University of Alberta

Indexing and Querying Natural Language Text

by

Pirooz Chubak

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Abstract

Natural language text is a prominent source of representing and communicating information and knowledge. It is often desirable to search in granularities of text that are smaller than a document or to query the syntactic roles and relationships within syntactically annotated text sentences, often represented by parse trees. In this thesis, we study the problems of efficiently indexing and querying natural language text in the scenarios where (1) text is modelled as flat sequences of words and (2) text is modelled as collections of syntactically annotated trees.

In the first scenario, we study some of the index structures that are capable of answering the class of queries referred to here as wild card queries and perform an analysis of their performance. Our experimental results on a large class of queries from different sources (including query logs and parse trees) and with various datasets reveal some of the performance barriers of these indexes. We present Word Permuterm Index (WPI) and show that it supports a wide range of wild card queries, is quick to construct and is highly scalable. Our experimental results comparing WPI to alternative methods on a wide range of wild card queries show a few orders of magnitude performance improvement for WPI while the memory usage is kept the same for all compared systems.

In the second scenario, we study index structures and access methods that improve the performance of querying over syntactically parsed sentences. We propose a novel indexing scheme over unique subtrees as index keys. We also introduce the *root-split* coding scheme that concisely stores structural subtree information, making it possible to perform exact axes matching over subtrees. We theoretically study the properties of our coding and the limitations it imposes over query processing. Our extensive set of experiments show that *root-split* coding reduces the index size of a baseline index which stores the interval codes of all nodes by a factor of up to 5 (i.e. 80% reduction) and speeds up querying runtime by more than 6 times on average, when subtrees of sizes $1, \ldots, 5$ are indexed.

Acknowledgements

Many people have supported, helped and encouraged me during my PhD program and thesis writing. It is my great pleasure to thank them and appreciate their work.

My most sincere thanks to my supervisor, Davood Rafiei, who has always provided me with constructive feedback and support. He has done a tremendous job as a mentor and supervisor, and put a great deal of time and energy into my research and other academic and life challenges. I would also like to thank him for being understanding and supportive of me working remotely in the final year of my PhD.

My appreciation and gratitude to my examiners, Raymond T. Ng, John Newman and Denilson Barbosa for carefully reading my thesis and providing me with excellent comments and deep questions which contributed greatly to the quality of this thesis. Also thanks to Dekang Lin and Jörg Sander for reading my candidacy proposal and providing me with valuable suggestions at early stages of my PhD research.

Many thanks to Rachel Pottinger who helped me greatly with settling down in Vancouver and supported me with space and resources in UBC during the last year of my PhD. I greatfully acknowledge Dennis Shasha, Shane Bergsma, and Christopher Pinchak for providing me with their expert opinion on specific subjects related to my research. Finally, many thanks to other database faculty members Osmar Zaiane and Mario Nascimento, from whom I learned much during my PhD.

I would like to also thank a few wonderful people at the Department of Computing Science, University of Alberta. Special thanks to Edith Drummond and Frances Moore for dealing with graduate student issues and always providing me with their help. Many thanks to Steve Sutphen for always helping with technical issues beyond his duties.

My friends and peers at University of Alberta and Edmonton have always been there whenever life was too difficult as a graduate student. I would like to thank Reza Sadoddin, Reza Sherkat and Azad Shademan in the department of Computing Science. Occasional discussions on a wide range of academic and non-academic topics, made a big difference in my view and experience as a PhD student. Also, I would like to thank my wonderful friends

Iman Khosravifard, Raman Yazdani and Ali Hendi. They made my life as a graduate student much easier and less frustrating.

My deepest thanks and appreciation go to my parents, who have always encouraged me with their love and support. They never stopped caring and encouraging me with my goals from so far away in my homeland. My warmest gratitude goes to my sister and my brother, who gave me lots of confidence and supported me throughout my life and studies. Finally, it is my pleasure to greatfully thank Niousha Bolandzadeh. She gave me lots of motivation and energy and did a tremendous job of helping me in so many ways, especially during the last year of my PhD.

