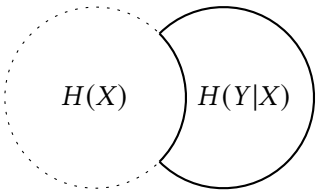


Figure 2.13: The relationship between individual entropy and conditional entropy.

2.5.3 Conditional Entropy

Conditional entropy $H(X, Y)$ quantifies the amount of information of a discrete random variable Y given knowledge of another discrete random variable X [CT06]. Equation 2.3 defines conditional entropy.

$$H(Y|X) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \cdot \log \frac{p(x, y)}{p(x)} \tag{2.3}$$

Similar to above, it is convention that $0 \cdot \log \frac{c}{0} = 0 \cdot \log 0 = 0$; for a constant $c > 0$. Figure 2.13 depicts the relationship between individual entropy and conditional entropy as Venn diagram.

3 Related Work

This chapter features related work to each of our contributions. In the first part, we classify related work on query algebras. In the second part, we outline string compression algorithms and cardinality estimation techniques. In the last part, we describe types of time series similarity search and the mathematical background.

3.1 Query Algebras

Developing methods (and systems) allowing to analyze large text corpora, e. g., from a linguistic or philosophic perspective, is a current trend that has created the digital humanities. We now review solutions from this field and declarative query languages in general.[1]

Work in digital humanities mainly consists of data processing and the analysis of text corpora [Hai17; War12]. For example, distant reading is a known idea for text analysis [Mor13]. Distant reading is a vision to study literature by applying digital methods to large text corpora. Thereby, the term distant reading refers to the higher vision rather than particular methods or systems. Distant reading contrasts with the conventional human close reading of individual texts.

Existing text-analysis solutions focus on linguistic and reflective properties as well as their evolution, i. e., changes over time, such as [HLJ16a; Pra+16; HLJ16b]. Respective systems cannot output the required information to conduct research on conceptual history in a comprehensive way. In addition, such systems do not provide a sufficiently abstract interface, a reason why experts are reluctant in using them [Hai17].

A very common query algebra is the relational algebra [Cod70; Mai83; AHV95]. However, it does not contain sufficiently specific operators, e. g., temporal or linguistic operators. There are extensions to add temporal operators [Sno87; Sno95], but not linguistic operators. To query relation between words, there are special-purpose query languages. For example, SQWRL is a language to query an ontology [OD09]. Querying word relations, e. g., from an ontology, does not include all linguistic relationships needed. Further, ontologies do not provide temporal information. SQWRL does not contain any temporal operator.—All of these algebras have in common that they do not cover both linguistic and temporal operators required for research on conceptual history.

[1] This section is an extended version of the related work section in our article [Wil+18]: Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. 2018. “A Query Algebra for Temporal Text Corpora.” In *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries* (*JCDL '18*). DOI: 10.1145/3197026.3197044.

3.2 String Cardinality Estimation

This section shows related work in the field of cardinality estimation for string attributes.[2] We split this section into two parts. First, we summarize lossless methods to compress strings and suffix trees. These methods reduce the memory consumption without loss of quality, i. e., they allow for a perfect reconstruction of the data and provide exact results. Second, we turn to pruning methods for suffix trees. Pruning is a lossy compression method that approximates the original data. In most cases, lossless compression can be applied in addition to a pruning technique to reduce the memory requirements of a string data structure.

3.2.1 String Algorithms and Lossless Suffix Tree Compression

This section gives a brief overview of string algorithms. First, we show lossless string compression methods. Second, we differentiate cardinality estimation from approximate string matching. Last, we describe techniques to compress suffix trees. In contrast to cardinality estimation, most string algorithms provide exact answers, i. e., without false positives, and aim at optimizing the space-time trade-off, i. e., they make the representation of the data structure more concise to save space. Therefore, lossless string compression methods are orthogonal to cardinality estimation methods and can be applied additionally.

String Compression Methods. There exist various methods to compress strings. One is statistical compression. Statistical compression exploits the variable frequency of symbols by assigning shorter codewords to more frequent symbols. Well-known codes are Huffman coding [Huf52] and Hu-Tucker coding [HT71]. Second, there are compressed text self-indexes. A self-index stores the full-text and supports indexed searches on the text [NM07; Fer+09]. A well-known compressed text self-index is the FM-index [FM05; Fer+07]. Third, there is dictionary-based compression. The approach is to construct an adaptive dictionary of symbol chains. The most common representative of this technique is the Lempel and Ziv (LZ) compression family [ZL77; ZL78; Wel84; AF14]. Fourth, there is an approach, called antidirectories, which defines a set of forbidden words. On a binary alphabet, the knowledge of forbidden words is used to achieve compression [Cro+00; CN02; FH08; OM10; FG19]. Fourth, there are grammar-based compression methods. The idea is to find a context-free grammar that generates a given string uniquely. For example, there exist Sequitur [NW97], Re-Pair [LM00], and other methods [Kie+00; CGW16]. However, all these compression methods are either incompatible with pattern matching or, as already stated, orthogonal to pruning methods.

Approximate String Matching. Approximate string matching is the problem of finding strings that are similar to the search string. This means to find either all strings within a maximal string distance [GG88; Bin+19] or the most similar strings [HD80; Ukk92].

[2] This section is an extended version of the related work section in our article [WSB21]: Jens Willkomm, Martin Schäler, and Klemens Böhm. 2021. "Accurate Cardinality Estimation of Co-occurring Words Using Suffix Trees." In *Proceedings of the 26th International Conference on Database Systems for Advanced Applications* (*DASFAA '21*). DOI: 10.1007/978-3-030-73197-7_50.

Therefore, approximate string matching finds slightly different strings but does not allow false-positive results.

Related to approximate string matching, there is string neighborhood generation that constructs a list of strings within a certain distance. One definition of a neighborhood is the reduced-alphabet neighborhood [Boy11]. The idea is a hash function that maps a string character-wise from the full alphabet to a reduced alphabet. The neighborhood generation works in two steps. First, it creates a neighborhood candidate list by using the reduced alphabet. To receive the exact string neighborhood, it, secondly, filters the candidate list by using the full alphabet. The approach to use a hash function to manipulate the alphabet is also of interest when working on cardinality estimation.

Suffix Tree Compression Methods. A suffix tree (or trie) is a data structure to index text. It efficiently implements many important string operations, e. g., matching regular expressions. To reduce the memory requirements of the suffix tree, there exist approaches to compress the tree based on its structure [NT02]. Earlier approaches are path compression [KP86; KP88; KMF17] and level compression [AN93; AN94]. More recent compression methods and trie transformations are path decompositions [Fer+08; HO13; GO15], top trees [Bil+15; HR15; BFG17], and hash tables [DCW93; PR15]. Such methods optimize the memory representation of the suffix tree structure to be more concise and, as already stated, are orthogonal to tree pruning methods.

In addition to structure-based compression, there exist alphabet-based compression techniques. Examples are the compressed suffix tree [GV05], the sparse suffix tree [KU96] and the idea of alphabet sampling [Cla+12; GR15]. These methods reduce the tree size at the expense of the query time. All these methods provide exact results, i. e., are lossless compression methods. Lossless tree compression is applicable in addition to tree pruning methods.

Relation to Cardinality Estimation. We presented important works on string indexing and string compression. All mentioned algorithms answer queries exactly, i. e., without false positives. Exact data structures often serve as fundament to store the necessary information to provide a cardinality estimate. Hence, cardinality estimation directly benefits from future improvements in the field of exact string compression.

3.2.2 Suffix Tree Pruning

Suffix trees allow estimating the cardinality of string predicates, i. e., the number of occurrences of strings of arbitrary length [VMS15]. With large string databases, in particular, a drawback of suffix trees is their memory requirement [DN00; VMS15]. To reduce the memory requirements of the suffix tree, variants of it save space by removing some information from the tree [KVI96].

We are aware of three approaches to select the information to be removed: A first category is data-insensitive, application-independent approaches. This includes shortening suffixes to a maximum length [KVI96]. Second, there are data-sensitive, application-independent pruning approaches that exploit statistics and features of the data, like remov-

ing infrequent suffixes [Gog+14]. Third, there are data-sensitive, application-dependent approaches. They make assumptions on the suffixes which are important or of interest for a specific application. Based on the application, less useful suffixes are removed [SAB08]. For example, suffixes with typos or optical character recognition errors are less useful for most linguistic applications. It is also possible to combine different pruning approaches. Here, we focus on data-sensitive and application-independent pruning.

Horizontal Pruning Approaches. Existing pruning techniques usually reduce the height of the tree by pruning nodes that are deeper than a threshold depth [KVI96]. Another perspective is that all nodes deeper than the threshold are merged into one node. We name such depth-limiting approaches *horizontal pruning*.

3.3 Time Series Similarity Search

This section describes related work in the field of time series similarity search using k-nearest neighbors (kNN) algorithms.[3] We split this section into six parts. First, we discern interval-focused kNN search from other similarity search problems. Second, we present three techniques that are used in related work to accelerate time series similarity search. Third, we show the mathematical definition of interval-focused kNN. Fourth, we define the dynamic time warping distance, a common similarity measure for time series data. Fifth, we write on lower bounds for the dynamic time warping distance measure. Last, we describe a cascade of lower bounds.

3.3.1 Types of Time Series kNN Search

Categories of related work in the field of time series kNN search are *whole matching*, *subsequence matching* and *interval-focused matching* [Aßf+07]. A whole matching query $\text{kNN}(k, Q)$ returns the k time series with the smallest distance to the query time series Q over the full time period, i. e., the full time series length. There are various approaches for whole matching kNN search [KJF97; CF99; CFY03; KR05; NRR10]. Whole matching is different from interval-focused kNN, as one would have to, say, build a whole matching kNN index for each window $[a, b]$. A subsequence matching query $\text{kNN}(\delta, M)$ returns all time series containing a time series motif M, at any point in time. A time series motif is a pattern for time series [CKL03]. Parameter δ defines the maximal deviation of the subsequence from the motif. Again, there are many approaches [FRM94; Li+02; LPK07; Du+08; Rak+13; LR13; Gil+15], and again, the problem is different from interval-focused kNN. An interval-focused matching query $\text{kNN}(k, Q, [a, b])$ returns the k time series most similar to the query time series Q where only the similarity in the range $[a, b]$ matters [Aßf+07]. This is the problem of interest here.

[3] This section is an extended version of the related work section in our article [Wil+19a]: Jens Willkomm, Janek Bettinger, Martin Schäler, and Klemens Böhm. 2019. "Efficient Interval-focused Similarity Search under Dynamic Time Warping." In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases* (*SSTD '19*). DOI: 10.1145/3340964.3340969.

3.3.2 Techniques to Accelerate Similarity Search

Due to the scarcity of work on interval-focused kNN, we have analyzed existing approaches from whole matching and subsequence matching and now review three basic techniques to accelerate time series kNN queries.

Spatial Access Methods. A common technique to speed up similarity search for time series is to index time series with a spatial search tree, e. g., an R-tree or one of its variants [Bec+90; SK91; AFS93; FRM94; Agr+95; LLM04; KR05; Fu+07; Ass+08; Gil+15]. Vanilla spatial search trees are not ideal to handle time series for the following reasons. First, spatial search trees poorly handle high dimensional spaces [WSB98; Sch+13]. Second, the usage of bounding rectangles restricts an R-tree to metric distance measures [Gut84], like the Euclidean distance. Third, spanning bounding rectangles across time series data points leads to hypercubes of very large volumes and, thus, insufficiently separate the elements. To avoid the problem of high dimensional space indexing, some approaches extract time series features, e. g., wavelet coefficients, and index the feature vectors instead of the time series using a spatial tree [CF99; Keo+01; KPC01; CFY03]. However, time series feature extraction approaches are unusable for interval-focused similarity search, since they usually remove the time domain.

Data Partitioning. When partitioning similar time series into groups [NRR10; KS10], a query starts with sequentially scanning the group most similar to the query. It continues scanning the groups in descending order of their similarity to the query and stops when the remaining groups are less similar than the best ones so far. This approach can achieve a high pruning rate if the time series groups have significantly different shapes. However, the number of partitions tends to grow linearly with the time series, i. e., browsing and deciding whether to scan or discard a partition becomes more expensive. This makes this approach inefficient for large data sets.

Lower Bounding Cascade. Various approaches improve a sequential scan by checking a lower bound or a cascade of lower bounds before computing the real distance [SYF05; Rak+12]. This works best for long time series, as it saves time by pruning time series without computing its real distance. However, since it relies on a sequential scan, it necessarily needs to sequentially scan the entire database and consider every time series.

Combining Various Techniques. Based on this analysis, we conclude that any existing approach that can (also) be used for interval-focused kNN only uses *one* of the above techniques. It is of interest to study their performance and combine existing approaches to a generalized technique. Before we study such a generalization, we look at the mathematical background in the following.

3.3.3 Mathematical Definition of Interval-focused Similarity Search

This section, first, describes the mathematical definition of a time series and, second, of interval-focused kNN search.

Time Series. A time series C is a sequence $\langle c_1, \ldots, c_a, \ldots, c_b, \ldots, c_l \rangle$ of length $l > 0$. $C[a, b]$ denotes a subsequence of time series C beginning at element c_a and ending at element c_b. $\mathcal{S}$ is a set of time series that contains $|\mathcal{S}| = L$ time series. All time series of set $\mathcal{S}$ have the same length l. The Google Books Ngram Corpus, for example, has this property.

Interval-focused k-Nearest Neighbor. Querying the k-nearest neighbors of a query time series Q results in a set $\mathcal{R} \subseteq \mathcal{S}$ of $\min(k, |\mathcal{S}|)$ time series so that for any two time series $C_R \in \mathcal{R}$ and $C_S \in \mathcal{S} \setminus \mathcal{R}$ it holds that $d(Q, C_R) \leq d(Q, C_S)$ regarding a distance measure d. This kNN variant is whole-matching kNN. A generalization is the interval-focused kNN query, defined as $\text{kNN}(k, Q, [a, b])$ [Aßf+07]. The interval-focused kNN query only considers the interval $[a, b]$ within time series Q and C to determine the distance. Obviously, $\text{kNN}(k, Q, [1, l]) = \text{kNN}(k, Q)$.

3.3.4 Dynamic Time Warping Distance

The result of a kNN query depends on the distance measure d. Here, we focus on the dynamic time warping distance (DTW). Equation 3.1 shows the distance between two time series Q and C of length m and l using DTW [SC78].

$$DTW(Q, C) = \sqrt[p]{D(m, l)} \tag{3.1}$$

$$D(i, j) = |q_i - c_j|^p + \min \begin{cases} D(i-1, j-1) \\ D(i-1, j) \\ D(i, j-1) \end{cases} \tag{3.2}$$

where $D(0, 0) = 0$ and $D(i, 0) = D(0, j) = \infty$ for $1 \leq i \leq m$ and $1 \leq j \leq l$.

3.3.5 Lower Bound for a Group of Time Series

In the following, we say how to represent a group of time series and how to compute a lower bound to such a group. We divide this section into four parts. The first part describes the idea of *envelopes* and defines them. The second part defines piecewise aggregate approximation, a technique to approximate time series. The third part says how to use piecewise aggregate approximation on time series envelopes. The last part shows how to compute a lower bound of the DTW distance for both an envelope and an approximate envelope.

Time Series Envelope. An envelope $E = \langle e_1, \ldots, e_i, \ldots, e_l \rangle$ represents a set C of time series $C = \langle c_1, \ldots, c_i, \ldots, c_l \rangle$. Envelope E consists of elements $e_i = \langle ue_i, le_i \rangle$ representing a so-called upper sequence $ue_i = \max_{C \in C}(c_i)$ and a lower sequence $le_i = \min_{C \in C}(c_i)$ [KR05]. Figure 3.1 illustrates the idea.

Piecewise Aggregate Approximation. A piecewise aggregate approximation (PAA) reduces the dimensionality of a time series, i. e., the number of data points [Keo+01]. To do so, PAA creates segments of a fixed size and aggregates all data points of a segment to one data

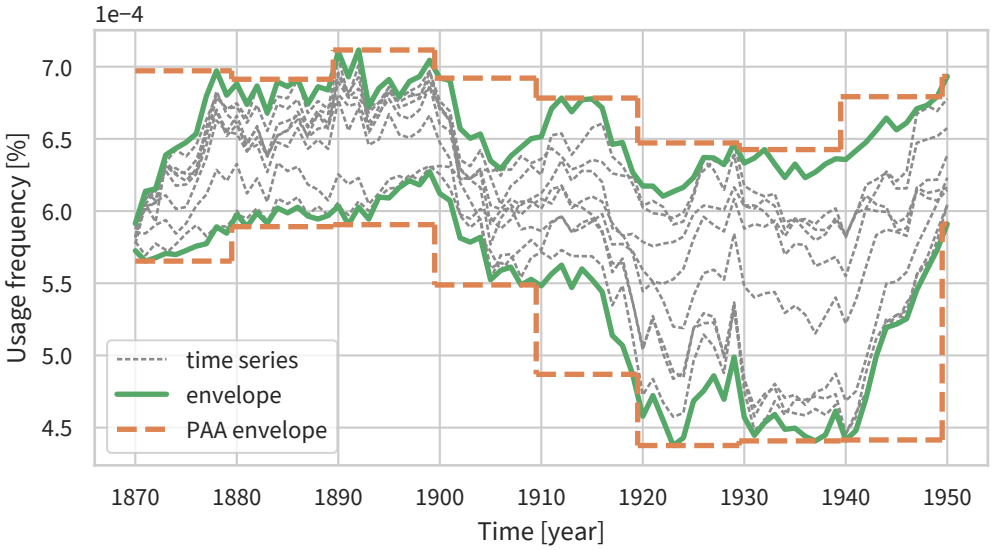


Figure 3.1: A set of time series enclosed by an envelope.

point, e. g., to the mean, min, or max value. We refer to the PAA version of time series C with a fixed segment length of T as C^T. The DTW distance for two PAA time series C^T and Q^T is a lower bound for the DTW distance of the original time series C and Q [SYF05]. Equation 3.3 shows this property.

$$\mathrm{DTW}(C^T, Q^T) \leq \mathrm{DTW}(C, Q) \tag{3.3}$$

PAA Envelope. Since an envelope consists of an upper and a lower sequence, one can create a PAA version E^T of an envelope E. To receive a valid PAA envelope, one must aggregate the upper sequence to the maximum and the lower sequence to the minimum [NRR10]. Equation 3.4 defines a PAA envelope.

$$E^T = \langle e_1^T, \ldots, e_i^T, \ldots, e_t^T \rangle \tag{3.4}$$
$$e_i^T = \langle ue_i^T, le_i^T \rangle \tag{3.5}$$
$$ue_i^T = \max(e_u, \ldots, e_v) \tag{3.6}$$
$$le_i^T = \min(e_u, \ldots, e_v) \tag{3.7}$$

where $u = (i-1) \cdot T + 1$ is the start of segment i and $v = i \cdot T$ the end. PAA reduces the envelope length to t with $1 \leq t \leq l$. Figure 3.1 illustrates an envelope and its coarse PAA variant.

Group Lower Bound. The lower bound for a group of time series (LBG) is a lower bound of the DTW distance from a time series Q to an envelope E, i. e., to all time series that envelope E encloses [NRR10]. See Equation 3.8.

$$\mathrm{LBG}(Q, E) \leq \arg\min_{C \in \mathcal{C}} \mathrm{DTW}(Q, C) \tag{3.8}$$

LBG works as follows: For every point in time i, the data point q_i of time series Q can either be inside envelope E, i. e., $le_i \leq q_i \leq ue_i$, or outside of it, i. e., $q_i > ue_i$ or $q_i < le_i$. If data point q_i is outside the envelope, LBG adds the DTW distance from q_i to the nearest envelope border, i. e., either ue_i or le_i. In turn, if data point q_i is inside the envelope, LBG adds a distance of 0 for point i. If envelope E encloses a time series Q, a lower bound is 0. A proof is in [NRR10].

LBG is also a valid lower bound on the PAA representations Q^T and E^T of a time series Q and an envelope E [SYF05]. See Equation 3.9.

$$\text{LBG}(Q^T, E^T) \leq \text{LBG}(Q, E) \tag{3.9}$$

This may lead to a less tight lower bound, but may speed up its calculation of the lower bound significantly.

Equation 3.10 shows the definition of the LBG lower bound.

$$\text{LBG}(Q^T, E^T) = \sqrt[p]{D(m, l)} \tag{3.10}$$

$$D(i, j) = T \cdot D_{seg}(q_i^T, e_j^T) + \min \begin{cases} D(i-1, j-1) \\ D(i-1, j) \\ D(i, j-1) \end{cases} \tag{3.11}$$

$$D_{seg}(q_i^T, e_j^T) = \begin{cases} |lq_i^T - ue_j^T|^p & \text{if } lq_i^T > ue_j^T \\ |le_j^T - uq_i^T|^p & \text{if } le_j^T > uq_i^T \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

where $D(0, 0) = 0$ and $D(i, 0) = D(0, j) = \infty$ for $1 \leq i \leq m$ and $1 \leq j \leq l$.

Observe the following properties of LBG. First, a smaller segment size T usually leads to a tighter lower bound. Second, Equation 3.10 is also valid for the LBG calculation on the full dimensionality, i. e., segment size $T = 1$. Third, LBG is also a valid lower bound for the DTW distance of two time series [SYF05]. In this case, envelope E only contains a single time series element. On a segment size of $T = 1$, LBG leads to the exact DTW distance of two time series. Fourth, LBG also works for different segment sizes for Q^{T_1} and E^{T_2} when substituting factor T with $\min(T_1, T_2)$ in Equation 3.11 [SYF05]. This property becomes interesting when studying interval-focused queries.

3.3.6 Cascading Lower Bounds

Rakthanmanon et al. analyze and compare the run times and tightness of lower bounds for the DTW distance [Rak+12]. To achieve a good compromise between run time and pruning factor, they propose to use two lower bounds in a cascade, as follows.

LB_KimFL(Q, C) was published by Kim et al. [KPC01] and modified by Rakthanmanon et al. [Rak+12]. It is the Euclidean distance for the first (F) and last (L) point of time series Q and C, since these points always have an alignment of 0. Kim's lower bound has a run-time complexity of $O(1)$. Thus, it is a very fast lower bound that removes many time series in spite of its looseness.

LB_Keogh(Q, C) was published by Keogh et al. [KR05]. It is the Euclidean distance between a bounding envelope around time series Q and time series C. Its run-time complexity is in $O(n)$, i. e., depends linearly on the length of the time series. Thus, the Keogh lower bound is cheaper than the quadratic run time of DTW, but prunes further candidate time series.

Rakthanmanon et al. additionally use $\max(LB\_Keogh(Q,C), LB\_Keogh(C,Q))$ as third step in their lower bound cascade [Rak+12], since LB_Keogh is not commutative, i. e., $LB\_Keogh(Q,C) \neq LB\_Keogh(C,Q)$.

4 A Query Algebra for Temporal Text Corpora

In this chapter, we design a query algebra to formulate queries for temporal text corpora. Thereby, we aim to provide an interface to examine complex hypotheses and questions in the field of conceptual history. To examine complex hypotheses, a query algebra must provide the functional scope to express information needs in conceptual history. This enables empirical analyses of questions about conceptual history on large amounts of books. Hence, a query algebra to formulate conceptual history hypotheses is the key to turn conceptual history into a data-driven discipline.

Example 4.1 depicts a complex real-world hypothesis that conceptual historians want to analyze.

Example 4.1 *A conceptual historian examines the impact of World War II and the Cold War on the German language. She hypothesizes that the words "Osten" and "Westen" (German for East and West) have acquired a political meaning after 1945. Put differently, both words have changed their semantics from only cardinal directions to also political concepts.*

To examine a complex hypothesis, a conceptual historian splits this hypothesis into a number of specific questions. To examine such questions regarding Example 4.1, a query algebra requires operators to express the usage frequency, co-occurring words, and how words relate to and contrast with each other over time [BCK04; RGG07].[1] Existing query algebras, like the one for the Structured Query Language [Cod70; Mai83; AHV95], have proven their worth, but lack specific support for such analyses. Other approaches from the literature, e. g., the Contextual Query Language [Con13], the Corpus Query Language [Jak+10], or the ANNIS Query Language [Zel+09], have similar issues. In the end, there exists no query language to query word features over time, e. g., co-occurrences.

In this chapter, we propose a query algebra to analyze temporal text corpora. The goal is to provide an algebra that can match all actual hypotheses of conceptual history. We address this problem by designing an algebra that is inspired by the work of Reinhart Koselleck [Ols12] (cf. Section 2.1.1). This chapter describes the outcome of an interdisciplinary collaboration with philosophers working on conceptual history.

Challenges. We have to solve two challenges: data characteristics and showing the completeness of our query algebra. We describe these challenges in more detail in the following.

[1] The remainder of this chapter bases on our article [Wil+18]: Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. 2018. "A Query Algebra for Temporal Text Corpora." In *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries* (*JCDL '18*). DOI: 10.1145/3197026.3197044.

Data Characteristics Koselleck has given intuitive definitions of his concept types that are more concrete for some concept types and more abstract for others. In contrast, when working with real data, we need observable data characteristics that specify the behavior of individual words.

Completeness To guarantee that domain experts can examine the whole conceptual history using our query algebra, we must show its completeness. We call our query algebra complete when one can formulate all potential hypotheses regarding Koselleck's conceptual history.

Contributions. In this section, we present a data model for temporal text corpora and a set of algebra operators that forms our CHQL algebra. Our algebra allows analyzing conceptual history related to arbitrary words and addresses the aforementioned challenges as follows.

We need to overcome the gap between the philosophical descriptions of information needs and their implementation. Therefore, we define a set of data characteristics. Such a data characteristic is a concrete and quantifiable piece of information, e. g., how often has a word been used in a specific year, or which words are used around this word. The philosophers' part is to propose data characteristics that allow to realize all of Koselleck's information needs. Our part, as computer scientists, is to offer data characteristics and suggest its realization. Our algebra is mostly based on these characteristics. We realize every data characteristic with one algebra operator. For example, our operator *surrounding-words* creates a set of words used around a target word. Another example is our *sentiment* operator. It maps every word to an integer that represents the sentiment value of this word.

Our completeness criterion is to cover all of Koselleck's hypotheses. We argue that using his hypotheses is a suitable and meaningful criterion, given his academic standing in the field. To show the completeness of our algebra in turn, we show that it completely covers the information needs to analyze conceptual history.

Finally, we arrive at first novel insights from experimenting with a proof-of-concept implementation.

Outline. We structure this chapter as follows. Section 4.1 presents the information needs to examine conceptual history. We depict the process to develop a query algebra in Section 4.2. We define the data model of our query language in Section 4.3 and list the operators in Section 4.4. Section 4.5 shows a proof of concept in which we formulate and test real-world hypotheses on conceptual history.

4.1 Koselleck's Information Types

Koselleck distinguishes a number of concept types by various criteria. His criteria are, for example, changes in the sentence structure or changes in a word's linguistic context. We denote Koselleck's criteria as *information types* and use these information types as the information needs in conceptual history.

Koselleck does not explicitly give a set of information types he uses for his research. However, experts in this field have extensive knowledge about the information Koselleck uses, and how he argues. We, and the philosophers in particular, now assemble all information types Koselleck considers in his work, as follows.

Geography What is the geographical dispersion of words? In which country are specific words used and which are not?

Conceptual Design Conceptual design means that words can be used as proxies for a more complex topic than their literal meaning. For instance, when talking about the topic war, one often does not mean the single word "war", but also words that belong to war, like "soldiers" and "death".

Context The context refers to the linguistic context. A linguistic context is the set of words that are used around a target word. The context is of special interest for concepts and ambiguous words. Considering the context of a word allows to detect its meanings. This is not restricted to ambiguous words. For distinct words, the context might describe a different view on the same circumstance. Take the word "ecology": It is used at least in political as well as in economic contexts. It has the same meaning, but triggers different targets, namely to take care of people's health—and a restriction to maximize the profit.

Sentence Structure The arrangement of words and phrases defines the structure of a sentence. Keeping track of changes in the arrangement of phrases is an indicator of a social change. For example, the phrase "the history of the farmers" changed to "the history of the trade". This might mean that trade has now taken the role agriculture used to have.

Neologisms A neologism is a word or phrase that either is completely new to a language or is used with a new meaning. It is not necessarily part of the mainstream language. A revolution usually goes along with new words to describe and categorize the new conditions.

World Affairs Conceptual history and the ability to interpret a concept and its meaning require knowledge about world affairs, e. g., changes of economic, political, or social circumstances. Such historical events often trigger a social change which is reflected in the language.

Coverage in the Query Algebra. We decide to not deal with geographical information in our algebra. Using a certain language data set, e. g. the British English or the German one, implicitly contains important geographical information already. We also do not explicitly model world affairs, for two reasons. First, the user needs extensive knowledge about world affairs to correctly interpret the results. Second, a user may be looking for special world affairs and their impact on language. When we explicitly model some events, we would only facilitate the investigation of known events.

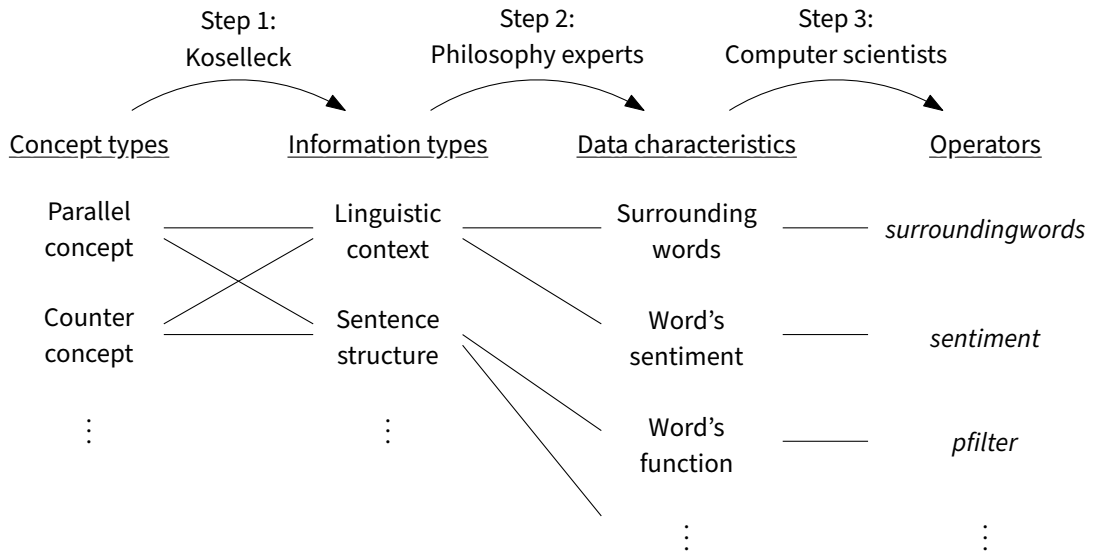


Figure 4.1: The relationship between concept types, information types, data characteristics, and operators.[2]

4.2 From Concept Types to Operators

In this section, we describe how one can search for concept types with the operators which we formally define in Section 4.4. We show that one can search for concept types that follow the definitions of Koselleck [BCK04]. We explain completeness in three steps, which Figure 4.1 illustrates. In Section 4.1, we show that Koselleck has come up with a relationship between concept types and a set of information types. Since Koselleck's specifications of information types are rather abstract, we now describe an interpretation of his information types. This interpretation is an original contribution of our work, based on the expert knowledge of the philosophers in our team. As part of this step, we also describe a mapping of those types to so-called data characteristics. A data characteristic is a quantitative feature either explicitly present in the data, e. g., the usage frequency of word "peace" in 1969, or a derived piece of information, e. g., the difference between the usage frequency of words "peace" and "war" in 1941. In principle, we can create numerous data characteristics from a temporal text corpus. Hence, in the second step, we describe which data characteristics are needed to simulate Koselleck's information needs. In the third step, we explain our realization of all data characteristics and their implementation as operators.

4.2.1 Step 1: From Concept Types to Information Types

One of Koselleck's assumptions is that any concept type has its specific characteristic, i. e., any concept type can be described as a combination of information types. For example,

[2] This figure is based on the one in our article [Wil+19b]: Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. 2019. "The CHQL Query Language for Conceptual History Relying on Google Books." In *Digital Humanities 2019: Book of Abstracts* (*DH '19*). URL: `https://dev.clariah.nl/files/dh2019/boa/0646.html`.

words that form a parallel concept might have a significant number of equal surrounding words. However, for most concept types, he does not specify this relationship explicitly. This means that we cannot directly look for concept types. We need to define operators to find the information types that are indicators for concept types. We give an example for Step 1 in the following.

Example 4.2 *We illustrate a relationship between concept types and information types, using counter concepts as example. Counter concepts shape an asymmetric relation between* us *and* them *[And03].—Examples that are part of the following subsections will continue this example and say what concept types and information types are in this specific case. We use counter concepts as a running example in this section wherewith we illustrate each of the three steps.*

4.2.2 Step 2: From Information Types to Data Characteristics

Every concept type has its own characteristics, e. g., roles and functions in text. Koselleck's information types are a set of such characteristics. If Koselleck's theory holds, one can describe every concept type *ct* as a combination of information types which characterizes *ct*. This combination always is a subset of all of Koselleck's information types. We therefore strive for a system that finds these information types in large corpora. To this end, we need a formal definition of any information type which is observable and quantifiable. We call such a definition *data characteristic*. Our philosophy experts define a mapping from information types to data characteristics. Before defining all our data characteristics, here is an example for Step 2.

Example 4.3 *This example continues Example 4.2. Words that are often used either near the word "us" or "them" potentially are counter concepts. Well-known counter concepts [And03] are:*

- *the bourgeois opposing to the proletarian*
- *the socialist opposing to the liberal*
- *the West opposing to the East*
- *the Protestant opposing to the Catholic*

Hence, one must consider the context of the information type to find indications for counter concepts.

Our Data Characteristics. We now describe our data characteristics which identify each of Koselleck's information types. In Section 4.4, we define one operator for each of these characteristics.

Conceptual Design Words can have single or multiple meanings, i. e., be ambiguous. We realize finding conceptual design by checking whether a word describes a topic additionally to its own meaning. A topic is a group of words that are used to

write about the same issue, e. g. politics or economy. Our experts define the data characteristic of conceptual design as the sum of the usage numbers of all words that belong to a topic. Hence, we propose an operator *topicgrouping* that groups words by their topic and sums their usage numbers. If a word has multiple topics, we sum it into all of these topics.

Context To determine a word's context in Koselleck's spirit requires two data characteristics: (1) a set of surrounding words for a target word, i. e., the linguistic context, and (2) the sentiment for this context, by summing up the sentiment values of the words of the context. Our *surroundingwords* operator and *sentiment* operator implement this.

Sentence Structure One needs to consider two data characteristics: (1) the function of a word, i. e., differentiate between the parts of speech, and (2) completing phrases, i. e., search for missing words in a phrase. The first data characteristic is implemented by our operator *pfilter*. We implement the second one as a pattern-matching operator which we call *textsearch*.

Neologisms The data characteristic of a neologism is an increasing word-usage frequency over time. To find this characteristic, we propose an operator *time series-based selection* that compares the time-series values with a constant. To allow for a temporal restriction, we also provide a *subsequence* operator that limits the selection to an arbitrary time interval. The combination of both operators facilitates the search for neologisms.

4.2.3 Step 3: Finding Data Characteristics with Operators

In the third step, we implement operators to find the specified data characteristic in a large text corpus. This allows to search for any of Koselleck's information types. The following example illustrates Step 3.

Example 4.4 *This example continues Example 4.3. To find potential counter concepts, we apply the* surroundingwords *operator to our data. It returns a set of surrounding words for every word in the corpus. We now use our* textfilter *operator to search these sets of surrounding words for words like "us" or "them". The more often one of these words is found, the stronger is the indication that this word is part of a counter concept.*

If our operators can identify all data characteristics, one can search for any of Koselleck's information types. This means that one can formulate and test all hypothetical relationships between information and concept types. Hence, we claim that our set of operators is complete with respect to the set of possible hypotheses.

4.3 Data Model

In this section, we define the data model behind our query algebra. We then describe additional data sources which would be necessary to realize all information types described

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
Begriffsgeschichte	$\emptyset$	70	54	58
peace	NOUN	312,031	330,389	295,867
war	NOUN	875,479	878,696	873,246
soldier	NOUN	70,196	72,941	72,587

(a) Example elements of set G_1.

$\vec{w} \in W^2$	$\vec{p} \in P^2$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
Reinhart Koselleck	$\emptyset$ $\emptyset$	65	24	19
conceptual history	$\emptyset$ $\emptyset$	37	31	27
modern history	$\emptyset$ $\emptyset$	3,074	3,165	3,459
history modern	$\emptyset$ $\emptyset$	1	6	4
history books	$\emptyset$ $\emptyset$	2,248	2,205	2,333

(b) Example elements of set G_2.

Table 4.1: Example sets for $gram_n$.

in Section 4.2. In the last part of this section, we describe shorthand notations and our data sources.

4.3.1 Ngram

There are three elementary types: words, parts of speech, and time series. We use w to represent an individual word and W for a set of words, e. g., $W = \{\emptyset, peace, war, soldier, \ldots\}$. $\vec{w} \in W^n$ is a vector of n words. For example, $W^2 = \{(conceptual, history), \ldots\}$. A part of speech p is an element of $P = \{\emptyset, NOUN, VERB, ADJ, ADV, PRON, DET, ADP, NUM, CONJ, PRT, X\}$. $\vec{p} \in P^n$ is a vector of parts of speech of length n. For example, $P^2 = \{(ADV, VERB), \ldots\}$. A time series $C \in \mathbb{Z}^l$ is a vector of integer values of length l. Using these basic types, we now define $gram_n$.

Definition 4.1 *A $gram_n$ is a tuple $(\vec{w}, \vec{p}, C) \in W^n \times P^n \times \mathbb{Z}^l$ consisting of a vector of n words, a vector of n parts of speech and a time series of length l. We abbreviate a set of $gram_n$ as G_n.*

Parts of speech are classes of words with similar grammatical properties, e. g., nouns or verbs. The part of speech elements are intuitive, e. g., NOUN denotes a noun and VERB a verb. A full description of the part of speech elements is in [Lin+12]. Table 4.1 shows example elements of sets G_1 and G_2.

Projecting on Single Elements. We define functions that project a $gram_n$ tuple to its components. To access the word vector, we have function *projwords* of type $G_n \rightarrow W^n$, as follows:

$$\text{projwords}(gram_n) := \vec{w} \tag{4.1}$$

w	sentiment
peace	+1
war	-1

Table 4.2: An example of sentiment information.

To access the parts of speech of a $gram_n$ tuple, we have function *projpos* of type $G_n \longrightarrow P^n$:

$$\text{projpos}(gram_n) := \vec{p} \tag{4.2}$$

To access the time series, we have function *projts* of type $G_n \longrightarrow \mathbb{Z}^l$:

$$\text{projts}(gram_n) := C \tag{4.3}$$

4.3.2 Shorthand Notations

We write $gram_n.w$ as shorthand for projwords($gram_n$), to access the word vector $\vec{w}$. We use $gram_n.p$ and $gram_n.C$ to project on the parts of speech element and the time series, respectively. We use the notation w_i to access the i-th element of vector $\vec{w}$. The same holds for p_i and vector $\vec{p}$. We additionally define two shorthand notations to access time-series values. The first one c_y projects a time series to its value of a specific year y. The second notation maps a time series C to a subsequence $\mathbb{Z}^l \longrightarrow \mathbb{Z}^t$ where $0 \leq t \leq l$. The access to a subsequence from year a to year b inclusively is as follows.

$$C[a,b] := \begin{cases} \text{subsequence from } a \text{ to } b & \text{if } a < b \\ \text{subsequence with element } c_a & \text{if } a = b \\ \text{empty subsequence} & \text{else} \end{cases} \tag{4.4}$$

4.3.3 Sentiment and Category

To realize all the information types, we need to combine information from different sources. We need to consider two additional kinds of information: the sentiment value of a word and its category. A word sentiment can be positive, negative, or neutral. To implement such a contrast-word rating, we use a binary word-classification mechanism. A category is a group of words that are used in the same topic, such as religion or economy. To realize this, we need the information which category a word belongs to. We now define both kinds of additional information: the sentiment and the category.

4.3.3.1 Sentiment

In conceptual history, it is important to investigate the relations between concepts, e. g., differences in positive and negative sentiments, or their orientation towards the past or the future. Signal words typically make these differences explicit. To illustrate, in the phrase “the hope for peace” the word “hope” is a word that signals a positive phrasing of the sentence. The additional information is the knowledge about signal words and their

w	category
war	military
soldier	military
military	military

Table 4.3: An example of category information.

positive or negative classification. To define the sentiment value for a word, we define a function *wordsentiment*: $W^n \rightarrow \mathbb{Z}$. The function maps single words as well as word vectors to a sentiment value. For example, the single word "power" is a positive word and may have a sentiment value of +1. The 2-gram "electrical power" in turn does neither have a positive nor a negative sentiment value. The sentiment value for a neutral rating is 0. *wordsentiment* returns integer values to allow modeling different grades of positive and negative sentiments. Table 4.2 lists example words and their sentiments.

4.3.3.2 Category

A category is a word that is used as proxy for a more complex topic than its literal meaning, e. g., the category military includes the word "war" as well as "soldiers" and others. To implement information type *conceptual design*, we need to model the relationship between words and their categories. We define *category* as a function of type $W^n \rightarrow Cat$ that maps words to the category they belong to, where $Cat \subset W^1$ is a set of categories. Every category in Cat is described as a single word of the set of all words W^1. For example, the word "military" is a word ("military" $\in W^1$) and a category ("military" $\in Cat$). The word "soldier" in turn is a word "soldier" $\in W^1$, but does not describe a category "soldier" $\notin Cat$. Table 4.3 shows an example of such a category grouping.

4.3.3.3 Information Sources

We store the sentiment information as well as the category information independent of one specific source. Therefore, we are able to use every source that contains this information and also to replace a source if a better one is available. Due to their expertise regarding text and word interpretation, the philosophers in our team have decided for the following sources. We extract the sentiment information from the LIWC list [Wol+08]. This list is a common reference in computerized text analysis. We use the OpenThesaurus[3] database [Nab05] for the mapping between words and categories.

4.4 Query Operators

Section 4.1 lists a complete set of information types to analyze conceptual history. We now propose a corresponding set of query operators. Some operators realize an entire information type, while one needs a combination of operators to realize other, more

[3] The OpenThesaurus word net is available at `https://www.openthesaurus.de`.

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
military	$\emptyset$	945,675	951,637	945,833

Table 4.4: Example result set of the topic grouping operator.

complex types. This section introduces the operators, gives their definitions and shows how they implement the information types.

4.4.1 Topic Grouping Operator

We realize information type *conceptual design* by defining a topic grouping operator. To realize this abstraction, this operator groups all words from the same topic. The groups have names consisting of one word. Thus, the result is of type $gram_1$. This allows us to define word categories e. g., military or religious, and analyze or compare their developments over time. We first define the set of occurrence topics for a given set of words and afterwards we define our *topicgrouping* operator.

Definition 4.2 *Cat_G is the set of categories that appear in set G.*

$$Cat_G := \bigcup_{g \in G} \{\text{category}(g.\vec{w})\} \tag{4.5}$$

Definition 4.3 topicgrouping *is an operator of type $\mathcal{P}(G_n) \rightarrow \mathcal{P}(G_1)$. It has the following semantics:*

$$\text{topicgrouping}(G) := \bigcup_{cat \in Cat_G} \left(cat, \emptyset, \sum_{\{g \in G | \text{category}(g.\vec{w}) = cat\}} g.C \right) \tag{4.6}$$

The topicgrouping operator groups all ngrams by their topic and sums together their time series.

Example. This example, as well as the ones that follow, use the data in Tables 4.1, 4.3, and 4.2. *topicgrouping* sums up the time series of all words belonging to category military. The result is a time series with the number of times authors have written about a military topic. For our example data, the operator yields the result in Table 4.4.

4.4.2 Surrounding Words Operator

When computing the context of words, a *target word* is the word we determine the context for, whereas *context words* are the words used around the target word. Before defining the operator, we define two auxiliary functions. The first one maps a word vector to the set of words. The second function maps a word vector to a word vector of a higher dimensionality that includes the same elements as the original word vector. We first define the help functions *split* and *expandwords* and secondly use these functions to define our *surroundingwords* operator.

Definition 4.4 split *is a function of type* $W^n \rightarrow \mathcal{P}(W^1)$.

$$\text{split}(w \in W^n) := \{w_1, w_2, \ldots, w_n\} \tag{4.7}$$

split maps a vector of n words to a set of n words.

Definition 4.5 expandwords *is a function of type* $\mathbb{N} \times W^n \rightarrow \mathcal{P}(W^m)$ *with* $n < m; n, m \in \mathbb{N}$.

$$\text{expandwords}(m, \vec{q} \in W^n) := \left\{ (w_1, w_2, \ldots, w_m) \in W^m \,\middle|\, \exists o : \bigwedge_{i=1}^{n} w_{i+o} = q_i \right\} \quad \text{with } 0 \leq o \leq m - n \tag{4.8}$$

expandwords maps a vector of words to a set of vectors that are of higher dimensionality and contain the original word vector. This includes word vectors with new words in the front or in the back or both. Parameter m defines the dimensionality of the target vectors. Function *expandwords* adds $m - n$ words to the input vector. For example, we expand the word vector "history" of length $n = 1$ to a word vectors of length $m = 2$. A partial result are the word vectors "the history", "conceptual history", or "history of".

Definition 4.6 surroundingwords *is an operator of type* $\mathbb{N} \times G_n \rightarrow \mathcal{P}(G_1)$. *It has the following semantics:*

$$\text{surroundingwords}(m, g \in G_n) := \bigcup_{i \in \text{expandwords}(m, g.\vec{w})} \bigcup_{j \in \text{split}(i)} \{\text{projwords}^{-1}(j)\} \tag{4.9}$$

The *surroundingwords* operator returns a set of context words occurring together with at least one of the target words in a window of size m. We split the description into three steps. First, we filter all multigrams of size m for those that contain at least one target word vector. Second, we split these multigrams into single words and, third, we add the corresponding usage-frequency time series to the context words.

Here are the three steps in more detail.

1. The help function *expandwords* selects all word vectors of size m which contain the target word vector $\vec{q}$, no matter at which position of the window it occurs. *expandwords* returns a set of word vectors.

2. Having the set of word vectors that contain the target word vector, we split these word vectors into single words. To perform this splitting, we need function *split*. This results in a set of single words.

3. To return elements of type $gram_1$ instead of single words, we use the inverse function projwords^{-1} to get the $gram_1$ elements that contain the corresponding word. We do this for every single word that function *split* returns. The result is the union over all $gram_1$ elements of all surrounding words.

Example. Suppose that we want all surrounding words for the word "history" within a window of size $m = 2$. Then we get the result shown in Table 4.5.

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
conceptual	$\emptyset$	75,586	78,319	84,518
modern	$\emptyset$	523,599	510,492	532,338
books	$\emptyset$	447,885	436,655	462,202

Table 4.5: Example result set of the surroundingwords operator.

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
peace	NOUN	+312,031	+330,389	+295,867
war	NOUN	-875,479	-878,696	-873,246

Table 4.6: Example result set of the sentiment operator.

4.4.3 Sentiment Operator

Another important information for conceptual history is a word's sentiment, i. e., the positive or negative emotions associated with a word. This operator can be used to determine the sentiment for a single word or for a set of context words. This operator is the second part to completely cover the *context* information types from Section 4.1.

Definition 4.7 sentiment *is an operator of type* $G_n \rightarrow G_n$*. It has the following semantics:*

$$\text{sentiment}(g \in G_n) := (g.\vec{w}, g.\vec{p}, g.C \cdot \text{wordsentiment}(g.\vec{w})) \tag{4.10}$$

This operator multiplies the usage frequency of a word by the word's sentiment value. This multiplication takes both into account, the word's sentiment value and the word's usage frequency in the text. The sentiment operator changes the meaning of the time-series values. The time-series values then no longer stand for the usage frequency of the word, but its sentiment over time. More precisely, the time-series values represent the sentiment value of an ngram over time.

The sentiment operator is defined for all lengths of ngrams. However, it only matches the same ngram length that is specified in the sentiment mapping, but not shorter or longer ones. For instance, the sentiment for "war" only matches the 1-gram "war", but not the 2-gram "civil war". This allows a fine-grained definition of sentiment values, as it becomes possible to have different sentiment values for "War", "Civil War", "First World War", "Second World War" and "Cold War".

Example. Table 4.6 shows an example result of *sentiment*.

$\vec{w} \in W^2$	$\vec{p} \in P^2$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
history modern	∅ ∅	1	6	4
history books	∅ ∅	2,248	2,205	2,333

Table 4.7: Example result set of the textsearch operator.

4.4.4 Text Search Operator

Having a large set of words, a fundamental need is to search for the words conceptual historians might be interested in and which matches some pattern pt. Pt is short for a set of patterns. $Z = \{\exists, \forall\}$ is the set of standard quantifiers.

Definition 4.8 textsearch *is an operator of type* $Pt \times Z \times \mathcal{P}(G_n) \rightarrow \mathcal{P}(G_n)$. *It maps its input as follows:*

$$\text{textsearch}(pt, \zeta, G_n) := \{g \in G_n \mid \zeta i : g.w_i \textit{ matches } pt\} \tag{4.11}$$

The operator selects all tuples that satisfy the given condition with $\zeta \in \{\exists, \forall\}$ as quantifier and pt as a search pattern. The quantifier controls whether all words need to match pt or just one of them.

Example. When searching for the pattern `histo*` the *textsearch* operator returns the set in Table 4.7.

4.4.5 Part of Speech Filtering

Analyses in conceptual history often only refer to specific parts of speech, such as nouns. So we propose a filter to keep only ngrams having a specific part of speech, e. g., it allows to select all nouns.