Table of Contents

1 Introduction **1**
1.1 Searching Beyond the Document Level 1
1.1.1 Document-level Search Systems 1
1.1.2 Question Answering Systems 2
1.1.3 Information Extraction Systems 2
1.2 Motivation 3
1.3 Problem 6
1.4 Contributions 7
1.5 Organization of the Thesis 8

2 Preliminaries and Background **10**
2.1 Sequential Model 10
2.1.1 Wild Card Queries 11
2.2 Structural Model 12
2.2.1 Data Model and Corpora 12
2.2.2 Query Model and Matching 13
2.2.3 Navigational Axes 15
2.2.4 Structural Indexes 15

3 Related Work **18**
3.1 Natural Language Text as Sequences of Words 19
3.1.1 Discussion of the Query Types 19
3.1.2 Supporting Wild Card Queries 20
3.2 Natural Language Text as Linguistically Annotated Trees 23
3.2.1 Querying over Linguistically Annotated Trees 23
3.2.2 Query Languages over XML Documents 27
3.2.3 Querying over Trees 32
3.2.4 Querying over Graphs 34
3.3 Natural Language Question Answering 36

4 Sequential Indexing and Querying of Natural Language Text **38**
4.1 Baseline Methods 38
4.1.1 Full Scan 38
4.1.2 Inverted Index 39
4.1.3 Neighbor Index 39
4.2 Permuterm Index over Natural Language Text 40
4.2.1 Word Level Burrows-Wheeler transformation 40
4.2.2 Maintaining the Alphabet 41
4.2.3 Rank Data Structures 42
4.2.4 Algorithms and Analysis 43

5 Structural Indexing and Querying of Natural Language Text **48**
5.1 Subtree Index 48
5.1.1 Subtree Indexes over Syntactically Parsed Trees 49
5.1.2 SI construction 52
5.1.3 Query Matching Over Subtree Indexes 54

5.1.4 Coding Schemes . . . 55
5.2 Query Splitting Strategies . . . 58
5.2.1 Monotonicity of Posting List Sizes . . . 58
5.2.2 Join Optimality . . . 60
5.3 Join Approaches over SI . . . 66
5.3.1 Joins . . . 67
5.3.2 Injective Matching . . . 67

6 Experimental Results 71
6.1 Natural Language Text as Sequences of Words . . . 71
6.1.1 Experimental Setup . . . 71
6.1.2 Performance of Querying . . . 72
6.1.3 WPI Performance with Limited Physical Memory . . . 74
6.1.4 Index Construction Time . . . 76
6.2 Natural Language Text as Syntactically Annotated Trees . . . 77
6.2.1 Experimental Setup . . . 77
6.2.2 Index Construction . . . 78
6.2.3 Querying Performance . . . 82

7 Conclusions and Future Directions 89
7.1 Summary and Discussion . . . 89
7.1.1 Word Permuterm Index . . . 89
7.1.2 Subtree Index and Root-split coding . . . 90
7.1.3 Discussion . . . 90
7.2 Future Directions . . . 91

Bibliography 93

A Supplementary Tables and Figures 100

B Proof of Lemmata and Theorems 107

C Supplementary Algorithms 110

List of Tables

2.1 Samples of natural language questions and their corresponding wild card queries . . . 12

3.1 A list of flat query types in the literature and their result sets . . . 20
3.2 Summary of the literature on query languages over syntactically annotated trees. Refer to the text for the meaning of abbreviations. . . . 25

4.1 Different wild card query patterns over WPI and their corresponding range of matches . . . 46
4.2 The running time complexity analysis of queries in Table 4.1 . . . 47

6.1 Summary of the performance of the indexes in terms of the running time in seconds . . . 72
6.2 Index construction time of WPI compared to the neighbor index in seconds 76
6.3 Ratio of the subtree index size when mss is 5 to the index size when mss is 1 . . . 80
6.4 Average running time of queries in seconds for queries in FB query set classes using Subtree index with root-split coding ($mss = 3$), ATreeGrep and Frequency-based approaches with varying frequency cutoff thresholds. 84
6.5 Total number of joins required over queries in the WH query set. r=root-split, s=subtree interval. . . . 85

A.1 The list of WH queries and their corresponding query structures . . . 102
A.2 List of *high*, *medium* and *low* frequency labels used in building FB queries. 103
A.3 Summary of the query runtimes for *who* parsed queries . . . 103
A.4 Summary of the query runtimes for *which* parsed queries . . . 103
A.5 Summary of the query runtimes for *where* parsed queries . . . 104
A.6 Summary of the query runtimes for *what* parsed queries . . . 104
A.7 Summary of the query runtimes for parsed queries over *low* frequency labels . . . 104
A.8 Summary of the query runtimes for parsed queries over *medium* frequency labels . . . 105
A.9 Summary of the query runtimes for parsed queries over *medium* and *low* frequency labels . . . 105
A.10 Summary of the query runtimes for parsed queries over *hight* frequency labels . . . 105
A.11 Summary of the query runtimes for parsed queries over *high* an *low* frequency labels . . . 106
A.12 Summary of the query runtimes for parsed queries over *high* and *medium* frequency labels . . . 106
A.13 Summary of the query runtimes for parsed queries over *high*, *medium* and *low* frequency labels . . . 106