Definition 4.9 pfilter *is an operator of type* $P^n \times Z \times \mathcal{P}(G_n) \rightarrow \mathcal{P}(G_n)$. *It maps its input as follows:*

$$\text{pfilter}(p, \zeta, G_n) := \{g \in G_n \mid \zeta i : g.p_i = p\} \tag{4.12}$$

Like in the definition of the textsearch operator, we use the quantifier $\zeta \in \{\exists, \forall\}$ to match the part of speech for at least one element or for all elements. In contrast to the textsearch operator, we can only search for part of speech tags.

Example. Selecting all nouns returns a set like the one in Table 4.8.

4.4.6 Time Series-based Selection

We may want to exclude words with extreme time-series values from our analysis, e. g., very rarely used words, because they might falsify our results. So we need a filter for time-series characteristics. $\Theta = \{<, \leq, =, \neq, \geq, >\}$ is the set of standard comparison operators.

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
peace	NOUN	312,031	330,389	295,867
war	NOUN	875,479	878,696	873,246

Table 4.8: Example result set of the pfilter operator.

$\vec{w} \in W^1$	$\vec{p} \in P^1$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
peace	NOUN	312,031	330,389	295,867
war	NOUN	875,479	878,696	873,246

Table 4.9: Example result set of the time series-based selection.

Definition 4.10 cond *is a tuple* $(\zeta, \theta, v) \in Z \times \Theta \times \mathbb{Z}$ *of conditions* Cond.

Definition 4.11 $F_{cond} \subseteq \mathbb{Z}^l$ *is a set of time series that fulfill condition* cond.

$$F_{(\zeta,\theta,v)} := \{C \in \mathbb{Z}^l \mid \zeta i : c_i \, \theta \, v\} \tag{4.13}$$

Definition 4.12 tsselection *is an operator of type* $Cond \times \mathcal{P}(G_n) \rightarrow \mathcal{P}(G_n)$. *It maps its input as follows:*

$$\text{tsselection}(cond, G_n) := \{g \in G_n | g.C \in F_{cond}\} \tag{4.14}$$

The time series-based selection operator selects $gram_n$ elements based on their time-series properties.

Example. Condition $cond = (\forall, >, 1000)$ is the condition to select all ngrams whose usage frequency is larger than 1,000 for every available year. The resulting set is shown in Table 4.9.

4.4.7 kNN Operator

Another information that is of interest for analyzing conceptual history is to find words with a similar evolution of usage frequency, sentiment, or surrounding words. This leads us to the definition of a kNN operator, which returns the k ngrams having the most similar time series to a given ngram. This information type is part of analyzing the *grammatical structure*.

Definition 4.13 kNN *is an operator of type* $\mathbb{N} \times G_n \times \mathcal{P}(G_n) \rightarrow \mathcal{P}(G_n)$. *It maps its input as follows:*

$$\text{kNN}(k, q, G_n) := \underset{\{S \in \mathcal{P}(G_n) : |S| = k\}}{\arg\min} \max_{s \in S} \|s.C - q.C\|_2 \tag{4.15}$$

$\vec{w} \in W^2$	$\vec{p} \in P^2$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
conceptual history	$\emptyset$ $\emptyset$	37	31	27

Table 4.10: Example result set of the kNN operator.

$\vec{w} \in W^2$	$\vec{p} \in P^2$	$C \in \mathbb{Z}^2$	
		1980	1981
Reinhart Koselleck	$\emptyset$ $\emptyset$	65	24
conceptual history	$\emptyset$ $\emptyset$	37	31

Table 4.11: Example result set of the subsequence operator.

Our kNN operator implements a kNN search based on the time-series values of the $gram_n$ elements. Input k defines the number of resulting elements, i. e., the number of nearest neighbors to search for. q defines the target $gram_n$ to search the neighbors for. Input set G_n defines the search space, i. e., the set of possible results. The kNN operator can be used with different distance measures. For example, our implementation currently supports the Euclidean distance and dynamic time warping.

Example. Searching for the $k = 2$ nearest neighbors of the target word "Reinhart Koselleck" yields the result in Table 4.10.

4.4.8 Subsequence Operator

Most time-specific information has a specific start or end date or is in a specific temporal range of historical events, e. g., in the period from 1945 to 1990. However, the operators above work over the whole time range. Hence, we need an operator that reduces time series to a certain time interval.

Definition 4.14 subsequence *is an operator of type* $G_n \times \mathbb{N} \times \mathbb{N} \rightarrow G_n$*. It maps its input as follows:*

$$\text{subsequence}(g, a, b) := (g.\vec{w}, g.\vec{p}, g.C[a, b]) \tag{4.16}$$

The subsequence operator has three parameters: the $gram_n$ element g, the start year a and the end year b. It takes the input ngram element and reduces its time series to the time window $[a, b]$.

Example. When applying function *subsequence* to consider only the years from 1980 to 1981, the result looks like the set in Table 4.11.

$\vec{w} \in W^2$	$\vec{p} \in P^2$	$C \in \mathbb{Z}^3$		
		1980	1981	1982
peace	NOUN	+312,031	+330,389	+295,867
war	NOUN	+875,479	+878,696	+873,246

Table 4.12: Example result set of the absolute operator.

4.4.9 Absolute Value Operator

Using the sentiment operator allows generating positive and negative time-series values representing positive or negative word emotions. For some investigations of conceptual history, one needs to quantify word's emotions, no matter whether they are positive or negative. To archive this, we first apply the sentiment operator and secondly the absolute value operator that maps all sentiment values to positive values, i. e., the absolute value of the sentiment value.

Definition 4.15 tsabs *is a function of type* $\mathbb{Z}^l \rightarrow \mathbb{N}_0^l$*, as follows:*

$$\text{tsabs}(C) := (|c_1|, |c_2|, \ldots, |c_l|) \tag{4.17}$$

tsabs maps every value of a given time series to its absolute value.

Definition 4.16 absolute *is an operator of type* $G_n \rightarrow G_n$*. It maps its input as follows:*

$$\text{absolute}(g) := (g.\vec{w}, g.\vec{p}, tsabs(g.C)) \tag{4.18}$$

The absolute value operator turns every time-series value of a $gram_n$ element into its absolute value. Function *tsabs* maps the values for every single year to its absolute value and creates a new time series with absolute values.

Example. Applying this operator to the previous set yields the result in Table 4.12.

4.4.10 Count Operator

When working with huge data sets, which is preferred for statistical analysis, for many queries a human user simply cannot look at all elements or count them manually. The count operator counts the number of elements of a given set.

Definition 4.17 count *is an operator of type* $\mathcal{P}(G_n) \rightarrow \mathbb{N}$*. It maps its input as follows:*

$$\text{count}(G \in \mathcal{P}(G_n)) := |G| \tag{4.19}$$

Given an input set, *count* counts its elements and returns the number of containing elements.

Example. When we want to count the number of nouns in our data set in Table 4.1, we first filter for nouns, using the *pfilter* operator, followed by the *count* operator. Table 4.13 shows the result.

integer value
3

Table 4.13: Example result of operator count.

$C \in \mathbb{Z}^3$		
1980	1981	1982
−563, 448	−548, 307	−577, 379

Table 4.14: Example result of operator sumup.

4.4.11 Sumup Operator

There is often the need to sum up the time series of ngrams, e. g., to compare the frequency of all nouns or adjectives. The sumup operator gets a set of ngrams, sums up their time series and returns the resulting time series. In comparison to the topicgrouping operator, the sumup operator does not group the elements, but sums up all received time series.

Definition 4.18 sumup *is an operator of type* $\mathcal{P}(G_n) \rightarrow \mathbb{Z}^l$. *It maps its input as follows:*

$$\text{sumup}(G \in \mathcal{P}(G_n)) := \sum_{g \in G} g.C \tag{4.20}$$

The sumup operator iterates over the given set of ngrams and sums together their time series. In the case of summing up sentiment time series, the operator may return a time series containing negative values.

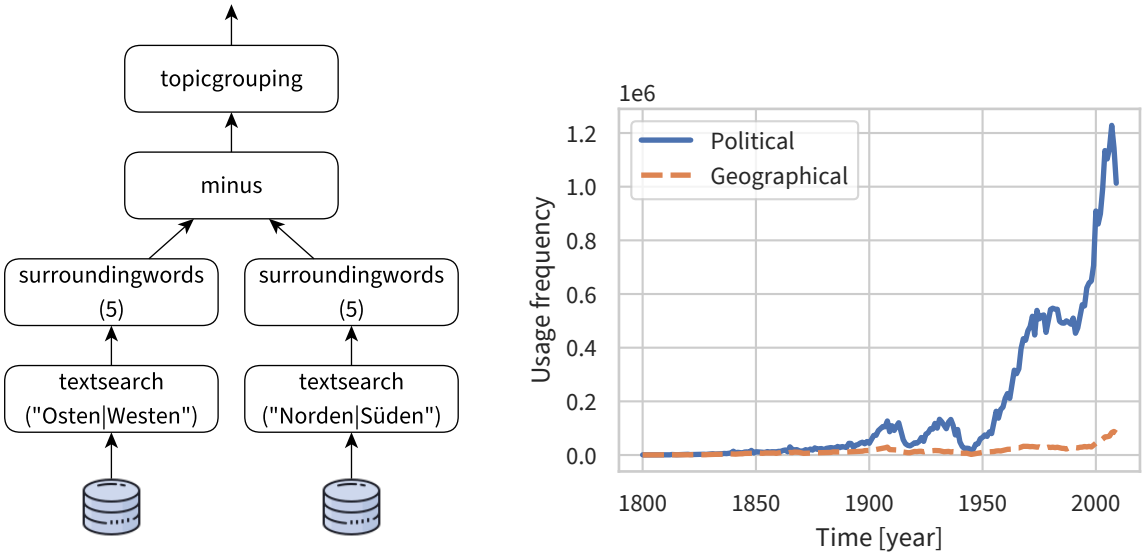
Example. The result set when applying this operator to the example set from the sentiment operator looks like the one in Table 4.14.

4.4.12 Set Operators

Since we are working on sets, we are able to use the set operators intersection ($\cap$), union ($\cup$), and minus ($\setminus$). The key on which these operators work is a combination of the word vector w and the corresponding part of speech, i. e., vector $\vec{p}$. The time-series attribute of the result set is taken from the left input set.

4.4.13 Algebraic Expressions

So far, we have introduced individual operators. In order to formulate complex hypotheses revealing novel insights for conceptual history, one generally needs to combine operators. For instance, in case one wants to find the k-nearest nouns to some given ngram in a specific time period, we can combine the existing operators to formulate an expression to compute the desired result. Operators are compatible if the output of the first operator is of the same type as the input of the second operator. Most of our operators result in a set of ngram tuples, but not all of them. There are two operators that do not result in such a set: *count* and *sumup*. Both operators yield a single value. These operators usually are the final operation in an algebraic expression, e. g., summing up the input tuples. How to come from a philosophical hypothesis to concrete queries, and how example results may look like is the topic of the next section.



(a) The operator tree to realize the expression to test Hypothesis H1.

(b) The result when executing operator tree to test Hypothesis H1.

Figure 4.2: Testing Hypotheses H1.

4.5 Proof of Concept

Our query algebra allows (1) hypotheses testing and (2) hypotheses engineering. We explain both in this section.

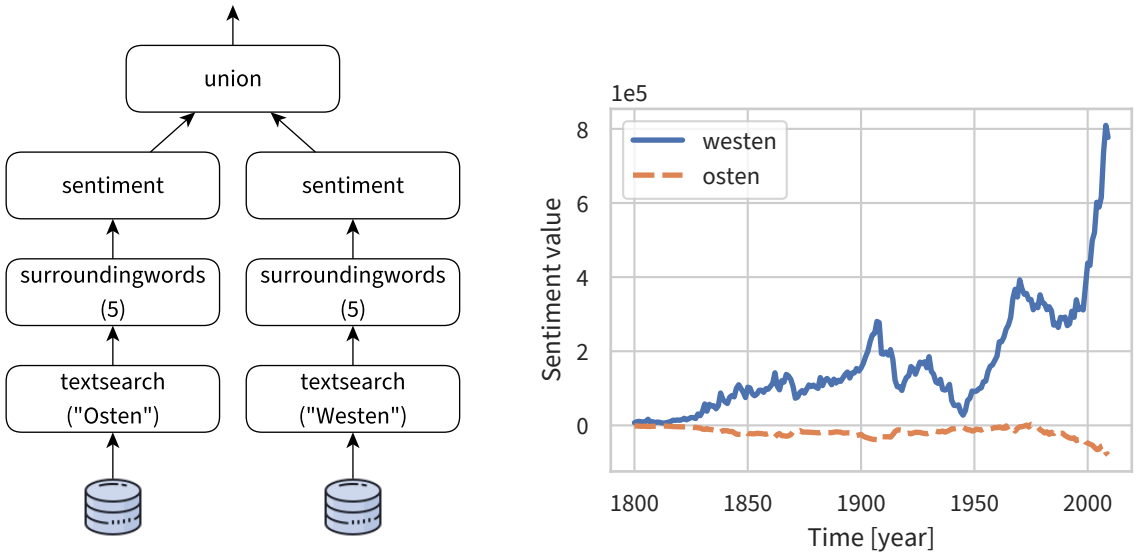
4.5.1 Hypotheses Testing

Hypotheses testing enables scholars to empirically test existing hypotheses regarding properties of different concept types. For example, scholars want to test a hypothesis that characterizes parallel concepts. This consists of the following steps: First, they formulate their hypothesis, i. e., translate it to an algebraic expression. Second, they let a machine evaluate it on a huge corpus. Third, they interpret the results. To illustrate, think of the two hypotheses:

Hypothesis H1 The words "Osten" and "Westen" (German for East and West) have acquired a political meaning, i. e., a political context, during the Cold War, while the words "Norden" and "Süden" (German for North and South) have not changed their merely geographical meaning.

Hypothesis H2 The two words "Osten" and "Westen" have developed into contrasting classes, i. e., have changed from parallel concepts to counter concepts.

Formulate Hypothesis H1. To formulate Hypothesis H1, we use several operators, see Figure 4.2a for the operator tree. First, we need the text-search operator for occurrences of the four cardinal directions. We then generate the common context of "Osten" and "Westen" as well as the one of "Norden" and "Süden". To remove the context words related



(a) The operator tree to realize the expression to test Hypothesis H2.

(b) The result when executing operator tree to test Hypothesis H2.

Figure 4.3: Testing Hypotheses H2.

to cardinal directions, we subtract the common context of "Norden" and "Süden" from the one of "Osten" and "Westen". This leads to the context of "Osten" and "Westen" that is not related to cardinal directions. In a final step, we categorize the remaining context words. Figure 4.2b visualizes the result of computing this operator tree.

Formulate Hypothesis H2. Given the validation of Hypothesis H1 that the words "Osten" and "Westen" become a political context, in Hypothesis H2 we check whether the semantics of the context for these two words develop into contrasting directions, i. e., if one word gets a positive context while the other one gets a negative one. We again start using our text-search operator to select the appropriate occurrences. We then separately generate the context for "Osten" and for "Westen" and perform a sentiment analysis. The operator tree to test Hypothesis H2 is shown in Figure 4.3a. Figure 4.3b visualizes the result. The result shows that the word "Westen" is used in a context mainly consisting of positive words while the words around "Osten" sum up to a negative sentiment. There can be two reasons for this: (1) The word "Westen" has more positive surrounding words that overcome the negative ones, or (2) the word "Osten" misses some surrounding words that the word "Westen" has and therefore sums to a negative sentiment. One can test these two possible reasons with a corresponding expression in our algebra.

4.5.2 Hypotheses Engineering

Hypotheses engineering means developing criteria and hypotheses to distinguish between concept types, i. e., speculating about their differences, formulating and executing algebraic expressions, and interpreting the result. We, especially the philosopher team, are interested in checking hypotheses to get a better understanding of the differences between parallel

and counter concepts. From close reading, they already have some hypotheses regarding the characteristics of parallel concepts. These hypotheses relate to the usage frequency, the surrounding words, and the sentiment values of words. We have formulated and tested all these hypotheses with the outcome that we could not confirm any of them. Our conceptual history experts then have had a closer look at the results of the expression. They have observed that some queries produce better results than others, i. e., contain more parallel concepts. This confirms that our approach gives access to important information for conceptual history experts to develop new hypotheses. At the end of this process, our experts have been able to identify important information types of a parallel concept, as follows: A parallel concept features two concepts whose surrounding words have a similar sentiment, and they share numerous surrounding words.

4.6 Summary

In this chapter, we design a query algebra for temporal text corpora. In contrast to existing query languages and algebras, our query algebra paves the way to formulate hypotheses about conceptual history. As part of our query algebra, we define a data model for temporal text corpora. In cooperation with domain experts, we identify primary information needs in conceptual history and study how this information manifests in a temporal text corpus. Based on our findings, we define a set of algebraic operators that reflects this information. In this way, our operators satisfy all information needs that Koselleck used in his works. Furthermore, our operators can be combined to expressions to represent arbitrary complex queries. In a proof of concept, we show that our query algebra allows to formulate important hypotheses in conceptual history. In addition, we test two hypotheses on the Google Books Ngram Corpus and discuss the results together with domain experts. Our query algebra is an essential building block to turn conceptual history into a data-driven discipline.

5 Accurate Cardinality Estimation of Co-occurring Words

In this chapter, we describe accurate cardinality estimation of co-occurring words to improve query optimization on large corpora. The declarative way to write queries and the existence of semantically equivalent algebraic expressions make room to generate different query plans, i. e., different ways to create the desired result that are differently expensive. Modern information systems provide a query optimizer that searches for the query plan that produces the lowest execution costs. To estimate these costs, one key factor of cost-based optimizers is cardinality estimation [Wu+13; Lei+15]. Such cardinality estimators determine the expected size of an intermediate result and the number of elements in a database that corresponds to a selection predicate. Since query optimization substantially relies on cardinality, finding a good query execution plan requires accurate cardinality estimation.

The estimation methods differ depending on both the data type, e. g., numerical attributes [KM10; MMK18] and textual attributes [KVI96; CGG04; SL19], and the operation to be estimated. In conceptual history, one essential kind of query is to analyze co-occurrences, i. e., two words alongside each other in a certain order [MS99; Min+12]. This is of special interest because changes in co-occurrences indicate changes in the word semantics [Kro15]. Example 5.1 illustrates such kinds of questions.

Example 5.1 *We continue Example 4.1. A conceptual historian examines the semantic change of the words "east" and "west" in the 20th century. To obtain more detailed information about this change, she studies the co-occurrences of both words. This leads to questions of the following kind:*

1. *What were the co-occurrences of "west" in the 20th century?*
2. *How did the co-occurrences of "west" change during the 20th century? In other words, which of its co-occurring words came up or disappeared in this period?*
3. *What words co-occur with "east", but not with "west"?*

Queries that select co-occurring words are dubbed *co-occurrence queries*. In CHQL, these are queries that contain the *surroundingwords* operator (cf. Section 4.4.2). Co-occurrence queries use the output of this operator as input for analyses. For example, such analyses are to compare co-occurrences for various words or various time intervals.

Optimizing a co-occurrence query is difficult.[1] A particularity of co-occurrences is that they only exist in chains of several words (word chain), like ngrams or phrases. This calls for cardinality estimates on a set of word chains. Compared to individual words, word chains form significantly longer strings, with a higher variance in their lengths. Our focus in this chapter is on the accuracy of such estimates with little memory consumption at the same time.

One approach to index string attributes is to split the strings into trigrams, i. e., break them up into sequences of three characters [AM93; KVI96; CGG04]. This seems to work well to index individual words. However, the trigram approach will not reflect the connection between words that are part of a word chain. Another method to index string attributes is the suffix tree [KVI96; Li+15a]. Since suffix trees tend to be large, respective pruning techniques have been proposed. A common technique is to prune the tree to a maximal depth [KVI96]. Since the size of a suffix tree depends on the length of the strings [Sad07], other pruning methods work similarly. We refer to approaches that limit the depth of the tree as *horizontal pruning*. With horizontal pruning however, all branches are shortened to the same length. So horizontal pruning takes away more information from long strings than from short ones. This leads to poor estimation accuracy for long strings and more uncertainty compared to short ones.

Challenges. Designing a pruning approach for long strings faces the following challenges. First, the reduction of the tree size should be independent of the length of the strings. Second, one needs to prune, i. e., remove information, from both short and long strings to the same extent rather than only trimming long strings. Third, the pruning approach should provide a way to quantify the information loss or, even better, provide a method to correct estimation errors.

Contributions. In this chapter, we propose what we call *vertical pruning*. In contrast to horizontal pruning that reduces any tree branch to the same maximal height, vertical pruning aims at reducing the number of branches, rather than their length. The idea is to map several strings to the same branch of the tree, to reduce the number of branches and nodes. This reduction of tree branches makes the tree *thinner*. So we dub the result of pruning with our approach *Thin Suffix Tree* (TST). A TST removes characters from words based on the information content of the characters. We propose different ways to determine the information content of a character based on empirical entropy and conditional entropy. We also present an approach to correct estimation errors. At build time, TST counts the number of suffixes merged into a node while, at query time, TST splits the merged cardinality counts on the merged suffixes. Our evaluation shows that our pruning approach reduces the size of the suffix tree depending on the character distribution in natural language (rather than depending on the length of the strings). TST prunes both short and long strings to the same extent. Our evaluation also shows that TST provides

[1] The remainder of this chapter bases on our article [WSB21]: Jens Willkomm, Martin Schäler, and Klemens Böhm. 2021. "Accurate Cardinality Estimation of Co-occurring Words Using Suffix Trees." In *Proceedings of the 26th International Conference on Database Systems for Advanced Applications* (*DASFAA '21*). DOI: 10.1007/978-3-030-73197-7_50.

significantly better cardinality estimations than a pruned suffix tree when the tree size is reduced by 60 % or less. Due to the redundancy of natural language, TST yields hardly any error for tree-size reductions of up to 50 %.

Outline. We structure this chapter as follows. We introduce the thin suffix tree in Section 5.1. We say how to correct estimation errors in Section 5.2. We describe the procedures *insert* and *query* in Section 5.3. Our evaluation is in Section 5.4.

5.1 The Thin Suffix Tree

To store word chains as long strings in a suffix tree efficiently, we propose TST, a novel pruning technique for suffix trees. In contrast to horizontal pruning approaches, it aims at reducing the number of branches in the tree, rather than their length. We refer to this as *vertical pruning*. The idea is to conflate branches of the tree to reduce its memory consumption. This means that one branch stands for more than one suffix. As usual, the degree of conflation is a trade-off between memory usage and accuracy. In this section, we (1) present the specifics of TST and (2) define interesting map functions that specify which branches to conflate.

5.1.1 Our Vertical Pruning Approach

To realize the tree pruning, we propose a map function that discerns the input words from the strings inserted into the tree. For every input word (preimage) that one adds to the tree, the map function returns the string (image) that is actually inserted. TST stores the image of every suffix. This map function is the same for the entire tree, i. e., we apply the same function to any suffix to be inserted (or to queries). Tree thinning occurs when the function maps several words to the same string. The map function is surjective.

Fixing a map function affects the search conditions of the suffix tree. A suffix tree and suffix tree approximation techniques usually search for exactly the given suffix, i. e., all characters in the given order. Our approximation approach relaxes this condition to words that contain the given characters in the given order, but may additionally contain characters that the map function has removed. For example, instead of querying for the number of words that contain the exact suffix `mnt`, one queries the number of words that contain the characters `m`, `n`, and `t` in this order, i. e., a more general pattern. We implement this by using a map function that removes characters from the input strings. We see the following two advantages of such a map function: (1) Branches of similar suffixes conflate. This reduces the number of nodes. (2) The suffix string becomes shorter. This reduces the number of characters. Both features save memory usage.

5.1.2 Character-removing Map Functions

A character-removing map function removes characters from a given string.

Definition 5.1 *A* character-removing map function *is a function that maps a preimage string to an image string by removing a selection of specific characters.*

To remove characters systematically, we consider the information content of the characters. According to Shannon's information theory, common characters carry less information than rare ones [Bro+92; MGR98]. This is known as Shannon entropy of the alphabet. The occurrence probability $P(c)$ of a character c is its occurrence frequency relative to the one of all characters of the alphabet Σ.

$$P(c) = \frac{\text{frequency}(c)}{\sum_{\sigma \in \Sigma} \text{frequency}(\sigma)} \tag{5.1}$$

The information content of a character c is inversely proportional to its occurrence probability $P(c)$.

$$\text{I}(c) = \frac{1}{P(c)} \tag{5.2}$$

According to Zipf's law, the occurrence probability of each c is inversely proportional to its rank in the frequency table [SEW04].

$$P(c) \sim \frac{1}{\text{rank}(c)} \tag{5.3}$$

Thus, the information content of character c is proportional to its rank.

$$\text{I}(c) \sim \text{rank}(c) \tag{5.4}$$

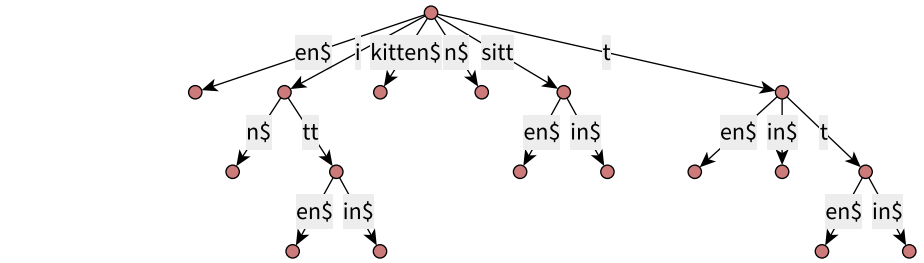
To create a tree of approximation level z, a map function removes the z most frequent characters in descending order of their frequency. Example 5.2 shows this for approximation level 3. Example 5.3 illustrates the effect of a character-removing map function on a suffix tree, especially on the memory requirement.

Example 5.2 *The characters* `e`, `t`, *and* `a` *(in this order) are the three most frequent characters in English text. At approximation level 3, a map function maps the input word* `requirements` *to the string* `rquirmns`.

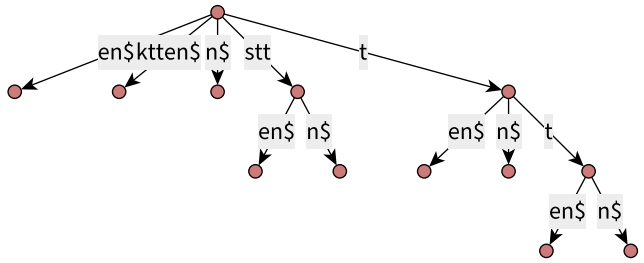
Example 5.3 *Figure 5.1a shows the full suffix tree for the words* `kitten`, `sitten`, *and* `sittin`. *Its size is 288 bytes. The exact result of the regular expression* `*itten` *is 2 counts. For illustrative purposes, we first show the impact of a map function that removes character* `i` *and of a second function that removes* `i` *and* `e`. *Figure 5.1b shows the first approximation level, i. e., character* `i` *removed. The tree is of size 224 bytes, i. e., it saves 22 % of memory used. When we query this tree for the regular expression* `*itten`, *the result is 2 counts. The tree still estimates correctly. Figure 5.1c shows a tree with character* `e` *removed additionally. It has a size of 192 bytes and, thus, saves 33 % of memory usage. When we query the regular expression* `*itten`, *the result is 3 counts. So it now overestimates the cardinality.*

5.1.3 More Complex Map Functions

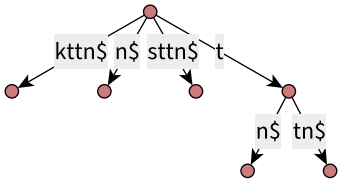
When removing characters of the input words, one can also think of more complex map functions. We see two directions to develop such more complex functions: (1) Consider character chains instead of single characters or (2) respect previous characters. We will discuss both directions in the following.



(a) A full suffix tree for the words `kitten`, `sitten`, and `sittin`.



(b) The tree when using a map function that removes `i`.



(c) The tree when using a map function that removes `i` and `e`.

Figure 5.1: The impact of character-removing map functions on a suffix tree.

Removing Character Chains. The map function considers the information content of combinations of several consecutive characters, i. e., character chains. Take a string $w = c_1c_2c_3$ that consists of characters c_1, c_2, and c_3. We consider character chains of length o and create a frequency table of character chains of this length. The information content of a character chain $c_1 \dots c_o$ is proportional to its rank.

$$\mathrm{I}(c_1 \dots c_o) \sim \mathrm{rank}(c_1 \dots c_o) \tag{5.5}$$

Our character-chain-removing map function is a character-removing map function that removes every occurrence of the z most frequent character chains of the English language of length o. See Example 5.4.

Example 5.4 *A character-chain-removing map function removing the chain* `re` *maps the input* `requirements` *to* `quiments`.

Using Conditional Entropy. We define a character-removing map function that respects one or more previous characters to determine the information content of a character c_1. Given a string $w = c_0c_1c_2$, instead of using the character's occurrence probability $P(c_1)$, a conditional-character removing map function considers the conditional probability $P(c_1|c_0)$. Using Bayes' theorem, we can express the conditional probability as follows.

$$P(c_1|c_0) = \frac{\mathrm{frequency}(c_0c_1)}{\mathrm{frequency}(c_0)} \tag{5.6}$$

Since the frequencies for single characters and character chains are roughly known (for the English language for instance), we compute all possible conditional probabilities beforehand. So we can identify the z most probable characters with respect to previous characters to arrive at a tree approximation level z. See Example 5.5.

Example 5.5 *A map function removes* `e` *if it follows* `r`*. Thus, the function maps the input word* `requirements` *to* `rquirments`*.*

5.1.4 A General Character-removing Map Function

In this section, we develop a general representation of the map functions proposed so far, in Sections 5.1.2 and 5.1.3. All map functions have in common that they remove characters from a string based on a condition. Our generalized map function has two parameters:

Observe The length of the character chain to observe, i. e., the characters we use to determine the entropy. We refer to this as o. Each map function requires a character chain of at least one character, i. e., $o > 0$.

Remove The length of the character chain to remove. We refer to this as r with $0 < r \leq o$. For $r < o$, we always remove the characters from the right of the chain observed. In more detail, when observing a character chain c_0c_1, we determine the conditional occurrence probability of character c_1 using $P(c_1|c_0)$. Therefore, we remove character c_1, i. e., the rightmost character in the chain observed.

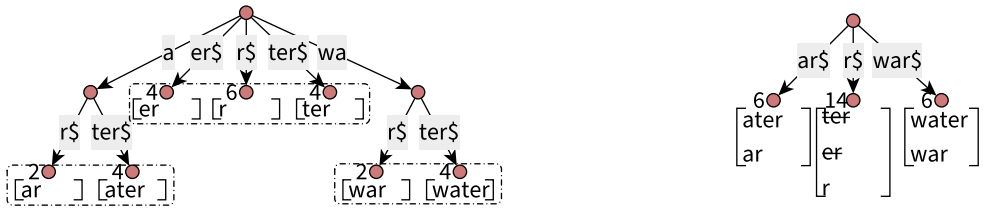
Our generalized map function allows to simulate all character-removing map functions as follows. We parameterize our generalized map function to observe a single character and, if necessary, remove a single character, i. e., $o = r = 1$. To simulate a character-chain-removing function, we observe and if necessary remove the same number of characters, i. e., $o > 1$ and $r = o$. We simulate a character-removing map function that respects one or more previous characters by observing more characters than we potentially remove, i. e., $o > r$. To remove a single character depending on the two previous characters, we use $o = 3$ and $r = 1$. Example 5.6 illustrates this.

Example 5.6 *To remove character* `t` *if it occurs after the characters* `m`*,* `e`*, and* `n` *(in this order), we specify a map function that observes character chains of length four ($o = 4$) and removes individual characters ($r = 1$). So this function takes the previous $4 - 1 = 3$ characters into account to decide whether to remove character* `t` *or not. Technically, the map function searches for the four-digit character chain* `ment` *and replaces it with the chain* `men`*.*

We refer to a map function that observes and removes single characters as `o1r1`, to ones that additionally observe one previous character and remove one character as `o2r1`, and so on. The map function in Example 5.6 is `o4r1`.

5.1.5 Cases of Approximation Errors

Character-removing map functions may lead to two sources of approximation error, by (1) conflating tree branches and (2) by the character reduction itself.



(a) A complete suffix tree for the words `water` (4 times) and `war` (2 times).

(b) The tree when using a map function that removes characters `e` and `t`.

Figure 5.2: Each node of the thin suffix tree includes a suffix count and, additionally to calculate the correction factor, a Bloom filter.

Case 1: Branch Conflation. In most cases, when removing characters, words still map to different strings and, thus, are represented in separate tree nodes. But in some cases, different words map to the same string and these words correspond to the same node. For example, when removing the characters `e` and `t`, the map function maps `water` and `war` to the same string `war`. The occurrences of the two words are counted in the same node, the one of `war`. Thus, the tree node of `war` stores the sum of the numbers of occurrences of `war` and `water`. So TST estimates the same cardinality for both words.

Case 2: Character Reduction. A character-removing map function shortens most words. But since it removes the most frequent characters/character chains first, it tends to keep characters with high information content. However, with a very high approximation degree, a word may be mapped to the empty string. Our estimation in this case is the count of the root node, i. e., the total number of strings in the tree.

5.2 Error Correction

Since we investigate the causes of estimation errors, we now develop an approach to correct them. Our approach is to count the number of different input strings that conflate to a tree node. Put differently, we count the number of different preimages of a node. To estimate the string cardinality, we use the multiplicative inverse of the number of different input strings as a correction factor. For example, think of a map function that maps two words to one node, i. e., the node counts the cardinality of both strings. TST estimates half of the node's count for both words. Example 5.7 illustrates how to compute a correction factor.

Example 5.7 *Imagine a suffix tree that contains the words `water` and `war`. `water` occurs 4 times and `war` occurs 2 times in the database. Figure 5.2a shows the full suffix tree. We now build a thin suffix tree by removing characters `e` and `t`. Both words map to the string `war`. Its node holds a count of* $4 + 2 = 6$*, and the tree will answer both queries, i. e., for `water` and for `war` with 6. Figure 5.2b shows the corresponding tree. When we query the cardinality of the two words, the relative error is* $\frac{6}{4} = 1.5$ *for `water` and* $\frac{6}{2} = 3$ *for `war`. Since we know that this node is reached by 2 different input words, we can answer a query with a cardinality of*

$6 \cdot \frac{1}{2} = 3$. *Using the correction factor* $\frac{1}{2}$ *reduces the relative error to* $\frac{3}{4} = 0.75$ *for* `water` *and* $\frac{3}{2} = 1.5$ *for* `war`.

5.2.1 Counting the Branch Conflations

To compute the correction factor, we need the number of different input words that map to a node. To prevent the tree from double counting input words, each node has to record words already counted. A first idea to do this may be to use a hash table. However, this would store the full strings in the tree nodes and increase the memory usage by much. The directed acyclic word graph (DAWG) [Blu+85] seems to be an alternative. It is a deterministic acyclic finite state automaton that represents a set of strings. However, even a DAWG becomes unreasonably large, to be stored in every node [BEH89]. There also are approximate methods to store a set of strings. In the end, we choose the Bloom filter [Blo70] for the following reasons: Firstly, it needs significantly less memory than exact alternatives. Its memory usage is independent of the number as well as of the length of the strings. Secondly, the only errors are false positives. In our case, this means that we may miss to count a preimage. This can lead to a slightly higher correcting factor, e. g., $\frac{1}{3}$ instead of $\frac{1}{4}$ or $\frac{1}{38}$ instead of $\frac{1}{40}$. As a result, the approximate correction factor is always larger than or equal to the true factor. This means that our correction factor only affects the estimation in one direction, i. e., it only corrects an overestimated count.

To sum up, our approach to bring down estimation errors has the following features: For ambiguous suffixes, i. e., ones that collide, it yields a correction factor in the range $(0, 1)$. This improves the estimation compared to no correction. For unambiguous suffixes, i. e., no collision, the correction factor is exactly 1. This means that error correction does not falsify the estimate.

5.2.2 Counting Fewer Input Strings

TST stores image strings, while our error correction relies on preimage strings. Since the preimage and the image often have different numbers of suffixes (they differ by the number of removed characters), our error correction may count too many different suffixes mapped to a node. For example, take the preimage `water`. A map function that removes characters `a` and `t` returns the image `war`. The preimage `water` consists of 5 suffixes, while the image `war` consists of 3. This renders the correction factor too small and may result in an underestimation. Example 5.8 illustrates how to avoid counting too many preimage strings.

Example 5.8 *Think of the input word* `water` *and its suffixes* `ater`, `ter`, `er`, *and* `r`. *We use a map function that removes the characters* `e` *and* `t`. *This will create a thin suffix tree with nodes that represent* `war`, `ar`, *and* `r`. *See Figure 5.2b. The preimage suffixes* `ter`, `er`, *and* `r` *are mapped to the same node. Since* `ter` *and* `er` *are mapped to the same string as their next shorter suffix, i. e.,* `er` *and* `r` *respectively, we do not count these suffixes for the correction factor. The only suffix we count in node* `r` *is suffix* `r`. *All this yields a thin tree with three nodes and a correction factor of* $\frac{1}{1}$ *in every node.*

We count too many preimages iff a preimage suffix maps to the same image as its next shorter suffix. This is the case for every preimage suffix that starts with a character that is removed by the map function. We call the set of characters a map function removes *trim characters*. This lets us discern between two cases: First, the map function removes the first character of the suffix (and maybe others). Second, the map function keeps the first character of the suffix and removes none, exactly one or several characters within it. To distinguish between the two cases, we check whether the first character of the preimage suffix is a trim character. This differentiation also applies to complex map functions that reduce multiple characters from the beginning of the suffix. To solve the issue of counting too many different preimages, our error correction only counts preimage strings which do not start with a trim character.

5.3 Functions Insert and Query

After describing the details of TST, we now turn to the implementation. We first cover the *map function* and then describe our realization of the functions *insert* and *query*.

5.3.1 Map Function

Algorithm 1 shows an implementation of our general character-removing map function (cf. Section 5.1.4). The function is parametrized by a dictionary that defines which characters to observe and which ones to remove. For example, it contains the three most frequent chains of a length of two characters. For English words, these are `th`, `he`, and `in`. To this end, it removes character `h` if it occurs after character `t`, `e` if it occurs after `h`, and `n` if it occurs after `i`. The map function manipulates only strings that include one of these chains.

Algorithm 1: Function to map words from an input alphabet to a string in tree alphabet.

Input: String in input alphabet

Result: String in tree alphabet

```
1 Function map(inputstring):
2     dict ⟵ getGlobalDictionary()
      /* E.g., {'th' : 't', 'he' : 'h', 'in' : 'i'}                    */
3     imagestring ⟵ inputstring
4     foreach (k, v) ∈ dict do
5         imagestring ⟵ imagestring.replace (k, v)
6     end
7     return imagestring
8 end
```

5.3.2 Insert Function

Algorithm 2 inserts a new word into the TST. It consists of the following steps: (1) map the string from the input alphabet to the tree alphabet (Line 3), (2) insert all its suffixes into the tree (Line 4), and (3) add the associated suffix in input alphabet to the Bloom filter of the respective end node of the tree (Line 9).

Algorithm 2: Function to insert a string.

```
   Input: String to add to the tree
1  Function insert(inputstring):
2      foreach suffix of inputstring do
3          imagestring ⟵ map(suffix)      /* Map suffix to the tree alphabet */
4          insertionpath ⟵ Go down the tree path according to imagestring
5          foreach node n on insertionpath do
               /* Increment the cardinality counter                          */
6              n.count ⟵ n.count + 1
7              if suffix[0] equals imagestring[0] then
8                  if not suffix in n.bloomFilter then
9                      n.bloomFilter.add(suffix)
                       /* Increment the number of seen suffixes              */
10                     n.nbDiffSuffixes ⟵ n.nbDiffSuffixes + 1
11                 end
12             end
13         end
14     end
15 end
```

5.3.3 Query Function

Algorithm 3 is the implementation of how to query the TST. It contains the following three steps: (1) map the given string from input alphabet to tree alphabet (Line 2), (2) search the tree node for this string using the same map function as for function *insert* (Line 3), and (3) estimate the frequency of the string by taking the number of node visits during insertion divided by the number of different suffixes corresponding this node (Line 7).

5.4 Experimental Evaluation

In this section, we evaluate our thin suffix tree approach. We (1) define the objectives of our evaluation, (2) describe the experimental setup, and (3) present and discuss the results.

Algorithm 3: Function to query the cardinality of a string pattern.

Input: String or regular expression

Result: Cardinality estimation for *inputstring*

```
1 Function query(inputstring):
2     imagestring ⟵ map(inputstring)    /* Map input to the tree alphabet */
3     node n ⟵ Go down the tree path according to imagestring
4     if n not exists then
5         return 0
6     end
      /* Correct the estimation (n.count) with the number of different
         input strings that map to node n (n.nbDiffSuffixes).          */
7     return n.count / n.nbDiffSuffixes
8 end
```

5.4.1 Objectives

The important points of our experiments are as follows.

Memory Usage We examine the impact of our map functions on the size of the suffix tree.

Map Function We study the effects of our map functions on the estimation accuracy and analyze the source of estimation errors.

Accuracy We investigate the estimation accuracy as function of the tree size.

Query Run Time We evaluate the query run times, i. e., the average and the distribution.

We rely on two performance indicators: *memory usage*, i. e., the total tree size, and the *accuracy* of the estimations. To quantify the accuracy, we use the q-error, a maximum multiplicative error commonly used to properly measure the cardinality estimation accuracy [KM10; Cor+11; Moe+14]. The q-error is the preferred error metric for cardinality estimation, since it is directly connected to costs and optimality of query plans [MNS09]. Given the true cardinality $\hat{f}$ and the estimated cardinality f, the q-error is defined as follows.

$$\text{q-error} := \max\left(\frac{f}{\hat{f}}, \frac{\hat{f}}{f}\right) \tag{5.7}$$

5.4.2 Setup

Our intent is to benchmark the approaches in a real-world scenario. In addition, we inspect and evaluate the impact of different character-removing map functions on the tree. We now describe the database, the queries used in the experiments, the parametrization, and technical details on the implementation and the hardware.

Database. For pattern search on word chains, we use the 5-grams from the English part of the Google Books Ngram Corpus [Lin+12] (cf. Section 2.2.4). We filter all words that contain a special character, like a digit.[2] At the end, we randomly sample the data set to contain 1 million 5-grams.

Queries. Users may be interested in querying the number of 5-grams that start with a specific word or a specific word chain. Others may want to query for the number of different words that are used together with a specific word or word chain. The answer to both types of question is the cardinality of a search pattern. For our evaluation, we create 1000 queries requesting the cardinality of the 1000 most common nouns in the English language. For example, we query the number of 5-grams containing words like `way`, `people`, or `information`.

Parametrization. Each TST uses a single character-removing map function. The indicators o and r specify the character-removing map function, i. e., how a map function works (cf. Section 5.1.4). Each function has a character frequency list that stores all characters in descending order of their frequency. Hence, we use the list provided by Peter Norvig in *English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU*[3]. To specify the approximation level of the suffix tree, we parametrize the chosen character-removing map function with the number of characters z that are removed by it. This means that the function takes the first z characters from the character frequency list to specify the set of characters removed by the map function. The competitor has the level of approximation as parameter. The parameter specifies the maximal length of the suffixes to store in the tree. Depending on the string lengths of the database, reasonable parameter values lie between 50 and 1 [SEW04].

Technical Details. Our implementation makes use of SeqAn[4], a fast and robust C++ library. It includes an efficient implementation of the suffix tree and of the suffix array together with state-of-the-art optimizations. We run our experiments on an AMD EPYC 7551 32-Core Processor with 125 GB of RAM. The machine's operating system is an Ubuntu 18.04.4 LTS on a Linux 4.15.0-99-generic kernel. We use the C++ compiler and linker from the GNU Compiler Collection in version 5.4.0.

5.4.3 Experiments

We now present and discuss the results of our experiments.

5.4.3.1 Experiment 1: Memory Usage

In Experiment 1, we investigate how a character-removing map functions affects the memory consumption of a TST. We also look at the memory needs of a pruned suffix

[2] We use Java's definition of special characters. See function `isLetter()` of class `Java.lang.Character` for the full definition.

[3] The article and the list are available at `https://norvig.com/mayzner.html`

[4] The SeqAn library is available at `https://www.seqan.de`.

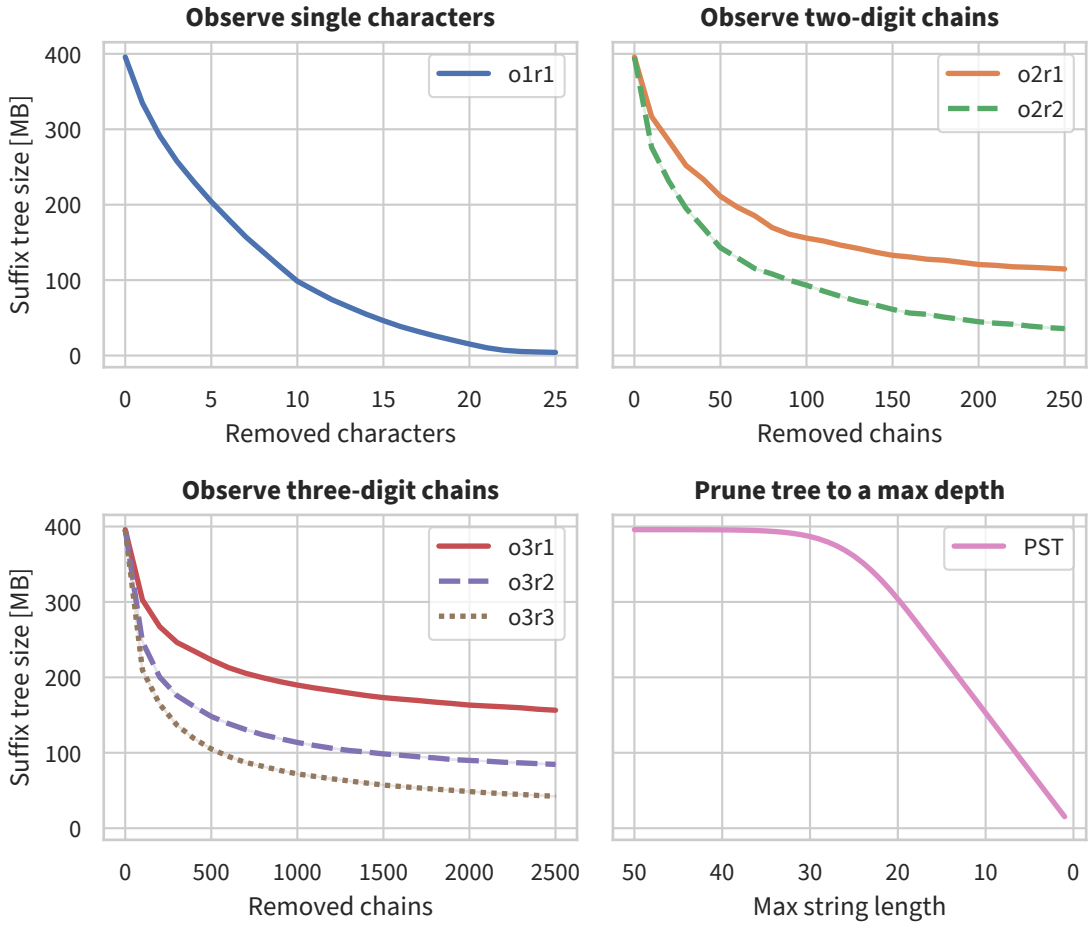


Figure 5.3: TST's memory usage for various map functions and approximation levels.

tree (PST) and compare the two. In our evaluation, we inspect chains of lengths up to 3 ($o = \{1, 2, 3\}$) and in each case remove 1 to o characters.

Figure 5.3 shows the memory usage of the TST for various map functions and approximation levels. The figure contains four plots. The two upper plots and the lower left plot show the memory usage for map functions that observe character chains of length 1, 2, or 3. The lower right plot shows the memory usage of the pruned suffix tree contingent on the maximal length of the suffixes. Note that there are different scales on the x-axis. The database we use in this evaluation includes 179 different characters, 3,889 different character chains of length two, and 44,198 different chains of length three.

The Effect of Character-Removing Map Functions. For a deeper insight into our pruning approach, we now study the effect of character-removing map functions on the memory consumption of a suffix tree in more detail. As discussed in Section 5.1.5, there are two effects that reduce memory consumption: *branch conflation* and *character reduction*. See Figure 5.3. All map functions yield a similar curve: They decrease exponentially. Hence, removing the five most frequent characters halves the memory usage of a TST with map function `o1r1`.

The frequency of characters and character chains in natural language follows Zipf's law, i. e., the Zeta distribution [SEW04]. Zipf's law describes a relation between the frequency of a character and its rank. According to the distribution, the first most frequent character nearly occurs twice as often as the second one and so on. All character-removing map functions in Figure 5.3 show a similar behavior: Each approximation level saves nearly half of the memory as the approximation level before. For example, map function `o1r1` saves nearly 65 MB from approximation level 0 to 1 and nearly 40 MB from approximation level 1 to 2. This shows the expected behavior, i. e., character reduction has more impact on the memory usage of a TST than branch conflation.

The Effect of Horizontal Pruning. The lower right plot of Figure 5.3 shows the memory usage of a pruned suffix tree. The lengths of words in natural language are Poisson distributed [Rot86; SEW04]. In our database, the strings have an average length of 25.9 characters with a standard deviation of 4.7 characters. Since the pruning only affects strings longer than the maximal length, the memory usage of the pruned suffix tree follows the cumulative distribution function of a Poisson distribution.

Summary. Experiment 1 reveals three points. First, the tree size of TST and of the pruned suffix are markedly different for the various approximation levels. Second, the memory reduction of a TST is independent of the length of the strings in the database. Its memory reduction depends on the usage frequency of characters in natural language. Third, the tree sizes for the different character-removing map functions tend to be similar, except for one detail: The shorter the chain of observed characters, i. e., the smaller o, the more linear the reduction of the tree size over the approximation levels.

5.4.3.2 Experiment 2: Map Functions

In Experiment 2, we investigate the impact of map functions on estimation accuracy. We take the map functions from Experiment 1 and measure the q-error at several approximation levels. See Figure 5.4. The right plot is the q-error with the pruned suffix tree. The points are the median q-error of 1000 queries. The error bands show the 95 % confidence interval of the estimation. Note that Figure 5.4 shows the estimation performance as a function of the approximation level of the respective map function. So one cannot compare the absolute performance of different map functions, but study their behavior. The plots show the following. First, the map functions behave differently. At low approximation levels, the map function `o1r1` has a very low q-error. The q-error for this function begins to increase slower than for map functions considering three-digit character chains. At high approximation levels, the q-error of map function `o1r1` is significantly higher than for all the other map functions. The other map functions, the ones that consider three-digit character chains in particular, only show a small increase of the q-error for higher approximation levels. Second, the sizes of the confidence interval differ. With map functions that remove characters independently from previous characters, i. e., `o1r1`, `o2r2`, and `o3r3`, the confidence interval becomes larger with a larger approximation level. For map functions that remove characters depending of previous characters, i. e., `o2r1`, `o3r1`, or `o3r2`, the confidence interval is smaller. This means that, for lower approximation

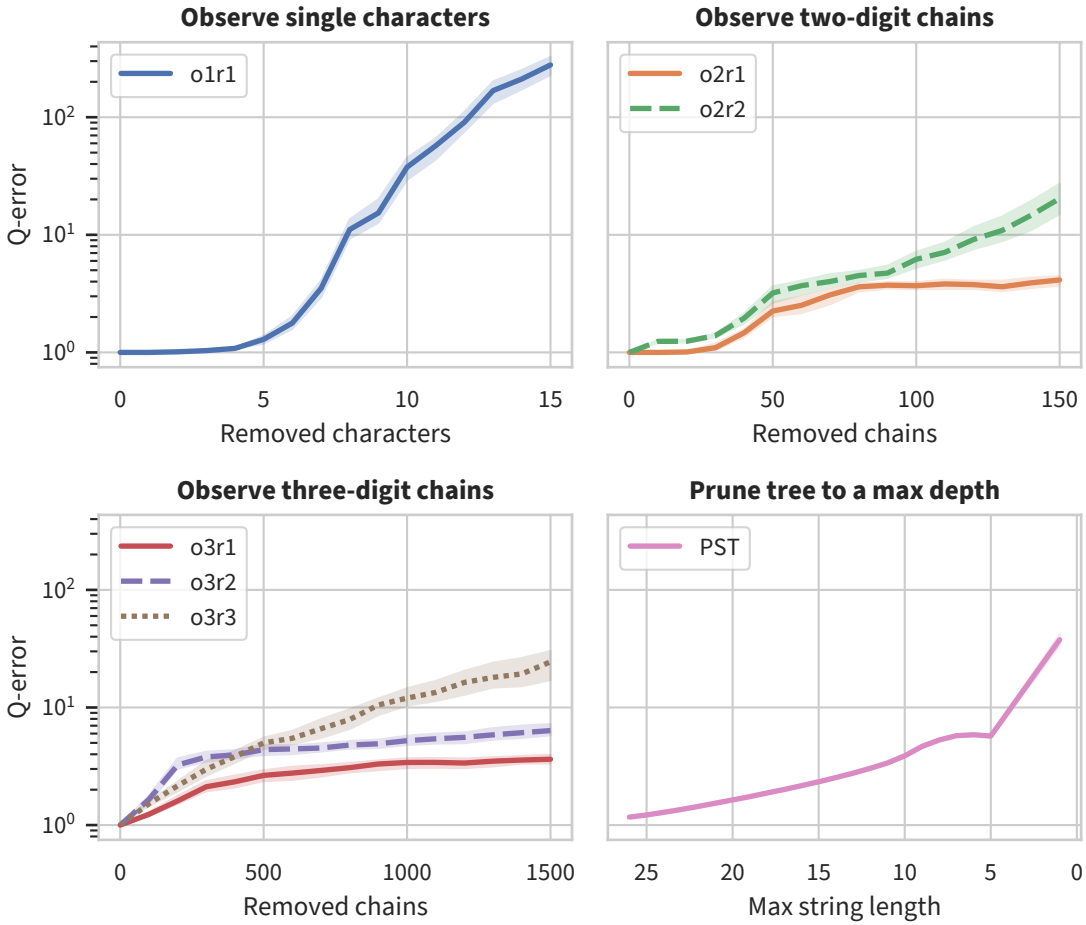


Figure 5.4: TST's q-error for various map functions and approximation levels.

levels and map functions that remove characters depending on previous characters in particular, our experiments yield reliable results. We expect results to be the same on other data. Third, the accuracy of the pruned suffix tree increases nearly linear with increasing approximation levels until it sharply increases for very short maximal strings lengths. This means that every character that is removed from the back of the suffix contributes a similar extent of error to an estimation.

Summary. None of our map functions dominates all the other ones. It seems to work best to remove single characters dependent on either 0, 1, or 2 previous characters, i. e., map functions `o1r1`, `o2r1`, or `o3r1`. For low approximation levels, say up to a reduction of 50 % of the tree size, map function `o1r1` performs well. For high approximation levels, say starting from a reduction of 50 % of the tree size, one should use a map function that removes characters dependent on previous ones, i. e., map function `o2r1` or `o3r1`.

The first five approximation levels of map function `o1r1` are of particular interest, as they show good performance in Experiments 1 and 2. In the first approximation levels, this map function yields a very low q-error, see Figure 5.4, while the tree size goes down very rapidly, see Figure 5.3. In Experiments 1 and 2, we inspect (1) the tree size depending on the approximation and (2) the q-error depending on the approximation level. In many

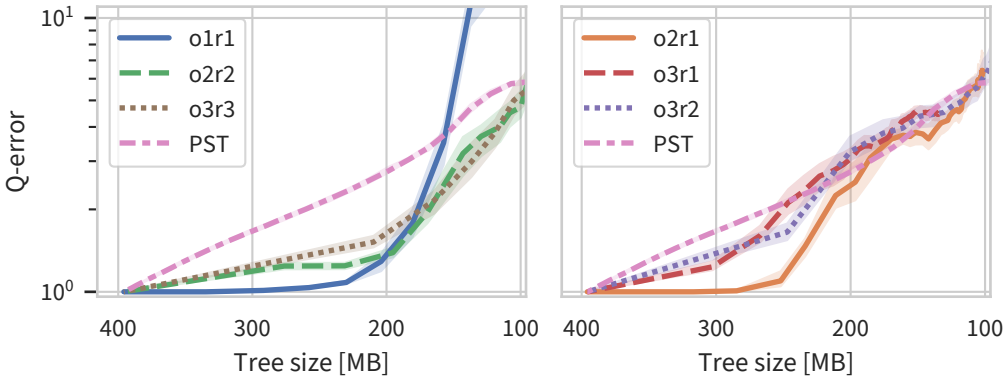


Figure 5.5: The estimation accuracy as function of the tree size.

applications, one is interested in the q-error depending on the tree size rather than on the approximation level. In our next experiment, we compare the q-error of our map functions for the same tree sizes.

5.4.3.3 Experiment 3: Accuracy

In Experiment 3, we compare the map functions from Experiment 1 against each other and against existing horizontal pruning. Figure 5.5 shows the q-error as function of the tree size. For the sake of clarity, there are two plots for this experiment: The plot on the left side shows the map functions that remove characters independently from previous characters. The plot on the right side is for the remaining map functions.

Vertical Pruning vs. Horizontal Pruning. Figure 5.5 shows the following: TST produces a significantly lower q-error than the pruned suffix tree for all map functions used and for tree sizes larger than 60 % of the one of the full tree. For smaller sizes, the accuracy of most map functions does not become much worse than the one of the pruned suffix tree. The only exception is `o1r1`. At a tree size of 50 %, the q-error of `o1r1` starts to increase exponentially with decreasing tree size.

Summary. As Experiments 1 and 2 already indicate, map function `o1r1` achieves a very low q-error to tree size ratio, for tree size reductions of up to 60 %. For this map function, TST yields a significantly lower q-error than the pruned suffix tree for comparable tree sizes. The intuition behind this result is that our vertical pruning respects the redundancy of natural language. Due to this redundancy, map function `o1r1` keeps most input words unique. This results in almost no errors for reductions of the tree size that are less than 50 %. TST also shows a higher degree of confidence, i. e., a smaller 95 % confidence interval, than the pruned suffix tree for reductions that are less than 40 %.

5.4.3.4 Experiment 4: Query Run Time

In Experiment 4, we compare the query run time of the TST using different map functions with the one of a pruned suffix tree. We consider sample tree sizes of 300 and 200 MB.

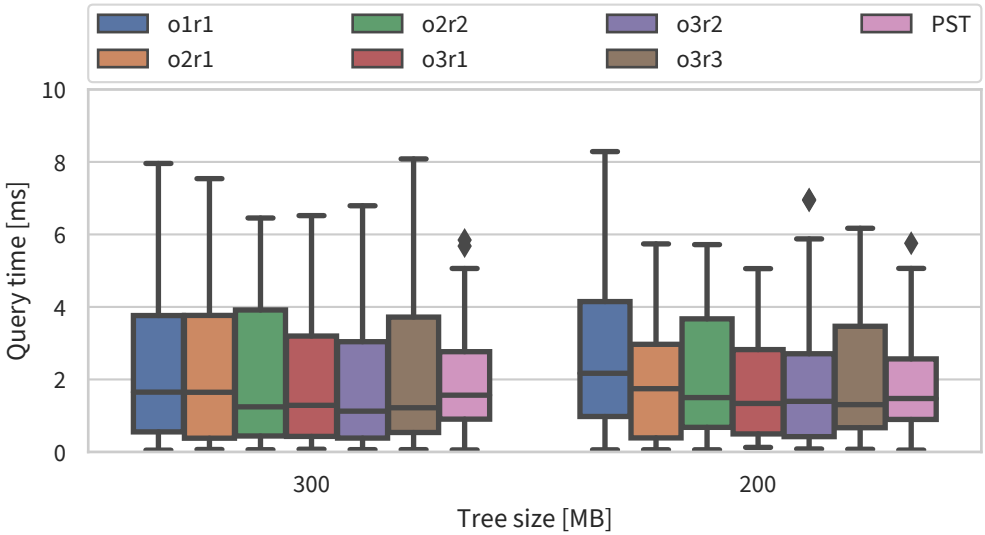


Figure 5.6: The query run time of the TST.

Figure 5.6 shows the average and distribution of the run time for all queries. There is no significant difference in the run time. TST potentially needs a slightly higher run time than a pruned suffix tree. This is because TST is potentially deeper than a pruned suffix tree and additionally executes a map function. To conclude, the additional work of TST is of little importance for the query run time compared to a pruned suffix tree.

5.5 Summary

In this chapter, we study cardinality estimation for co-occurrence queries, i. e., the co-occurrence of two words within an ngram. In contrast to existing work, our approach aims at databases with long strings, especially ngrams. One way to estimate cardinality on string databases is the suffix tree. But since they tend to use much memory, they usually are pruned down to a certain size. We propose a novel pruning technique for suffix trees for long strings. Existing pruning methods mostly are horizontal, i. e., prune the tree to a maximum depth. Here we propose what we call *vertical pruning*. It reduces the number of branches by merging them. We define map functions that remove characters from the strings based on the entropy or conditional entropy of characters in natural language. Our experiments show that our thin suffix tree approach does result in almost no error for tree size reductions of up to 50 % and a lower error than horizontal pruning for reductions of up to 60 %. TST is a crucial part for query optimization on large ngram databases and temporal text corpora.

6 Efficient Interval-focused Time Series Similarity Search

In this chapter, we look at an efficient way to access ngrams based on their usage frequency time series. A typical way to provide efficient data access is to index the database. The resulting index data structure facilitates to quickly locate specific data. Index structures differ depending on the particularities of both the data and the query to accelerate. When using temporal text corpora, the data consist of ngrams and frequency time series. We already presented our data model in Section 4.3. We now look at a typical query that requires accessing ngrams based on their time series. Example 6.1 depicts a typical examination of conceptual historians.

Example 6.1 *A conceptual historian studies the impact of a world war on language and society. This means, her interest is confined to the time interval, say, between the two world wars or the time interval from the end of World War I to the end of World War II. To examine the social changes coming and going with a war, a conceptual historian might search for words that have a similar usage frequency to the word "war". Since a social system needs some time to adjust to a change, the usage frequency of interesting words might also rise and fall somewhat delayed to "war".*

We further specify this example in the following. A conceptual historian examines the economic effects of the two world wars. For this purpose, she searches for words similar to the word "battleship" in the time interval from 1910 to 1960. When searching the Google Books Ngram Corpus, the result set contains the word "reparations" whose shape is similar to the one of "battleship" with an offset of approximately 5 years. Figure 6.1 shows both time series.

Example 6.1 illustrates the following two requirements to search for words with a similar usage frequency. First, one needs a warping similarity measure to determine the similarity between two time series. Warping describes a non-linear temporal alignment, i. e., it allows a temporal delay between the two time series. A popular warping similarity measure is DTW (cf. Section 3.3.4) [SC78; KR05; SYF05].[1] Figure 6.1 illustrates DTW's non-linear alignments for the usage frequencies of the words "reparations" and "battleship" in the Google Books Ngram Corpus. Second, the information need in the above example is the nearest neighbor search on time series confined to arbitrary intervals. This kind of information need is called *interval-focused similarity search* [Aßf+07].

[1] The remainder of this chapter bases on our article [Wil+19a]: Jens Willkomm, Janek Bettinger, Martin Schäler, and Klemens Böhm. 2019. "Efficient Interval-focused Similarity Search under Dynamic Time Warping." In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases* (*SSTD '19*). DOI: 10.1145/3340964.3340969.

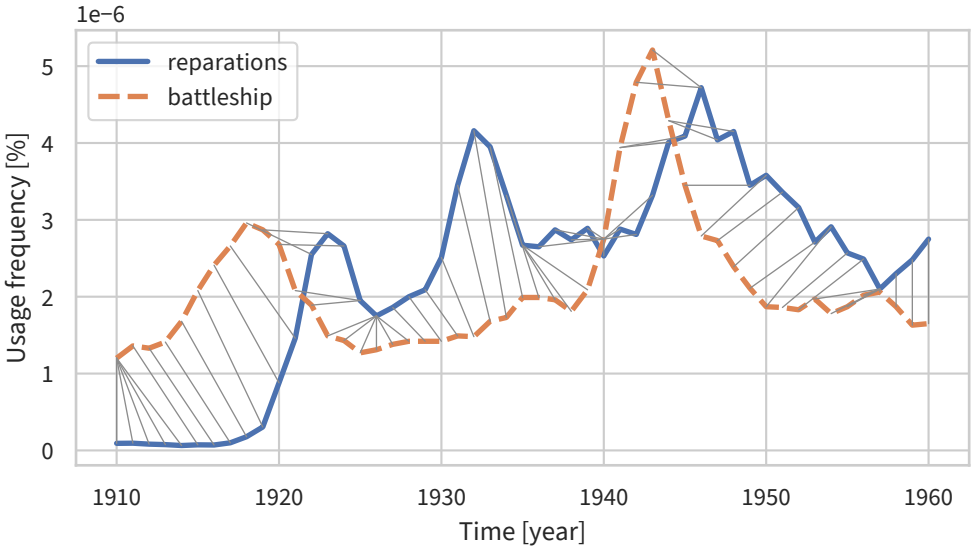


Figure 6.1: DTW's non-linear alignments between the usage frequency of the words "reparations" and "battleship" in the time interval $[1910, 1960]$.

In this chapter, we address the problem of efficiently answering interval-focused kNN queries on many time series. A kNN search returns k time series with the smallest distance to the query time series regarding a time series distance measure. Its generalized version, the interval-focused kNN search $\text{kNN}(k, Q, [a, b])$, restricts the distance measure to a time range $[a, b]$ that the user specifies as part of the query. As described in Example 6.1, we focus on DTW as distance measure. However, this does not impose any limitation, as one can easily replace the DTW measure with the Euclidean distance or any other L^p norm without violating any lower bound used subsequently. This is because these distances are special cases of DTW, i. e., the DTW distance without any alignment.

The literature gives only little attention to interval-focused similarity search. However, an extensive study—of whole matching similarity search—has resulted in two fundamental techniques. One common technique uses a spatial data structure, like an R- or R*-tree, to index time series using minimal bounding boxes [Bec+90; SK91; FRM94; Agr+95; Fu+07; Ass+08]. This only works for metric distances, like the Euclidean distance, but does not work for semimetric distances that miss the triangle inequality, like the DTW distance [Vid+88]. In contrast, there are other techniques that use lower bounds for the DTW distance to prune either single time series [KPC01; KR05] or groups of time series [KS10; NRR10]. These techniques sequentially scan the data element-wise or group-wise. We refer to the first one as *lower bound* and the second one as *partitioning*. To sum up, most related works either lack supporting DTW as a distance measure, require considering every element, or do not work for interval-focused queries. This indicates that efficiently answering interval-focused kNN queries is difficult.

Challenges. To efficiently answer kNN queries on time series, like the one shown in Example 6.1, one must cope with the following challenges.

Interval-focused Queries Since a query can refer to an arbitrary time interval, i. e., having an arbitrary start and endpoint, an index has to hold relevant information, like a lower bound estimation, for every possible time interval.

Efficient Pruning A data structure needs to organize the data in a way so that the search needs to only deal with a share of the elements. This reduces the number of distance computations.

High Dimensionality Time series are high-dimensional objects. This typically leads to high tree-traversal costs.

Contributions. In this chapter, we present the *Time Series Envelopes Index Tree* (TSEIT)[2], a novel tree-based index structure to efficiently evaluate interval-focused kNN queries. TSEIT is a dynamic data structure that supports operations to *insert*, *query* and *delete* time series. Our first contribution is the notion of hierarchically organized envelopes that form the basis of our tree structure TSEIT. The leaf nodes store the time series, and the inner nodes store envelopes that allow TSEIT to prune subtrees during query evaluation. Our second contribution is a tree-height-based envelope approximation. It sets the envelope approximation degree based on the node's height. This speeds up the tree traversal near the root node and at the same time yields strong lower bounds near the leaf nodes. Our third contribution is a lower bound for arbitrary time intervals between an envelope and a time series. The lower bound holds for envelopes and time series in full dimensionality as well as for their piecewise aggregate approximation representation. We compare our approach against state-of-the-art methods for whole matching similarity search that we have modified to replace the whole time series with the queried interval during query time. TSEIT reduces the query execution time for a data set of 5 million time series by up to 50 %.

Outline. We structure this chapter as follows. In Section 6.1, we describe the details of our data structure. Section 6.2 shows the superiority of our approach.

6.1 The Time Series Envelopes Index Tree

In this section, we introduce TSEIT, our novel time series index. First, we focus on the tree structure and, second, describe the operations.

6.1.1 Tree Structure

TSEIT is a search tree that stores time series envelopes in its nodes. Envelopes seem to be a more suitable representation of time series groups than rectangles. Figure 6.2 illustrates a Time Series Envelopes Index Tree with two leaf nodes and one inner node. TSEIT is a balanced tree that grows upwards, i. e., towards the root. Every node stores an envelope that wraps all time series reachable by this node. TSEIT distinguishes between two types

[2] TSEIT is pronounced [tsaɪ̯t], like the German word *Zeit* (English: *time*).

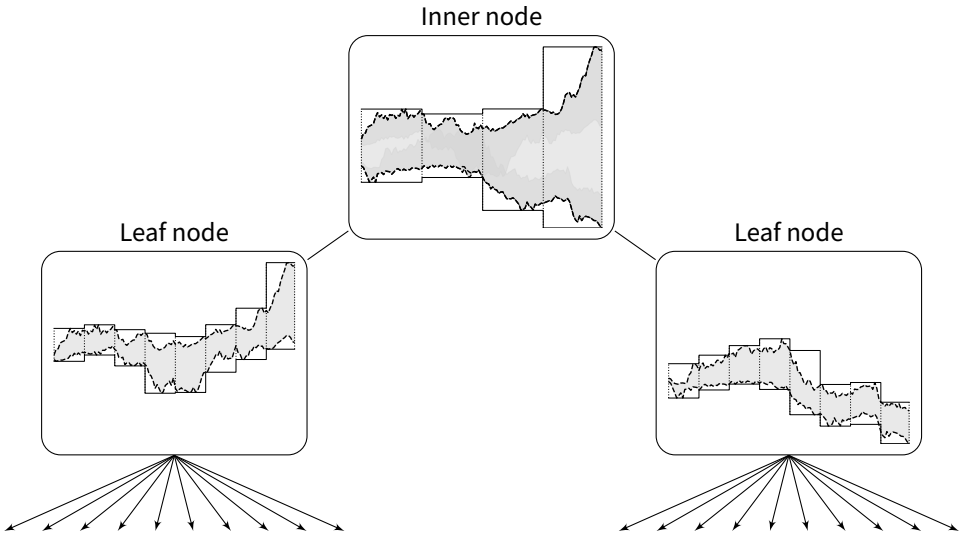


Figure 6.2: An example of a Time Series Envelopes Index Tree with two leaf nodes and a single inner node.

of nodes: leaf nodes and inner nodes including the root node. Leaf nodes hold envelopes in full dimensionality, whereas inner nodes store PAA envelopes. The envelopes of the inner nodes towards the root grow into *two* directions: (1) They grow on the *domain axis*: The envelope encloses the minimum and maximum of its child nodes. (2) They grow on the *time axis*, i. e., the PAA segment size increases with the tree height and thus the envelope dimensionality of higher nodes decreases. In other words, we start from the root node with a loose envelope that gets tighter every step downwards the tree on the domain axis as well as on the time axis. Using envelopes instead of bounding boxes is our solution to address the problem of large-volume rectangles discussed in Section 3.3.2. Figure 6.2 illustrates TSEIT's envelope-growing behavior. We describe the PAA envelope of inner nodes in the following.

6.1.1.1 PAA Envelope of Inner Node

Recall that leaf nodes store envelopes in full dimensionality. This is equal to a PAA representation with a segment size of 1. In contrast, inner nodes at the same height have the same dimensionality. Parent nodes double the PAA segment size of their children. This is, inner nodes at height 1 have envelopes with segment size 2, while the nodes one level up have envelopes with segment size 4 and so on. Since TSEIT grows upwards and thus all its child nodes have the same depth, the PAA segment size T of a node is a function of the node's height h:

$$T(h) = 2^h \tag{6.1}$$

6.1.2 Interval-focused Query

To answer query($Q, k, [a, b]$), TSEIT searches its leaf nodes in ascending order of the node's lower bound to Q until the lower bound distance is larger than the best-so-far distance. See Algorithm 4.

Algorithm 4: query(Q, k, [a, b])

Input: Q ← query time series
k ← desired number of results
[a, b] ← query interval

Data: root ← root node of TSEIT
L ← a priority queue
R ← a sorted list of tuples of $\langle \text{DTW}(Q, C), C \rangle$
bsf ← the best-so-far distance

Result: A list with k elements having the lowest DTW distance to time series Q

```
1  begin
2      L.enqueue(root)
3      while L.has_next() do
4          node ← L.next()
5          T ← 2^node.height                          /* node's segment size */
6          if bsf ≤ LBG(Q^T[a, b], node.E^T[a, b]) then
7              break
8          end
9          if node.is_leaf_node() then
10             cands ← seq_scan(node, Q[a, b], k, bsf)
11             R.insert(cands); R.set_size(k)
12             bsf ← max(R.get_distances())
13         else
14             L.enqueue(node.get_child_nodes())
15         end
16     end
17     return R
18 end
```

The following describes specifics of TSEIT's query algorithm: (1) tree traversal with PAA envelopes, (2) query an interval that differs from PAA segment borders, and (3) scanning a leaf node.

6.1.2.1 Tree Traversal Based on PAA Envelopes

The tree traversal is based on the LBG lower bound between query time series Q^T and the node's envelope E^T (Line 6). Q^T and E^T are PAA versions of the query time series and the

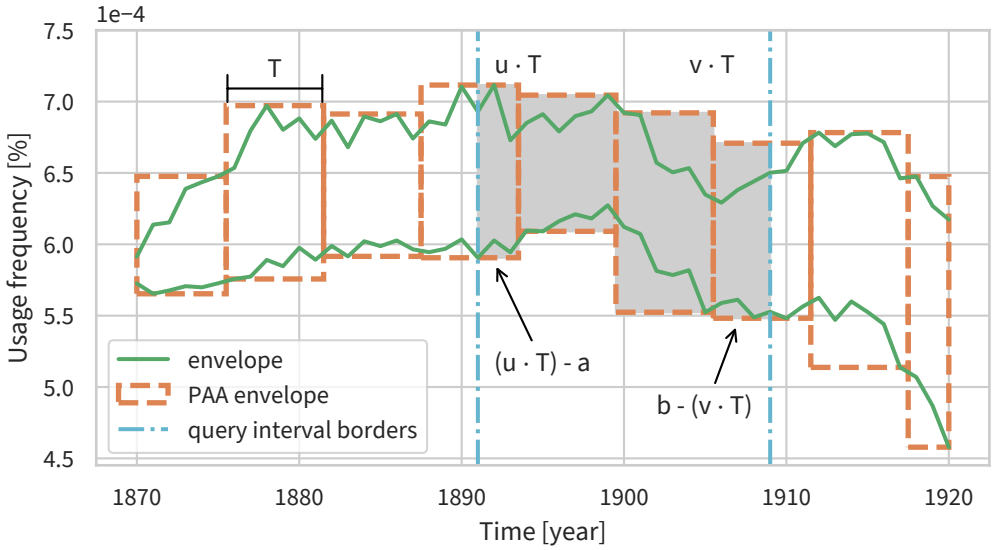


Figure 6.3: An LBG computation that partially includes the PAA segments at start and end of interval $[a, b]$.

envelope of the current node. Line 5 specifies the PAA segment length used on the current tree level. Note that the PAA version of the envelope E^T is already available. Note also that we need to compute the PAA representations of the query time series Q^T only once per query. The number of necessary PAA representations depends on the height of the tree. Using height based PAA representations has the following nice properties: It speeds up the tree traversal, and the lower bounds become more accurate towards the leaf nodes.

6.1.2.2 Query Interval Border Inside a PAA Segment

The approximation with PAA replaces all data points inside a segment with a single data point. The query intervals a and b might lie inside a PAA segment. We call these segments *side segments*. Figure 6.3 illustrates a query whose interval borders lie inside a PAA segment. Including the side segments to the lower bound invalidates the bound. Excluding the side segments leads to a less tight lower bound and thus to a lower pruning rate. Therefore, we modify the lower bound to partially include the side segments and thus keep the lower bound's tightness in arbitrary intervals.

The first relevant segment ends at the first PAA border u inside the interval $[a, b]$. See Equation 6.2.

$$u = \arg\min_{1 \le x \le t}(x \cdot T \ge a) \tag{6.2}$$

The last relevant segment starts at the last PAA border v inside the interval $[a, b]$. See Equation 6.3.

$$v = \arg\max_{1 \le y \le t}(y \cdot T \le b) \tag{6.3}$$

Altogether, the first relevant PAA segment u inside the interval ends at position $u \cdot T$ and the last relevant PAA segment v starts at position $v \cdot T$. We have to include the first

segment with length $(u \cdot T) - a$ and the last segment with $b - (v \cdot T)$. Figure 6.3 illustrates the side segments and its length.

Equation 6.4 is a version of LBG lower bound that includes side segments.

$$\text{LBG}(Q^T, E^T, [a, b]) = \sqrt[p]{D(v, v)} \tag{6.4}$$

$$D(i, j) = T_{seg}(i, j) \cdot D_{seg}(q_i^T, e_j^T) + \min \begin{cases} D(i-1, j-1) \\ D(i-1, j) \\ D(i, j-1) \end{cases} \tag{6.5}$$

$$T_{seg}(i, j) = \begin{cases} (u \cdot T) - a & \text{if } i < u \text{ or } j < u \\ b - (v \cdot T) & \text{if } i > v \text{ or } j > v \\ T & \text{otherwise} \end{cases}$$

$$D_{seg}(q_i^T, e_j^T) = \begin{cases} |lq_i^T - ue_j^T|^p & \text{if } lq_i^T > ue_j^T \\ |le_j^T - uq_i^T|^p & \text{if } le_j^T > uq_i^T \\ 0 & \text{otherwise} \end{cases}$$

where $D(u-1, u-1) = 0$ and $D(i, u-1) = D(u-1, j) = \infty$ for $1 \leq i, j \leq t$.

6.1.2.3 Optimized Leaf Node Sequential Scan

If the lower bound between a query time series and a node's envelope is smaller than the best-so-far distance, we need to sequentially scan a leaf node. Since LBG computes a lower bound for all time series of this node, we first compute a lower bound to the individual time series or, more precisely, a cascade of lower bounds. Because of its good performance [Rak+12], we use a cascade of the two lower bounds LB_KimFL and LB_Keogh (cf. Section 3.3.6). This might already prune many time series of the node. For the remaining time series, we need to compute the true DTW distance.

6.1.3 Insert Operation

To insert a new time series, we search the tree for the leaf node with the lowest insertion cost. This section describes: (1) TSEIT's insertion cost functions, (2) the node overflow handling, and (3) TSEIT's parameters.

6.1.3.1 Insertion Cost

Recall that the LBG results in a lower bound of 0 if the time series completely lies between the upper and lower sequence of the envelope (cf. Section 3.3.5). To maximize the lower bound, TSEIT minimizes the size of the envelopes, i. e., the area surrounded by the upper and lower sequence. TSEIT's primary insertion cost function is the enlargement of an envelope, i. e., the area it will grow by after inserting the new time series. Equation 6.6 de-

fines function enlargement(E, C) that returns the size envelope E will grow when inserting time series C.

$$\text{enlargement}^2(E, C) = \sum_{i=1}^{t} \begin{cases} (c_i - le_i)^2 & \text{if } c_i > ue_i \\ (ue_i - c_i)^2 & \text{if } c_i < le_i \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

If there are several envelopes with the same enlargement costs, TSEIT picks the envelope with the least expansion. The expansion of an envelope is the area between its upper and lower sequence. See Equation 6.7.

$$\text{expansion}(E^T) = \sum_{i=1}^{t} T \cdot \left(eu_i^T - el_i^T \right) \tag{6.7}$$

While function enlargement considers the growth of an envelope, function expansion considers its total area. In other words, if two envelopes do not need an enlargement, TSEIT chooses the smaller envelope.

6.1.3.2 Node Overflow Treatment

If an overflow occurs to a node for the first time, TSEIT reinserts time series of this node, to globally minimize envelope expansion. If an overflow occurs the second time, TSEIT splits this node. A node split aims to *locally* minimize envelope expansion. Since its envelope wraps all time series of a node, we initialize the split with the envelope's upper and lower sequence and minimize the sum of the expansion of the resulting envelopes using k-means. We call this heuristic *EnvelopeSplit*.

6.1.3.3 TSEIT Parameters

Like any tree, TSEIT has parameters to control the tree development. The first parameter pair sets the minimal and maximal capacity of the leaf nodes. We call these parameters *minNodeSize* and *maxNodeSize*. The second parameter pair defines the minimal and maximal number of child nodes. We call these parameters *minNodeChildren* and *maxNodeChildren*.

6.1.4 Generalizing Other Approaches

We design TSEIT as a generalization of two state-of-the-art approaches: TWIST [NRR10] and UCR Suite Cascading Lower Bounds (UCR) [Rak+12]. TSEIT can simulate the TWIST approach by setting the maximal number of child nodes to infinity. This forces TSEIT to always append new leaf nodes to the root and keep a tree height of two. The leaf nodes simulate the partitions, and the traversal step of the root node simulates the pruning of single partitions. TSEIT can also simulate the UCR approach by setting TSEIT's node size to infinity. This prevents TSEIT from splitting the root and always keeps a single node.

6.2 Experimental Evaluation

In this section, we conduct three experiments to gain insights into the benefits and drawbacks of TSEIT. We start with a micro benchmark (Experiment 1) systematically evaluating the influence of parameters, such as the maximum node size, on the response time. We are primarily interested in the response-time robustness of different parameters. In addition, we aim at revealing the most significant parameters influencing the response time and finding a good parameter combination for the remaining experiments. In Experiment 2, we compare the query performance of TSEIT the reference points. The primary objective is to investigate whether TSEIT's combined usage of lower bounding and partitioning, i. e., our generalization, results in significant performance improvements. To this end, we examine how LBG and DTW computations are related to performance differences of the approaches. In the final experiment, we evaluate whether the build times of TSEIT are reasonable. Before we start with the experiments, we describe the experimental setup of all subsequent experiments.

6.2.1 Experimental Setup

This section explains (1) the selection of reference points for this evaluation, (2) technical details, (3) used data sets, and (4) data preprocessing.

Reference Points In the experiments, our primary objective is to investigate whether one of the existing techniques (cf. Section 6.1.4) is dominant, i. e., the main reason for performance increases, or whether the combination of techniques is required. To this end, we rely on a sequential scan as baseline. Further, we use two state-of-the-art approaches: one applying only partitioning, named TWIST [NRR10], and one using only lower bounds, named UCR Suite Cascading Lower Bounds [Rak+12]. As we described in Section 6.1.4, our own approach generalizes both approaches, i. e., it can also be configured to mimic them.

Technical Details. We run all our benchmarks on a machine having an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz and 126 GB of RAM. Our machine runs Ubuntu 16.04.4 LTS as operating system. We implement all approaches in Java and execute them on an OpenJDK 64-Bit Server VM in version 1.8.0_181. Our implementation ensures that all approaches, our TSEIT approach as well as the competitors, keep all their data in memory.

Data Sets. For our experiments, we rely on the Google Books Ngram Corpus [Lin+12] (cf. Section 2.2.4). To ensure generality of our findings, we use the 2-gram data of the English, German, and Spanish corpus. For each 2-gram, the corpus contains its usage frequency starting at year 1800 and ending with year 2008.

Data Preprocessing. The raw data of the Ngram corpus has several quality issues, such as OCR errors. To this end, we preprocess the data as follows. We remove words from the data set that contain:

Special characters We filter words that contain special characters, e. g., digits. For the definition of special characters, we use the Java function `java.lang.Character.isLetter()`.

Annotations The corpus offers semantic annotations of the words, e. g., whether the word is used as noun or as verb. We filter the annotated words and keep the ones without annotation.

As final preprocessing step, we normalize the time series, to remove the autocorrelation of the time series. Google provides the corpus with the word's absolute usage frequency. With a rising number of published books, the number of word usages also rises every year, leading to autocorrelation. We remove this trend by calculating the word's relative usage, i. e., the share of the word usage in the total usage of all words in this year.

6.2.2 Experiment 1: Parameter Influence

TSEIT is configurable by four parameters: Two specify the node size and two specify a valid number of child nodes. In this subsection, we have the following two objectives: First, we inspect the robustness of the parameters and analyze their effect on the query run time. Second, we aim to find the best parameter settings to benchmark the query performance in Section 6.2.3.

6.2.2.1 Influence Analysis

We create data sets of different sizes by randomly subsampling the English corpus. Figure 6.4 shows the results of our analysis over data sets of different sizes.

6.2.2.2 Result Interpretation

Recollect that we have two interests: (1) robustness of kNN query performance, and (2) finding the best parameter settings used subsequently. The second item will also result in first insights whether a combination of lower bounding *and* partitioning minimizes the query time. For instance, in case the best performance is achieved if the maximal node size is close to the data set size, i. e., TSEIT behaves like UCR, it is lower bounding that affects the performance. In contrast, in case every leaf node contains a share of the elements, we see this as an indication that lower bounding *and* partitioning are advantageous.

Parameter Robustness. The results clearly indicate that the most relevant parameter for TSEIT's query performance is *maxNodeSize*. Since we find good values for this parameter independent of the size of the data set (even if the optimal values slightly increase with the size of the data set), we conclude that TSEIT's kNN query performance is robust.

Optimized Parameter Setting. Based on the parameter influence analysis, we found the following parameter setting to be most efficient on the considered data sets, and we therefore rely on it in the subsequent experiments: We set the maximum node size to 1,000 time series and the minimum node size to 250 time series. We set the valid number of node children from 1 to 3, i. e., we allow a maximum of three child nodes per node. This

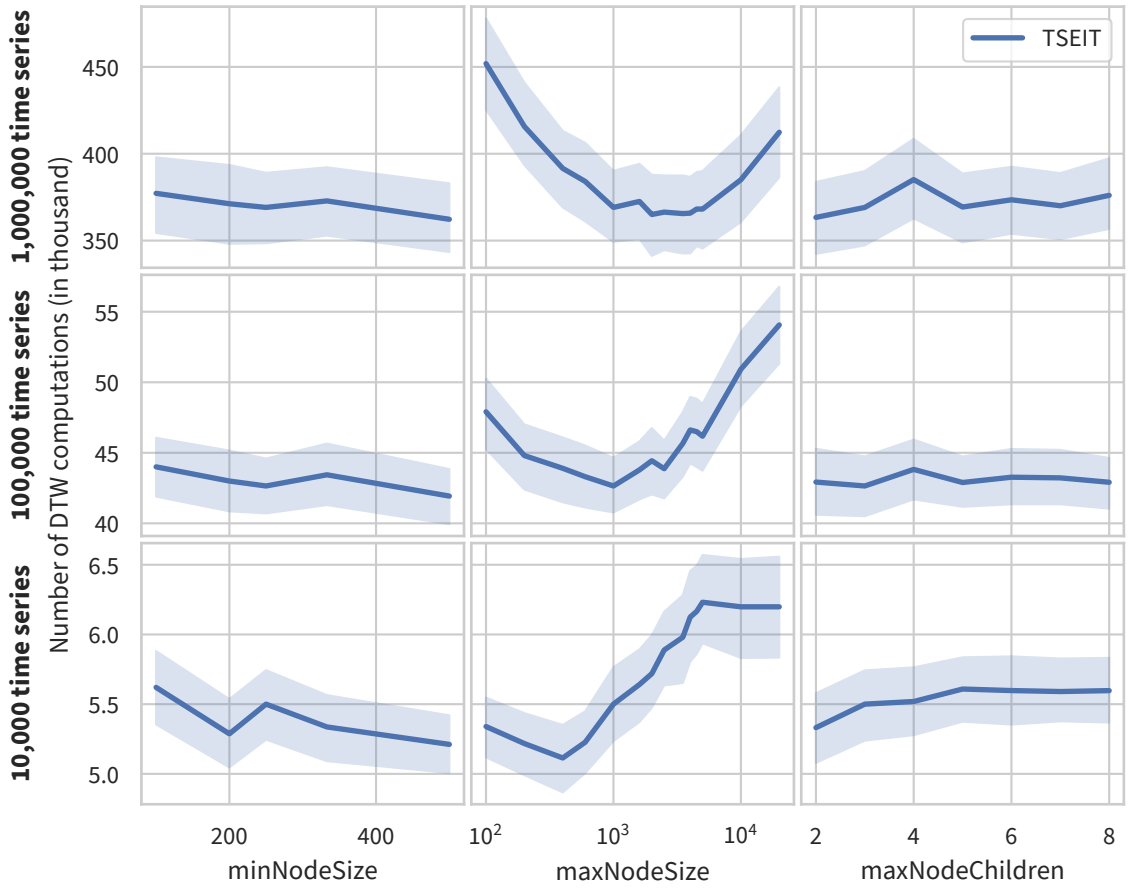


Figure 6.4: The impact of TSEIT's node size on the number of required DTW computations.

optimized configuration reveals that TSEIT organizes its envelopes hierarchically, without any degeneration yielding behavior like the one of TWIST or UCR. This suggests that our generalization is superior to existing approaches. We investigate this in detail as part of Experiment 2.

So we fix those parameter settings for all further experiments. To have a fair comparison, we set TWIST's partition size to TSEIT's node size. UCR is parameter-free.

6.2.3 Experiment 2: kNN Query Performance

TSEIT's main purpose is minimizing the response time of interval-focused kNN queries. The prior experiment in Section 6.2.2 gives first indications that our generalized approach is faster than approaches optimizing a single pruning technique. We now examine the query run times in more detail using the following setup.

6.2.3.1 Benchmark Setup

To provide reproducible, valid results, we provide details of how we have implemented this benchmark. To this end, we first describe the process to select the parameter values of a query. We then specify our performance indicators.

Query Selection. An interval query kNN($k, Q, [a, b]$) depends on the number of neighbors to search k, the query time series Q, and the interval $[a, b]$. This section specifies how we set these parameters.

To have different result sizes, we uniformly vary parameter k from 1 to 10. These sizes yield results interpretable by domain experts, conceptual historians in our case. Parameter Q specifies the query time series. Here, we select a random time series from the full corpus. To simulate a real-world workload, we weight the probability of every time series with its usage frequency, i. e., more frequent words have a proportionally higher probability to be selected. This reflects the fact that a common word as a query is more likely than, say, a word with a typo in reality. We also vary the start and endpoint of the query interval $[a, b]$ uniformly, i. e., even if the same time series is selected by chance, the interval is most likely different. Since the run time of DTW depends on the length of the time series and thus the length of the interval query, we have to keep a fixed interval size for our experiments. Otherwise, the interval length dominates the query run time and thus disturbs the variance of our benchmarks. We choose an interval size of 150 years and set DTW's warping window to the same value.

Performance Indicators. Since our main objective is to minimize the query run time, this is our main performance indicator. To explain run time differences and to investigate whether our generalization is the main reason for the observed differences, we rely on two additional implementation-independent measures. The first one quantifies the overhead of the data structure to search for elements, by counting the LBG computations. The rationale is that this computation is the only expensive operation in the traversal (cf. Algorithm 4). The second one counts the distance computations between the query and candidate time series to quantify the effect of partitioning. For both additional indicators, we use the fraction instead of the absolute count to obtain a number unbiased from the size of the data set size that makes different sizes comparable. To sum up, we use the following performance indicators.

Data Structure Search Costs (LBG) We investigate the average fraction of LBG computations necessary to search the data structure, e. g., to traverse the index. A value of 100 % means that the approach needs as many LBG computations as it contains time series. Fewer LBG computations means lower costs to search the data structure and thus should accelerate query processing. Note that, by definition, this measure is 0 for the sequential scan and UCR. This is because these approaches do not use this technique.

Necessary Distance Computations (DTW) We investigate the average fraction of full DTW computations necessary to find the nearest neighbors. A value of 100 % means that the exact distance to each other time series in the data set is computed, i. e., the approach computes as many distances as the sequential scan.

Query Wall-clock Time We measure the time from calling the query routine until it returns. For statistical soundness, we use the mean and variance of executing 100 randomly selected query time series and intervals.

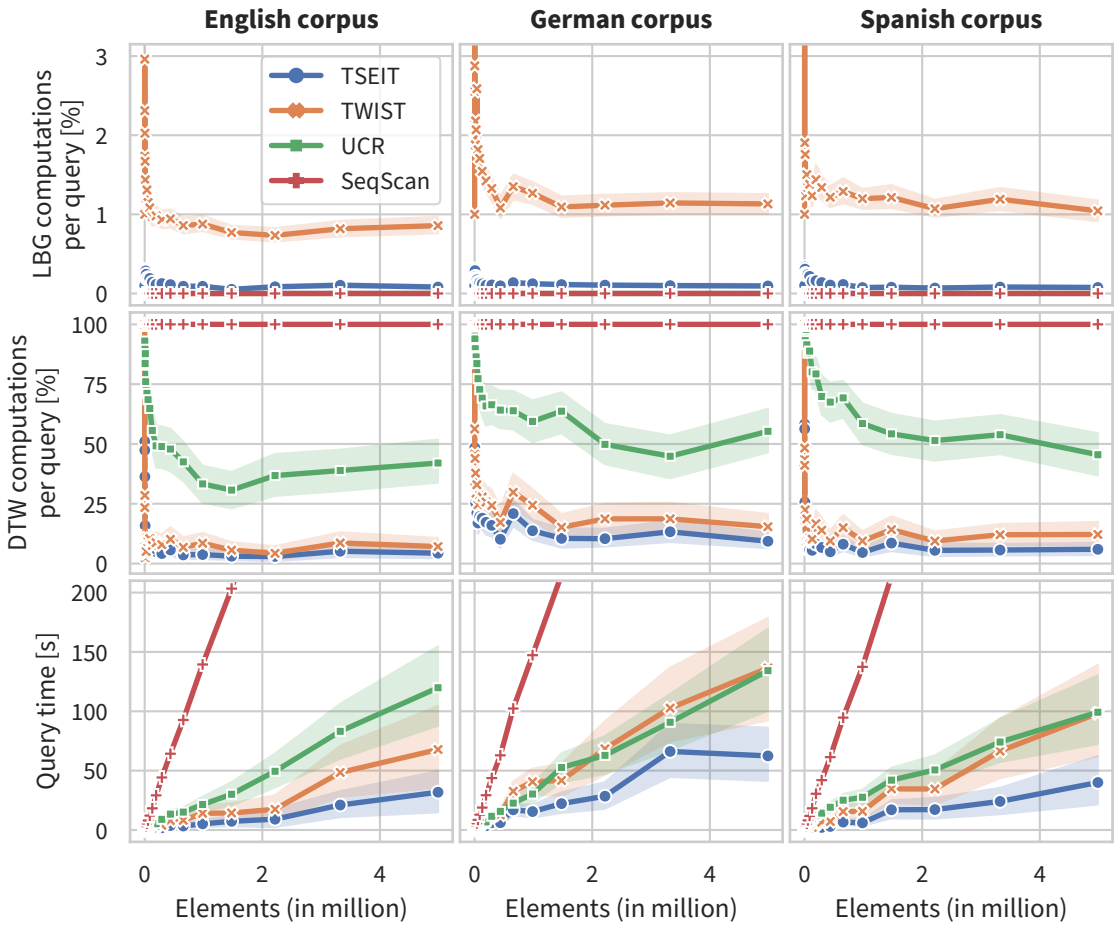


Figure 6.5: The query performance measurements depending on the number of elements and different corpora.

6.2.3.2 Benchmark Results

Figure 6.5 plots all performance indicators depending on the number of time series, on three different language corpora. We now describe these results.

LBG Computations. The first row of Figure 6.5 plots the mean and the confidence band (95 %) of the share of LBG computations necessary to answer a single query. Recall that UCR and the sequential scan do not do any LBG computation. We observe that TSEIT performs very few LBG computations for all sizes of the data set on all three corpora. The value is smaller than 0.4 % in any case. In contrast, TWIST needs 8 to 10 times more LBG computations than TSEIT. TWIST is inefficient for very small data sets, but gets more efficient for larger ones. So we conclude that the working of TSEIT with our optimized parameter settings is significantly different from the one of TWIST. This is another indication that both partitioning and lower bounding are required.

DTW Computations. The second row of Figure 6.5 plots the mean and the confidence band (95 %) of the number of DTW distance computations necessary to answer a query.

Remember that a sequential scan computes for each query point Q the distance to all points in the data set, i. e., has a value of 100 % for this measure. For all other approaches, we observe that they compute only some of the distances. The only exception is very small data sets, where all approaches converge towards a sequential scan. For larger data sets, we clearly see that the most DTW computations are required after applying UCR. Interestingly, the graphs for TSEIT and TWIST have a very similar shape. Observe the small number of distance computations, which usually is between 10 and 20 %. This saves a lot of distance computations since there are up to 5 million time series.

The results regarding this measure allow the following conclusions. First, there is a significant difference between TSEIT and UCR: UCR does not prune groups of irrelevant candidate time series. Second, since TSEIT and TWIST rely on partitioning, both have few DTW distance computations as an effect of LBG pruning. Since the plots of the numbers of DTW computations required for TSEIT and TWIST are very similar, a significant difference in the query run times would mean that indeed lower bounding and partitioning are required.

Query Run Time. The third row of Figure 6.5 plots the mean and the confidence band (95 %) of the wall-clock time to answer a query. As expected, the run time of the sequential scan grows linearly with the size of the data set. We observe that, for any corpus and size of the data set, all approaches outperform this baseline. On the English corpus, TWIST is faster than UCR, while both have a similar run time on the German and Spanish corpus. Our TSEIT approach shows the best run time on all three corpora. On average, it performs a query 50 % faster than TWIST or UCR.

We conclude that indeed the combination of lower bounding and portioning results in the best response times. These times are not observed for any competitor relying only on one of these techniques. Considering the kNN query times, TWIST is second even if TSEIT clearly outperforms it. To get the full picture, we examine the build times of the index in the next experiment.

6.2.4 Experiment 3: Index Build Times

The purpose of studying the build time is to evaluate whether a large build time is a counterargument for TSEIT. To this end, we compare TSEIT's build time to the one of the reference points. To avoid biasing our results, we shuffle all time series before inserting them, i. e., we insert them in a random order.

Figure 6.6 plots the build time of our TSEIT data structure as well as of TWIST, contingent on the size of the data set. UCR and the sequential scan do not build any index, so they are not part of this experiment. The results suggest that the build times of TWIST are quadratic and might rather be an argument against this approach. This is important as TWIST is second behind TSEIT according to Experiment 2.

In contrast to TWIST, the build times for our TSEIT approach appear to be linear. Building the TSEIT index for a data set of 5 million times series in less than 5 minutes is not constraining for most purposes. Thus, our insertion benchmark reveals that TSEIT also is suitable to index large sets of time series.

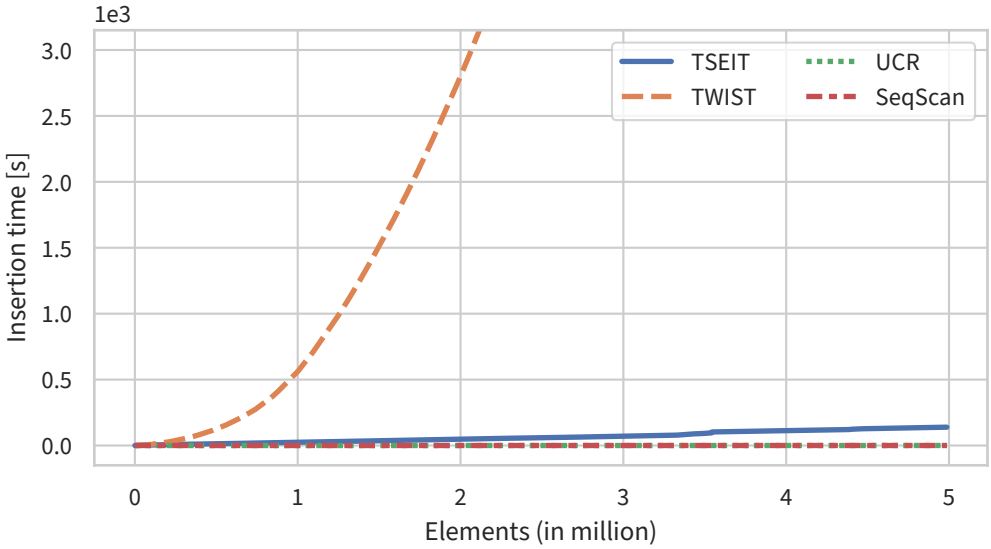


Figure 6.6: Build times for the English 2-gram corpus.

6.2.5 Results

Based on Experiment 2, we conclude that TSEIT consistently results in the lowest run times. This is because TSEIT is able to prune large parts of the data by lower bounding and exclusion of child nodes. On average, TSEIT traverses only small parts of the index, indicated by the small number of LBG computations. Moreover, due to partitioning, it also requires the fewest DTW computations. Our results also indicate that TSEIT's optimal configuration does not converge towards TWIST or UCR.

Finally, studying the build times reveals that indexing even large sets with millions of time series is not a problem for TSEIT. In contrast, the quadratic costs of inserting a time series with the next fastest approach (TWIST) might be problematic for various use cases.

6.3 Summary

In this chapter, we address the problem of efficient similarity search in time series data. In contrast to existing work, our approach features similarity search in an arbitrary time interval. We present the idea of hierarchically structured envelopes and, based on it, propose a novel tree-like data structure, named TSEIT. TSEIT is a search tree with an envelope for each node that wraps all time series of the subtree of the node. Time series envelopes allow to search the tree given a time series as similarity query and a time interval. Moreover, TSEIT combines various pruning techniques from literature. We are first to systematically consider combinations of pruning techniques for time series. Our evaluation systematically compares the performance with our data structure to the ones of state-of-the-art approaches. The run time of our reference points is either dominated by searching the data structure or by distance computations. According to our experiments, we achieve the best query performance with a compromise between minimizing the costs of searching the data structure and the number of distance computations. Our data structure features a

parameter to trade off these two aspects. All this reduces the query times of TSEIT by up to 50 % in comparison to the reference points. TSEIT is a major component to efficiently access temporal text corpora, especially when the access is limited to subperiods.

7 Conclusions

In this thesis, we designed an information system to query and efficiently search large temporal text corpora, like the Google Books Ngram Corpus. The intended purpose of our system is to study conceptual history, especially by testing hypotheses empirically on large amounts of books. To realize such a system, we proposed the technical foundations to (1) query Koselleck's information needs, (2) optimize queries for co-occurring words, and (3) provide efficient interval-focused similarity search. We summarize our contributions and highlight their value for the overall system in the following.

We started our research by identifying the information needs in conceptual history. This is an interdisciplinary task that we accomplished in cooperation with philosophers that have expertise in conceptual history. Together with these experts, we studied the works of the famous conceptual historian Reinhart Koselleck and identified information needs in conceptual history (Chapter 4). To formulate these information needs as queries for an information system, we defined a query algebra for temporal text corpora, named CHQL. Queries formulated in CHQL allow testing conceptual history hypotheses on large digital corpora. This includes hypotheses that had to be tested by manual literature research until now. We empirically evaluated our algebra by formulating major hypotheses of conceptual history in CHQL and tested them on the Google Books Ngram Corpus. Our interdisciplinary study shows empirical support for existing hypotheses.

After designing CHQL and testing some hypotheses, we noticed two query types that run inefficiently. Since both query types appear quite often, they were the performance bottleneck in the system. We overcame these performance issues with the following two contributions.

One frequently used query type is the co-occurrence query, i. e., to query for a set of words that co-occur with a given word or pattern in the ngrams. To optimize this kind of query, we developed a pruning technique for suffix trees, namely Thin Suffix Tree (TST). TST provides accurate cardinality estimations when using co-occurrence selection predicates (Chapter 5). To that, TST removes characters with low information content from all strings before adding the strings to the tree. Our pruning method is inspired by the redundancy of natural language. This means, humans are usually able to read a word with a typo or a missing character. We found that suffix trees are very often able to uniquely identify words when removing a few characters from the input strings. More specifically, character removal has two effects on the tree: (1) the strings become shorter and (2) it may cause collisions when two or more preimage strings map to the exact image string. The first effect saves memory while the second effect produces estimation errors. However, our evaluation shows that our pruning method produces hardly any error when reducing the memory consumption by half.

Another frequently used query type is the similarity search in arbitrary time intervals. To provide efficient data access for this kind of query, we presented the Time Series Envelopes

Index Tree (TSEIT). TSEIT is a time series index that generalizes existing approaches from literature (Chapter 6). We found that the run time of existing approaches is either dominated by searching the data structure or by distance computations between time series. Both aspects are related since searching the data structure usually reduces the number of necessary distance computations. Furthermore, the cost for distance computations depends on the length of the time series. Hence, this cost varies for interval-focused similarity search, where the time series length depends on the query. All told, TSEIT provides a balance between those two cost aspects and, thus, works particularly efficiently for interval-focused similarity search. We achieve the best query performance with a compromise between minimizing the costs of searching the data structure and the number of distance computations. Our evaluation shows that our generalization reduces the query times by up to 50 % compared to the original approaches.

Altogether, our three contributions shape the technical foundations to implement an information system to support studying conceptual history on the basis of temporal text corpora, i. e., millions of books. As part of an interdisciplinary research project, we implemented such a system and also produced insights on the application side. In cooperation with conceptual history experts, we used our system to test real-world hypotheses in conceptual history on the Google Books Ngram Corpus. We published a function description of our system as well as novel insights into conceptual history [Wil+19b; Wil+19c] on the flagship *Digital Humanities conference*[1], a competitive conference in the intersection of digital technologies and the disciplines of the humanities.

To conclude, the availability of large temporal text corpora and our research work on an information system to query, search, and analyze these corpora enable researchers to systematically study the origin and history of concepts—which has never been possible before. Most notably, this enables conceptual historians to test fundamental hypotheses with large amounts of data empirically. As a result, our system gives researchers new insights into temporal text corpora and allows them to gain new knowledge about the evolution of language.

We conclude this thesis with possible next steps and future research directions based on our findings and contributions.

7.1 Outlook

In this thesis, we showed how to design an information system to examine language changes using a temporal text corpus. During our research, we came across several ideas and open questions for future work. This section gives an outlook on questions that are related to our work.

Discover Unknown Concept Changes. Our system allows conceptual history examinations on large temporal text corpora. Conceptual historians can use our system to analyze known semantic changes of concepts. For example, we analyzed the change of "emancipation" from its previous relation to "Catholics" to its present relation to "women" (cf. Example 1.1). With

[1] For more information on the Digital Humanities conference (DH), see `https://dh2019.adho.org`.

such specific analyses, it is very improbable to find unknown semantic changes. To identify unknown changes on large corpora, one needs an automated discovery process. Beyond this thesis, we did work on change detection on the Google Books Ngram Corpus [Eng+19]. We achieved promising results to detect semantic changes of words since we were able to detect existing and also novel word changes in the corpus. Using change detection may also be interesting to automatically detect changes in conceptual history, where one has to consider both the words and their underlying concepts (cf. Sections 2.1.5 and 4.5). This leads to the questions of whether one can generalize change detection to concepts and how to integrate such a feature into our system, i. e., into our query algebra CHQL.

Data Cleansing for Temporal Text Corpora. The Google Books Ngram Corpus is a huge, automatically constructed corpus that contains errors, e. g., optical character recognition errors. Such errors can influence the examination results, especially when analyzing marginal language changes or rare words. To produce reliable results, a clean and consistent database is desirable. The process to detect and correct errors in a database is data cleansing. Data cleansing methods, however, mostly depend on the application for which the data will be used. At this point, several questions emerge. One question is which methods are required to detect errors in a text corpus. A follow-up question is how to correct detected errors. Furthermore, it is interesting to know whether different objects of conceptual history examinations require different detection or correction methods. Answering these questions is an interdisciplinary task and requires the expertise of data scientists and conceptual historians. All things considered, data cleansing methods for temporal text corpora might increase the reliability of the results of our system.

A Compact Representation of High-dimensional Temporal Data. The count values of a temporal text corpus depend on both the ngram and the year. Basically, there are two physical representations to store such data: horizontally, which means to build 2-tuples of ngram and its frequency time series, or vertically, which means to build 3-tuples of ngram, year, and count. In general, the horizontal data model is more compact than the vertical data model since it avoids storing the year for each count value. Therefore, we decide to use the horizontal data model (cf. Section 4.3). However, our data model stores each count value as a single integer value. In literature, many time series compression techniques and time series transformations exist, like the Fourier transform or wavelets. Most of these approaches also work on multi-dimensional data. Now, the question arises to what extent these approaches can reduce the storage requirements of a temporal text corpus. In addition, it is interesting to study how good multi-dimensional approaches can handle the high-dimensional Google Books Ngram Corpus with its billions of ngrams. A further question is whether a concise corpus representation improves the data selection speed of our system.

Query-based Corpus Sampling. This thesis considers exact processing of queries on a temporal text corpus. To further reduce the query run times significantly, one may allow approximate results. This requires the user to accept a slight inaccuracy of the results in exchange for noticeably shorter run times. One promising way to achieve this is to sample

the corpus. For example, if we leave out 10 % of the rarest ngrams, we may have almost no effect on the end result but likely save 10 % of the query run time. Currently, it is an open question to what extent different sampling methods influence run time and accuracy of different query types in CHQL. For example, some queries may be more accurate when using the most common words, while others may be more accurate when using other sampling methods, like random sampling. To that end, one should systematically investigate the interrelation between query type, query run time, and result accuracy. With this knowledge, one may be able to design and implement a query execution engine that automatically samples the corpus depending on the query type. In other words, the query execution engine should choose the best sampling strategy depending on the received query. In addition, a sampling approach enables the user to manually control the trade-off between run time and result accuracy for each query.

Query Log Analyses to Optimize Intermediate Result Usage. A query log is a record of queries that users have sent to a database system, including the timestamp of the query arrival. Such query logs contain a wealth of information, like the user's topical interests or temporal search behavior. One can analyze query logs to detect more complex information. For example, a query log might provide interesting semantic relationships between queries or allows one to learn the user's querying patterns to predict the probable next queries. Due to our research and implementation of our information system, it is now possible to create query logs on how users explore temporal text corpora. With such query logs available, several questions arise on both the application side and the technical side. On the application side, analyzing this data may help to better understand similarities and differences in the research methodology of conceptual history and help to establish a consistent methodology in this field. On the technical side, one research question is if and how accurate one can predict the next queries. This may allow managing intermediate results ahead of time, i. e., before receiving the query. Suppose one can predict the next query accurately enough. In that case, one might be able to decide which intermediate results to keep and which to precompute to accelerate the probable next queries best.

Bibliography

[AF14] Julian Arz and Johannes Fischer. "LZ-Compressed String Dictionaries". In: *Data Compression Conference (DCC '14)*. IEEE, Mar. 2014. DOI: 10.1109/dcc.2014.36.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. "Efficient Similarity Search in Sequence Databases". In: *Foundations of Data Organization and Algorithms (FODO '93)*. Springer Berlin Heidelberg, 1993, pp. 69–84. DOI: 10.1007/3-540-57301-1_5.

[Agr+95] Rakesh Agrawal, King-Ip Lin, Harpreet Singh Sawhney, and Kyuseok Shim. "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases". In: *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 490–501. ISBN: 1558603794. DOI: 10.5555/645921.673155.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Reading, Mass: Addison-Wesley, 1995. ISBN: 9780201537710.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays". In: *Journal of Discrete Algorithms (JDA)* 2.1 (Mar. 2004), pp. 53–86. ISSN: 1570-8667. DOI: 10.1016/s1570-8667(03)00065-0.

[AM93] Elizabeth Shaw Adams and Arnold Charles Meltzer. "Trigrams As Index Element in Full Text Retrieval: Observations and Experimental Results". In: *Proceedings of the 1993 ACM conference on Computer science (CSC '93)*. New York, NY, USA: ACM Press, 1993, pp. 433–439. DOI: 10.1145/170791.170891.

[AN93] Arne Andersson and Stefan Nilsson. "Improved behaviour of tries by adaptive branching". In: *Information Processing Letters* 46.6 (July 1993), pp. 295–300. DOI: 10.1016/0020-0190(93)90068-k.

[AN94] Arne Andersson and Stefan Nilsson. "Faster Searching in Tries and Quadtrees—An Analysis of Level Compression". In: *Second Annual European Symposium on Algorithms (ESA '94)*. Springer Berlin Heidelberg, 1994, pp. 82–93. DOI: 10.1007/bfb0049399.

[AN95] Arne Andersson and Stefan Nilsson. "Efficient Implementation of Suffix Trees". In: *Software: Practice and Experience (SPE '95)* 25.2 (Feb. 1995), pp. 129–141. DOI: 10.1002/spe.4380250203.

[And03] Niels Åkerstrøm Andersen. *Discursive Analytical Strategies: Understanding Foucault, Koselleck, Laclau, Luhmann*. Policy Press, Jan. 29, 2003. 160 pp. ISBN: 1861344392.

[Ass+08] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. "The TS-Tree: Efficient Time Series Search and Retrieval". In: *Proceedings of the 11th international conference on Extending database technology Advances in database technology (EDBT '08)*. ACM Press, 2008, pp. 252–263. DOI: 10.1145/1353343.1353376.

[Aßf+07] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. "Interval-Focused Similarity Search in Time Series Databases". In: *Advances in Databases: Concepts, Systems and Applications (DASFAA '07)*. Springer Berlin Heidelberg, 2007, pp. 586–597. DOI: 10.1007/978-3-540-71703-4_50.

[BCK04] Otto Brunner, Werner Conze, and Reinhart Koselleck, eds. *Geschichtliche Grundbegriffe Bände 1–8: Historisches Lexikon zur politisch-sozialen Sprache in Deutschland*. Vol. 1–8. Klett-Cotta, Sept. 1, 2004. 9000 pp. ISBN: 3608915001.

[Bec+90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles". In: *ACM SIGMOD Record* 19.2 (May 1990), pp. 322–331. DOI: 10.1145/93605.98741.

[BEH89] Anselm Blumer, Andrzej Ehrenfeucht, and David Haussler. "Average sizes of suffix trees and DAWGs". In: *Discrete Applied Mathematics* 24.1-3 (1989), pp. 37–45. DOI: 10.1016/0166-218x(92)90270-k.

[BFG17] Philip Bille, Finn Fernstrøm, and Inge Li Gørtz. "Tight Bounds for Top Tree Compression". In: *String Processing and Information Retrieval*. Springer International Publishing, 2017, pp. 97–102. DOI: 10.1007/978-3-319-67428-5_9.

[Bil+15] Philip Bille, Inge Li Gørtz, Gad Menahem Landau, and Oren Weimann. "Tree compression with top trees". In: *Information and Computation* 243 (Aug. 2015), pp. 166–177. DOI: 10.1016/j.ic.2014.12.012.

[Bin+19] Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. "COBS: A Compact Bit-Sliced Signature Index". In: *26th International Symposium on String Processing and Information Retrieval (SPIRE '19)*. Springer International Publishing, 2019, pp. 285–303. DOI: 10.1007/978-3-030-32686-9_21.

[Bla12] Andreas Blank. *Prinzipien des lexikalischen Bedeutungswandels am Beispiel der romanischen Sprachen*. Walter de Gruyter GmbH, Oct. 24, 2012. 549 pp. ISBN: 3110931605.

[Blo70] Burton Howard Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692.

[Blu+85] Anselm Blumer, Janet Andrea Blumer, David Haussler, Andrzej Ehrenfeucht, Mu-Tian Chen, and Joel I. Seiferas. "The smallest automation recognizing the subwords of a text". In: *Theoretical Computer Science* 40 (1985), pp. 31–55. DOI: 10.1016/0304-3975(85)90157-4.

[Boy11] Leonid Boytsov. "Indexing Methods for Approximate Dictionary Searching: Comparative Analysis". In: *ACM Journal of Experimental Algorithmics* 16.1.1 (May 2011). DOI: 10.1145/1963190.1963191.

[Bro+92] Peter F. Brown, Vincent J. Della Pietra, Robert Leroy Mercer, Stephen Andrew Della Pietra, and Jennifer C. Lai. "An Estimate of an Upper Bound for the Entropy of English". In: *Comput. Linguist.* 18.1 (Mar. 1992), pp. 31–40. ISSN: 0891-2017. DOI: 10.5555/146680.146685.

[CF99] Kin-Pong Chan and Ada Wai-Chee Fu. "Efficient Time Series Matching by Wavelets". In: *Proceedings 15th International Conference on Data Engineering (ICDE '99)*. IEEE, 1999, pp. 126–133. ISBN: 978-0-7695-0071-3. DOI: 10.1109/icde.1999.754915.

[CFY03] Franky Kin-Pong Chan, Ada Wai-Chee Fu, and Clement Yu. "Haar Wavelets for Efficient Similarity Search of Time-Series: With and Without Time Warping". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE '03)* 15.3 (May 2003), pp. 686–705. DOI: 10.1109/tkde.2003.1198399.

[CGG04] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. "Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem". In: *Proceedings. 20th International Conference on Data Engineering (ICDE '04)*. IEEE Comput. Soc, 2004. DOI: 10.1109/icde.2004.1319999.

[CGW16] Patrick Hagge Cording, Pawel Gawrychowski, and Oren Weimann. "Bookmarks in Grammar-Compressed Strings". In: *23rd International Symposium on String Processing and Information Retrieval (SPIRE '16)*. Springer International Publishing, 2016, pp. 153–159. DOI: 10.1007/978-3-319-46049-9_15.

[CKL03] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. "Probabilistic Discovery of Time Series Motifs". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '03)*. ACM Press, 2003, pp. 493–498. DOI: 10.1145/956750.956808.

[Cla+12] Francisco Claude, Gonzalo Navarro, Hannu Peltola, Leena Salmela, and Jorma Tarhio. "String matching with alphabet sampling". In: *Journal of Discrete Algorithms* 11 (Feb. 2012), pp. 37–50. DOI: 10.1016/j.jda.2010.09.004.

[CN02] Maxime Crochemore and Gonzalo Navarro. "Improved Antidictionary Based Compression". In: *Proceedings of the 12th International Conference of the Chilean Computer Science Society (SCCC '02)*. IEEE Computer Society, 2002, pp. 7–13. DOI: 10.1109/sccc.2002.1173168.

[Cod70] Edgar Frank Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. DOI: 10.1145/362384.362685.

[Con13] The Library of Congress. *The Contextual Query Language*. Aug. 30, 2013. URL: https://www.loc.gov/standards/sru/cql/.

[Cor+11] Graham Cormode, Minos Garofalakis, Peter Haas, and Chris Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". In: *Foundations and Trends in Databases* 4.1-3 (2011), pp. 1–294. DOI: 10.1561/1900000004.

[Cro+00] Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. "Data Compression Using Antidictionaries". In: *Proceedings of the IEEE* 88.11 (Nov. 2000), pp. 1756–1768. DOI: 10.1109/5.892711.

[Cry92] David Crystal. *An encyclopedic dictionary of language and languages*. Oxford, UK Cambridge, Mass., USA: Blackwell, 1992. ISBN: 9780631176527.

[CT06] Thomas M. Cover and Joy Aloysius Thomas. *Elements of Information Theory*. John Wiley & Sons, Sept. 8, 2006. 792 pp. ISBN: 0471241954.

[DCW93] John Joseph Darragh, John Gerald Cleary, and Ian Hugh Witten. "Bonsai: A compact representation of trees". In: *Software: Practice and Experience (SPE '93)* 23.3 (Mar. 1993), pp. 277–291. DOI: 10.1002/spe.4380230305.

[DN00] Bogdan Dorohonceanu and Craig Graham Nevill-Manning. "Accelerating Protein Classification Using Suffix Trees". In: *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB '00)*. Vol. 8. AAAI Press, Aug. 2000, pp. 128–133. DOI: 10.5555/645635.660986.

[Du+08] Yi Du, Chuanqun Jiang, Wen-an Tan, Detang Lu, and Daolun Li. "Effective Subsequence Matching in Compressed Time Series". In: *2008 Third International Conference on Pervasive Computing and Applications (ICPCA '08)*. IEEE, Oct. 2008, pp. 922–926. DOI: 10.1109/icpca.2008.4783742.

[EN16] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Hoboken, NJ: Pearson, 2016. ISBN: 9780133970777.

[Eng+19] Adrian Englhardt, Jens Willkomm, Martin Schäler, and Klemens Böhm. "Improving Semantic Change Analysis by Combining Word Embeddings and Word Frequencies". In: *International Journal on Digital Libraries (IJDL '19)* (May 2019), pp. 1–18. DOI: 10.1007/s00799-019-00271-6.

[Far97] Martin Farach. "Optimal Suffix Tree Construction with Large Alphabets". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science (SFCS '97)*. IEEE Comput. Soc, 1997. DOI: 10.1109/sfcs.1997.646102.

[FB16] Alexander Friedrich and Chris Biemann. "Digitale Begriffsgeschichte?: Methodologische Überlegungen und Exemplarische Versuche am Beispiel Moderner Netzsemantik". In: *Forum Interdisziplinäre Begriffsgeschichte (FIB)* 2 (2016), pp. 78–96. ISSN: 2195-0598.

[Fer+07] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. "Compressed Representations of Sequences and Full-Text Indexes". In: *ACM Transactions on Algorithms* 3.2 (May 2007), p. 20. DOI: 10.1145/1240233.1240243.

[Fer+08] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. “On Searching Compressed String Collections Cache-Obliviously”. In: *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS ’08)*. ACM Press, June 2008, pp. 181–190. DOI: 10.1145/1376916.1376943.

[Fer+09] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. “Compressed Text Indexes: From Theory to Practice”. In: *ACM Journal of Experimental Algorithmics* 13.12 (Feb. 2009). DOI: 10.1145/1412228.1455268.

[FG19] Gabriele Fici and Paweł Gawrychowski. “Minimal Absent Words in Rooted and Unrooted Trees”. In: *26th International Symposium on String Processing and Information Retrieval (SPIRE ’19)*. Springer International Publishing, 2019, pp. 152–161. DOI: 10.1007/978-3-030-32686-9_11.

[FH08] Martin Fiala and Jan Holub. “DCA Using Suffix Arrays”. In: *Data Compression Conference (DCC ’08)*. IEEE, Mar. 2008. DOI: 10.1109/dcc.2008.95.

[Fir57] John Rupert Firth. “A synopsis of linguistic theory, 1930–1955”. In: *Studies in Linguistic Analysis* (1957), pp. 1–32.

[FM05] Paolo Ferragina and Giovanni Manzini. “Indexing Compressed Text”. In: *Journal of the ACM* 52.4 (July 2005), pp. 552–581. DOI: 10.1145/1082036.1082039.

[Fri06] Gerd Fritz. *Historische Semantik*. OCLC: 962752812. Stuttgart: J.B. Metzler, 2006. ISBN: 978-3-476-01408-5. DOI: 10.1007/978-3-476-01408-5.

[Fri11] Gerd Fritz. *Einführung in die historische Semantik*. Walter de Gruyter GmbH, Dec. 22, 2011. 256 pp. ISBN: 3110942372. DOI: 10.1515/9783110942378.

[FRM94] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. “Fast Subsequence Matching in Time-Series Databases”. In: *Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD ’94)*. ACM Press, 1994, pp. 419–429. ISBN: 978-0-89791-639-4. DOI: 10.1145/191839.191925.

[Fu+07] Ada Wai-Chee Fu, Eamonn Keogh, Leo Yung Hang Lau, Chotirat Ann Ratanamahatana, and Raymond Chi-Wing Wong. “Scaling and time warping in time series querying”. In: *The VLDB Journal* 17.4 (Mar. 2007), pp. 899–921. DOI: 10.1007/s00778-006-0040-z.

[GG88] Zvi Galil and Raffaele Giancarlo. “Data Structures and Algorithms for Approximate String Matching”. In: *Journal of Complexity* 4.1 (Mar. 1988), pp. 33–72. DOI: 10.1016/0885-064x(88)90008-8.

[Gil+15] Myeong-Seon Gil, Bum-Soo Kim, Mi-Jung Choi, and Yang-Sae Moon. “Fast Index Construction for Distortion-Free Subsequence Matching in Time-Series Databases”. In: *2015 International Conference on Big Data and Smart Computing (BIGCOMP ’15)*. IEEE, Feb. 2015, pp. 130–135. ISBN: 978-1-4799-7303-3. DOI: 10.1109/35021bigcomp.2015.7072822.

[GO15] Roberto Grossi and Giuseppe Ottaviano. "Fast Compressed Tries through Path Decompositions". In: *Journal of Experimental Algorithmics* 19 (Feb. 2015), pp. 11–120. DOI: 10.1145/2656332.

[Gog+14] Simon Gog, Alistair Moffat, Jason Shane Culpepper, Andrew Turpin, and Anthony Wirth. "Large-Scale Pattern Search Using Reduced-Space On-Disk Suffix Arrays". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE '14)* 26.8 (Aug. 2014), pp. 1918–1931. DOI: 10.1109/tkde.2013.129.

[GR15] Szymon Grabowski and Marcin Raniszewski. "Sampling the Suffix Array with Minimizers". In: *String Processing and Information Retrieval*. Springer International Publishing, 2015, pp. 287–298. DOI: 10.1007/978-3-319-23826-5_28.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, May 1, 1997. 554 pp. ISBN: 0521585198.

[Gut84] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84)*. ACM Press, 1984, pp. 47–57. DOI: 10.1145/602259.602266.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching". In: *SIAM Journal on Computing* 35.2 (Jan. 2005), pp. 378–407. DOI: 10.1137/s0097539702402354.

[Hai17] Shalin Hai-Jew. *Data Analytics in Digital Humanities*. Multimedia Systems and Applications. Springer-Verlag GmbH, May 7, 2017. ISBN: 3319544985.

[Har54] Zellig Sabbetai Harris. "Distributional Structure". In: *WORD* 10.2-3 (Aug. 1954), pp. 146–162. DOI: 10.1080/00437956.1954.11659520.

[HD80] Patrick A. V. Hall and Geoff R. Dowling. "Approximate String Matching". In: *ACM Computing Surveys* 12.4 (Dec. 1980), pp. 381–402. DOI: 10.1145/356827.356830.

[Her99] Hans Jürgen Heringer. *Das höchste der Gefühle: Empirische Studien zur distributiven Semantik*. Stauffenburg Verlag, Jan. 1, 1999. 250 pp. ISBN: 3860577166.

[Hig10] Colin De La Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, Apr. 1, 2010. 417 pp. ISBN: 0521763169.

[HLJ16a] William L. Hamilton, Jure Leskovec, and Dan Jurafsky. "Cultural Shift or Linguistic Drift? Comparing Two Computational Measures of Semantic Change". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP '16)*. Association for Computational Linguistics, Nov. 2016. DOI: 10.18653/v1/d16-1229.

[HLJ16b] William L. Hamilton, Jure Leskovec, and Dan Jurafsky. "Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP '16)*. Nov. 2016.

[HO13] Bo-June (Paul) Hsu and Giuseppe Ottaviano. “Space-Efficient Data Structures for Top-k Completion”. In: *Proceedings of the 22nd international conference on World Wide Web (WWW ’13)*. ACM Press, 2013, pp. 583–594. DOI: 10.1145/2488388.2488440.

[HR15] Lorenz Hübschle-Schneider and Rajeev Raman. “Tree Compression with Top Trees Revisited”. In: *14th International Symposium on Experimental Algorithms (SEA ’15)*. Springer International Publishing, 2015, pp. 15–27. DOI: 10.1007/978-3-319-20086-6_2.

[HT71] Te Chiang Hu and Alan Curtiss Tucker. “Optimal Computer Search Trees and Variable-Length Alphabetical Codes”. In: *SIAM Journal on Applied Mathematics* 21.4 (Dec. 1971), pp. 514–532. DOI: 10.1137/0121057.

[Huf52] David Albert Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the Institute of Radio Engineers (IRE)* 40.9 (Sept. 1952), pp. 1098–1101. DOI: 10.1109/jrproc.1952.273898.

[Jak+10] Miloš Jakubíček, Adam Kilgarriff, Diana McCarthy, and Pavel Rychlý. “Fast Syntactic Searching in Very Large Corpora for Many Languages”. In: *Pacific Asia Conference on Language, Information and Computation (PACLIC ’10)*. Tokyo, 2010, pp. 741–747.

[Keo+01] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. “Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases”. In: *Knowledge and Information Systems (KAIS ’01)* 3.3 (Aug. 2001), pp. 263–286. DOI: 10.1007/pl00011669.

[Kie+00] John C. Kieffer, En-Hui Yang, Greg J. Nelson, and Pamela Cosman. “Universal Lossless Compression Via Multilevel Pattern Matching”. In: *IEEE Transactions on Information Theory* 46.4 (July 2000), pp. 1227–1245. DOI: 10.1109/18.850665.

[KJF97] Flip Korn, Hosagrahar Visvesvaraya Jagadish, and Christos Faloutsos. “Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences”. In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data (SIGMOD ’97)*. New York, NY, USA: ACM Press, 1997, pp. 289–300. ISBN: 978-0-89791-911-1. DOI: 10.1145/253260.253332.

[KM10] Carl-Christian Kanne and Guido Moerkotte. “Histograms Reloaded: The Merits of Bucket Diversity”. In: *Proceedings of the 2010 international conference on Management of data (SIGMOD ’10)*. ACM Press, 2010, pp. 663–674. DOI: 10.1145/1807167.1807239.

[KMF17] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. “Practical Implementation of Space-Efficient Dynamic Keyword Dictionaries”. In: *String Processing and Information Retrieval*. Springer International Publishing, 2017, pp. 221–233. DOI: 10.1007/978-3-319-67428-5_19.

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming*. Addison-Wesley, Apr. 24, 1998. 800 pp. ISBN: 0201896850.

[Kol12] Kathrin Kollmeier. "Begriffsgeschichte und Historische Semantik". In: *Docupedia-Zeitgeschichte* (2012). DOI: 10.14765/ZZF.DOK.2.257.V2.

[Kol16] Olga Kolesnikova. "Survey of Word Co-occurrence Measures for Collocation Detection". In: *Computación y Sistemas* 20.3 (Sept. 2016). DOI: 10.13053/cys-20-3-2456.

[Kos02] Reinhart Koselleck. *The Practice of Conceptual History: Timing History, Spacing Concepts (Cultural Memory in the Present)*. Stanford University Press, July 2, 2002. 384 pp. ISBN: 0804743053.

[Kos04] Reinhart Koselleck. *Futures Past: On the Semantics of Historical Time*. Columbia University Press, 2004. 344 pp. ISBN: 0231127715.

[Kos06] Reinhart Koselleck. *Begriffsgeschichten: Studien zur Semantik und Pragmatik der politischen und sozialen Sprache*. Frankfurt am Main: Suhrkamp, 2006. 569 pp. ISBN: 9783518584637.

[KP86] Peter Kirschenhofer and Helmut Prodinger. "Some further results on digital search trees". In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1986, pp. 177–185. DOI: 10.1007/3-540-16761-7_67.

[KP88] Peter Kirschenhofer and Helmut Prodinger. "Further results on digital search trees". In: *Theoretical Computer Science* 58.1-3 (June 1988), pp. 143–154. DOI: 10.1016/0304-3975(88)90023-0.

[KPC01] Sang-Wook Kim, Sanghyun Park, and Wesley Wei-Chin Chu. "An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases". In: *Proceedings 17th International Conference on Data Engineering (ICDE '01)*. IEEE Comput. Soc, 2001, pp. 607–614. DOI: 10.1109/icde.2001.914875.

[KR05] Eamonn Keogh and Chotirat Ann Ratanamahatana. "Exact indexing of dynamic time warping". In: *Knowledge and Information Systems (KAIS '05)* 7.3 (Mar. 2005), pp. 358–386. ISSN: 0219-3116. DOI: 10.1007/s10115-004-0154-9.

[Kro15] Paul Kroeger. *Analyzing Grammar: An Introduction*. Cambridge University Press, Dec. 1, 2015. 384 pp. ISBN: 0521016533.

[KS10] Maciej Krawczak and Grazyna Szkatula. "Time series envelopes for classification". In: *2010 5th IEEE International Conference Intelligent Systems (IS '10)*. IEEE, July 2010, pp. 156–161. ISBN: 978-1-4244-5163-0. DOI: 10.1109/is.2010.5548371.

[KU96] Juha Kärkkäinen and Esko Ukkonen. "Sparse Suffix Trees". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996, pp. 219–230. DOI: 10.1007/3-540-61332-3_155.

[KVI96] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. "Estimating Alphanumeric Selectivity in the Presence of Wildcards". In: *ACM SIGMOD Record* 25.2 (June 1996), pp. 282–293. DOI: 10.1145/235968.233341.

[Lei+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015), pp. 204–215. DOI: 10.14778/2850583.2850594.

[Li+02] Ai-Jun Li, Yun-Hui Liu, Ying-Jian Qi, and Si-Wei Luo. "An approach for fast subsequence matching through KMP algorithm in time series databases". In: *Proceedings. International Conference on Machine Learning and Cybernetics (ICMLC '02)*. IEEE, 2002, pp. 1292–1295. DOI: 10.1109/icmlc.2002.1167412.

[Li+15a] Dong Li, Qixu Zhang, Xiaochong Liang, Jida Guan, and Yang Xu. "Selectivity Estimation for String Predicates Based on Modified Pruned Count-Suffix Tree". In: *Chinese Journal of Electronics (CJE '15)* 24.1 (Jan. 2015), pp. 76–82. DOI: 10.1049/cje.2015.01.013.

[Li+15b] Yitan Li, Linli Xu, Fei Tian, Liang Jiang, Xiaowei Zhong, and Enhong Chen. "Word Embedding Revisited: A New Representation Learning and Explicit Matrix Factorization Perspective". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI '15)*. AAAI Press, 2015, pp. 3650–3656.

[Lin+12] Yuri Lin, Jean-Baptiste Michel, Erez Lieberman Aiden, Jon Orwant, Will Brockman, and Slav Orlinov Petrov. "Syntactic Annotations for the Google Books Ngram Corpus". In: *Annual Meeting of the Association for Computational Linguistics (ACL '12)*. Jeju Island, Korea: Association for Computational Linguistics, 2012, pp. 169–174. DOI: 10.5555/2390470.2390499.

[LLM04] Quanzhong Li, Inés Fernando Vega López, and Bongki Moon. "Skyline Index for Time Series Data". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE '04)* 16.6 (June 2004), pp. 669–684. DOI: 10.1109/tkde.2004.14.

[LM00] Niklas Jesper Larsson and Alistair Moffat. "Off-Line Dictionary-Based Compression". In: *Proceedings of the IEEE* 88.11 (Nov. 2000), pp. 1722–1732. DOI: 10.1109/5.892708.

[Lot02] M. Lothaire, ed. *Combinatorics on Words*. Cambridge University Press, Jan. 13, 2002. 260 pp. ISBN: 0521599245.

[LPK07] Seung-Hwan Lim, Heejin Park, and Sang-Wook Kim. "Using multiple indexes for efficient subsequence matching in time-series databases". In: *Information Sciences* 177.24 (Dec. 2007), pp. 5691–5706. ISSN: 0020-0255. DOI: 10.1016/j.ins.2007.07.004.

[LR13] Xiao-Ying Liu and Chuan-Lun Ren. "Fast subsequence matching under time warping in time-series databases". In: *International Conference on Machine Learning and Cybernetics (ICMLC '13)*. IEEE, July 2013, pp. 1584–1590. ISBN: 978-1-4799-0260-6. DOI: 10.1109/icmlc.2013.6890855.

[LS03] Alberto Lerner and Dennis Shasha. "AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments". In: *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB '03)*. VLDB Endowment, 2003, pp. 345–356. DOI: 10.1016/b978-012722442-8/50038-0.

[Mai83] David Maier. *Theory of Relational Databases*. Rockville, Md: Computer Science Press, 1983. ISBN: 0914894420.

[McA98] Tom McArthur, ed. *The concise Oxford companion to the English language*. New York: Oxford University Press, 1998. ISBN: 9780192800619.

[MGR98] Hamid Moradi, Jerzy Witold Grzymala-Busse, and James A. Roberts. "Entropy of English text: Experiments with humans and a machine learning system based on rough sets". In: *Information Sciences* 104.1-2 (Jan. 1998), pp. 31–47. DOI: 10.1016/s0020-0255(97)00074-1.

[Mic+10] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin Andreas Nowak, and Erez Lieberman Aiden. "Quantitative Analysis of Culture Using Millions of Digitized Books". In: *Science* 331.6014 (Dec. 2010), pp. 176–182. DOI: 10.1126/science.1199644.

[Min+12] Gary Miner, John Elder, Andrew Fast, Thomas Hill, Robert Nisbet, and Dursun Delen. *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications*. Amsterdam: Academic Press, 2012. ISBN: 9780123869791.

[MM93] Udi Manber and Gene Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM Journal on Computing (SICOMP '93)* 22.5 (Oct. 1993), pp. 935–948. DOI: 10.1137/0222058.

[MMK18] Magnus Müller, Guido Moerkotte, and Oliver Kolb. "Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses". In: *Proceedings of the VLDB Endowment* 11.9 (May 2018), pp. 1016–1028. DOI: 10.14778/3213880.3213882.

[MNS09] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors". In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pp. 982–993. DOI: 10.14778/1687627.1687738.

[Moe+14] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Boehm. "Exploiting Ordered Dictionaries to Efficiently Construct Histograms with Q-Error Guarantees in SAP HANA". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14)*. ACM Press, 2014. DOI: 10.1145/2588555.2595629.

[Mor13] Franco Moretti. *Distant Reading*. London: Verso, 2013. ISBN: 9781781681121.

[Mor68] Donald R. Morrison. "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM (JACM)* 15.4 (Oct. 1968), pp. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481.

[MS16] Ernst Müller and Falko Schmieder. *Begriffsgeschichte und historische Semantik: Ein kritisches Kompendium*. Suhrkamp Verlag AG, July 11, 2016. 1027 pp. ISBN: 3518297171.

[MS99] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press Ltd, May 28, 1999. 720 pp. ISBN: 0262133601.

[Nab05] Daniel Naber. "OpenThesaurus: Ein offenes deutsches Wortnetz". In: *Sprachtechnologie, mobile Kommunikation und linguistische Ressourcen: Beiträge zur GLDV-Tagung 2005 in Bonn*. 2005, pp. 422–433. ISBN: 9783631538746.

[NB00] Gonzalo Navarro and Ricardo Baeza-Yates. "A Hybrid Indexing Method for Approximate String Matching". In: *Journal of Discrete Algorithms* 1.1 (2000), pp. 205–239. ISSN: 1468-0904.

[NM07] Gonzalo Navarro and Veli Mäkinen. "Compressed Full-Text Indexes". In: *ACM Computing Surveys* 39.1 (Apr. 2007), p. 2. DOI: 10.1145/1216370.1216372.

[NRR10] Vit Niennattrakul, Pongsakorn Ruengronghirunya, and Chotirat Ann Ratanamahatana. "Exact Indexing for Massive Time Series Databases under Time Warping Distance". In: *Data Mining and Knowledge Discovery* 21.3 (Feb. 2010), pp. 509–541. ISSN: 1573-756X. DOI: 10.1007/s10618-010-0165-y.

[NT02] Stefan Nilsson and Mikko Tikkanen. "An Experimental Study of Compression Methods for Dynamic Tries". In: *Algorithmica* 33.1 (May 2002), pp. 19–33. DOI: 10.1007/s00453-001-0102-y.

[NW97] Craig Graham Nevill-Manning and Ian Hugh Witten. "Identifying Hierarchical Structure in Sequences: A linear-time algorithm". In: *Journal of Artificial Intelligence Research (JAIR '97)* 7 (Sept. 1997), pp. 67–82. DOI: 10.1613/jair.374.

[NZ97] Hwee Tou Ng and John Zelle. "Corpus-Based Approaches to Semantic Interpretation in Natural Language Processing". In: *AI Magazine* 18.4 (Dec. 1997), pp. 45–64. DOI: 10.1609/aimag.v18i4.1321.

[OD09] Martin O'Connor and Amar Das. "SQWRL: A Query Language for OWL". In: *Proceedings of the 6th International Conference on OWL: Experiences and Directions (OWLED '09)*. Vol. 529. Aachen, DEU: CEUR-WS.org, 2009, pp. 208–215. DOI: 10.5555/2890046.2890072.

[Ols12] Niklas Olsen. *History in the Plural: An Introduction to the Work of Reinhart Koselleck*. Berghahn Books Inc., Jan. 1, 2012. 346 pp. ISBN: 0857452959.

[OM10] Takahiro Ota and Hiroyoshi Morita. "On the Adaptive Antidictionary Code Using Minimal Forbidden Words with Constant Lengths". In: *2010 International Symposium On Information Theory & Its Applications (ISITA '10)*. IEEE, Oct. 2010. DOI: 10.1109/isita.2010.5649621.

[PR15] Andreas Poyias and Rajeev Raman. "Improved Practical Compact Dynamic Tries". In: *String Processing and Information Retrieval*. Springer International Publishing, 2015, pp. 324–336. DOI: 10.1007/978-3-319-23826-5_31.

[Pra+16] Vinodkumar Prabhakaran, William L. Hamilton, Dan McFarland, and Dan Jurafsky. "Predicting the Rise and Fall of Scientific Topics from Trends in their Rhetorical Framing". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL '16)*. Vol. 1 (Long Papers). Association for Computational Linguistics, 2016, pp. 1170–1180. DOI: 10.18653/v1/p16-1111.

[Rak+12] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '12)*. ACM Press, 2012, pp. 262–270. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339576.

[Rak+13] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. "Addressing Big Data Time Series: Mining Trillions of Time Series Subsequences Under Dynamic Time Warping". In: *ACM Transactions on Knowledge Discovery from Data (TKDD '13)* 7.3 (Sept. 2013), pp. 1–31. ISSN: 1556-4681. DOI: 10.1145/2513092.2500489.

[RGG07] Joachim Ritter, Karlfried Gründer, and Gottfried Gabriel. *Historisches Wörterbuch der Philosophie*. Vol. 1–13. Basel: Schwabe, 2007. ISBN: 9783796501159.

[Rot86] Lord Rothschild. "The distribution of English dictionary word lengths". In: *Journal of Statistical Planning and Inference* 14.2-3 (Jan. 1986), pp. 311–322. DOI: 10.1016/0378-3758(86)90169-2.

[SAB08] Guido Sautter, Cristina Abba, and Klemens Böhm. "Improved Count Suffix Trees for Natural Language Data". In: *Proceedings of the 2008 international symposium on Database engineering & applications (IDEAS '08)*. ACM Press, 2008. DOI: 10.1145/1451940.1451972.

[Sad07] Kunihiko Sadakane. "Compressed Suffix Trees with Full Functionality". In: *Theory of Computing Systems* 41.4 (Feb. 2007), pp. 589–607. DOI: 10.1007/s00224-006-1198-x.

[SC78] Hiroaki Sakoe and Seibi Chiba. "Dynamic Programming Algorithm Optimization for Spoken Word Recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing (TASSSP '78)* 26.1 (Feb. 1978), pp. 43–49. DOI: 10.1109/tassp.1978.1163055.

[Sch+13] Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. "QuEval: Beyond high-dimensional indexing à la carte". In: *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013), pp. 1654–1665. DOI: 10.14778/2556549.2556551.

[Sch15] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons Inc, Mar. 20, 2015. 784 pp. ISBN: 1119096723.

[SEW04] Bengt Sigurd, Mats Eeg-Olofsson, and Joost van Weijer. "Word length, sentence length and frequency - Zipf revisited". In: *Studia Linguistica* 58.1 (Apr. 2004), pp. 37–52. DOI: 10.1111/j.0039-3193.2004.00109.x.

[Sha48] Claude Elwood Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[SK91] Ralf Schneider and Hans-Peter Kriegel. "The TR*-tree: A new Representation of Polygonal Objects Supporting Spatial Queries and Operations". In: *Computational Geometry-Methods, Algorithms and Applications (CG '91)*. Springer Berlin Heidelberg, 1991, pp. 249–263. ISBN: 978-3-540-46459-4. DOI: 10.1007/3-540-54891-2_19.

[SL19] Ji Sun and Guoliang Li. "An End-to-End Learning-based Cost Estimator". In: (June 6, 2019). arXiv: 1906.02560v1 [cs.DB].

[Sno87] Richard Snodgrass. "The Temporal Query Language TQuel". In: *ACM Transactions on Database Systems (TODS '87)* 12.2 (June 1987), pp. 247–298. DOI: 10.1145/22952.22956.

[Sno95] Richard Thomas Snodgrass, ed. *The TSQL2 Temporal Query Language*. The Springer International Series in Engineering and Computer Science. Springer US, Aug. 31, 1995. Chap. An Algebra for TSQL2. 704 pp. ISBN: 0792396146.

[SYF05] Yasushi Sakurai, Masatoshi Yoshikawa, and Christos Faloutsos. "FTW: Fast Similarity Search under the Time Warping Distance". In: *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '05)*. ACM Press, 2005, pp. 326–337. DOI: 10.1145/1065167.1065210.

[Ukk92] Esko Ukkonen. "Approximate string-matching with q-grams and maximal matches". In: *Theoretical Computer Science* 92.1 (Jan. 1992), pp. 191–211. DOI: 10.1016/0304-3975(92)90143-4.

[Ukk95] Esko Ukkonen. "On-line Construction of Suffix Trees". In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. DOI: 10.1007/bf01206331.

[Vid+88] Enrique Vidal, Francisco Casacuberta, José Miguel Benedi, Maria José Lloret, and Hector Rulot. "On the verification of triangle inequality by dynamic time-warping dissimilarity measures". In: *Speech Communication* 7.1 (Mar. 1988), pp. 67–79. DOI: 10.1016/0167-6393(88)90022-2.

[VMS15] Luciana Vitale, Álvaro Martín, and Gadiel Seroussi. "Space-efficient representation of truncated suffix trees, with applications to Markov order estimation". In: *Theoretical Computer Science* 595 (Aug. 2015), pp. 34–45. DOI: 10.1016/j.tcs.2015.06.013.

[Vog12] Kai Vogelsang. "Conceptual History: A Short Introduction". In: *Oriens Extremus* 51 (2012), pp. 9–24. ISSN: 00305197.

[Wan+10] Kuansan Wang, Christopher Thrasher, Evelyne Viegas, Xiaolong Li, and Bojune Hsu. "An Overview of Microsoft Web N-gram Corpus and Applications". In: *Proceedings of the NAACL HLT 2010 Demonstration Session*. HLT-DEMO '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 45–48.

[War12] Claire Warwick. *Digital Humanities in Practice*. The Facet Digital Heritage Collection. London: Facet Publishing, 2012. ISBN: 9781856047661.

[Wel84] Terry Welch. "A Technique for High-Performance Data Compression". In: *Computer* 17.6 (June 1984), pp. 8–19. DOI: 10.1109/mc.1984.1659158.

[Wil+18] Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. "A Query Algebra for Temporal Text Corpora". In: *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries (JCDL '18)*. ACM Press, May 2018, pp. 183–192. DOI: 10.1145/3197026.3197044.

[Wil+19a] Jens Willkomm, Janek Bettinger, Martin Schäler, and Klemens Böhm. "Efficient Interval-focused Similarity Search under Dynamic Time Warping". In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD '19)*. ACM Press, Aug. 2019, pp. 130–139. DOI: 10.1145/3340964.3340969.

[Wil+19b] Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. "The CHQL Query Language for Conceptual History Relying on Google Books". In: *Digital Humanities 2019: Book of Abstracts (DH '19)*. 2019.

[Wil+19c] Jens Willkomm, Christoph Schmidt-Petri, Martin Schäler, Michael Schefczyk, and Klemens Böhm. "Using Ngrams to Develop a Query Algebra for Conceptual History". In: *Digital Humanities 2019: Book of Abstracts (DH '19)*. 2019.

[Wol+08] Markus Wolf, Andrea B. Horn, Matthias R. Mehl, Severin Haug, James W. Pennebaker, and Hans Kordy. "Computergestützte quantitative Textanalyse". In: *Diagnostica* 54.2 (Apr. 2008), pp. 85–98. DOI: 10.1026/0012-1924.54.2.85.

[WSB21] Jens Willkomm, Martin Schäler, and Klemens Böhm. "Accurate Cardinality Estimation of Co-occurring Words Using Suffix Trees". In: *Proceedings of the 26th International Conference on Database Systems for Advanced Applications (DASFAA '21)*. Ed. by Christian S. Jensen, Ee-Peng Lim, De-Nian Yang, Wang-Chien Lee, Vincent S. Tseng, Vana Kalogeraki, Jen-Wei Huang, and Chih-Ya Shen. Vol. 12682. Cham: Springer International Publishing, Apr. 2021, pp. 721–737. DOI: 10.1007/978-3-030-73197-7_50.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces". In: *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. San Francisco, CA, United States: Morgan Kaufmann Publishers Inc., Aug. 1998, pp. 194–205. ISBN: 978-1-55860-566-4. DOI: 10.5555/645924.671192.

[Wu+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacıgümüş, and Jeffrey F. Naughton. “Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?” In: *2013 IEEE 29th International Conference on Data Engineering (ICDE ’13)*. IEEE, Apr. 2013. DOI: 10.1109/icde.2013.6544899.

[Zel+09] Amir Zeldes, Anke Lüdeling, Julia Ritz, and Christian Chiarcos. “ANNIS: a search tool for multi-layer annotated corpora”. In: *Proceedings of Corpus Linguistics*. 2009. DOI: 10.18452/13437.

[ZL77] Jacob Ziv and Abraham Lempel. “A Universal Algorithm for Sequential Data Compression”. In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. DOI: 10.1109/tit.1977.1055714.

[ZL78] Jacob Ziv and Abraham Lempel. “Compression of Individual Sequences via Variable-Rate Coding”. In: *IEEE Transactions on Information Theory* 24.5 (Sept. 1978), pp. 530–536. DOI: 10.1109/tit.1978.1055934.