List of Figures

1.1 (a) Parse tree of a sample question, (b) parse tree of a sample match. The bold labels and dashed edges indicate the match. Edge labels indicate role of the child relative to its parent and node labels include offset, word and POS tags. 4
1.2 Constituency Parse trees a sample query and a sample sentence containing a match. The bold labels and dashed edges indicate a match. Internal nodes indicate the syntactic role of its subtree and leaves indicate words in a sentence. 5

3.1 Architecture of an inverted index 21
3.2 Architecture of a neighbor index 22
3.3 Sample query, data tree and match results for a twig matching problem. . . 29

4.1 Sorted permutations of a sample set of sentences and the first and last word lists, F and L. 41
4.2 Rank function computes the occurrences of e in prefix $1 \dots i$ of L 43
4.3 A sample wavelet tree. In each node a bit string and two arrays, `super block rank` and `block rank`, are stored. 44
4.4 A constant-time $nodeRank$, returning binary rank at each node 45
4.5 backwardSearch algorithm for traversing L in backward order 45

5.1 Number of index keys (unique subtrees) as a function of the input size in terms of the number of sentences 50
5.2 Average number of subtrees extracted in terms of the branching factor of roots of subtrees 52
5.3 Algorithm for extracting all unique subtrees of sizes 1 to mss from a tree t 53
5.4 An example of how unique subtrees are extracted, (a) input tree, (b) unique subtrees of size 2, (c) unique subtrees of size 3 53
5.5 A naive algorithm that guarantees a root-split cover 58
5.6 Example of a query having deep branching anomaly for $mss = 4$ for a root-split join optimal cover 62
5.7 Join optimality on all possible queries of size 5 with mss values of $2, 3$ and 4 from left to right columns. 63
5.8 Algorithm that computes a join optimal cover of size mss 65
5.9 Algorithm that computes the best root-split cover of size mss 66
5.10 Example of a query and its corresponding cover that can lead to false positive matches 68
5.11 The algorithm that computes extra joins that guarantee injective matching . 70

6.1 The performance of the indexes based on the number of bindings of queries over 10 million sentences of news data 73
6.2 The performance of the indexes based on the number of bindings of queries over 1 million documents of web data 74
6.3 Scalability of the indexes over web data of growing sizes. 75
6.4 The performance of WPI vs. the neighbor index using paging on News Data of sizes 4, 6, 8, 10,12, 14, 16 and 18 million sentences 76

6.5 The performance of WPI vs. the neighbor index using paging on Web Data of sizes 0.4, 0.8, 1.2, 1.6 and 2 million documents 77
6.6 SI size for filter-based, root-split and subtree interval codings, with $mss = 1, \ldots, 5$. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences. 79
6.7 Total number of postings over all keys for filter-based, root-split and subtree interval codings, with varying input sizes and mss values. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences. 80
6.8 Total number of index keys for varying input sizes and mss values. (Left) absolute number of keys, (right) cumulative number of keys. 81
6.9 Index construction time for filter-based, root-split and subtree interval codings, with $mss = 1, \ldots, 5$. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences. 82
6.10 Average runtime of queries in terms of their number of matches for filter-based, root-split and subtree interval codings and mss values of 1 to 5 . . . 86
6.11 Average runtime of queries in terms of the size of queries for filter-based, root-split and subtree interval codings and mss values of 1 to 5 87
6.12 Average runtime of queries ($mss = 3$) over groups of FB queries over datasets of $1k$, $10k$, $100k$ and $1m$ sentences and using different coding schemes. 88

A.1 Sample natural language text sentences that contain nodes with high branching factors 100
A.2 Number of subtrees with sizes varying between 2 to 5 in terms of the branching factor of the nodes over which the subtrees are constructed . . . 101

C.1 Algorithm for extracting subtrees of size n rooted at t. 111
C.2 Algorithm that computes all the combinations of sizes of children that lead to a given subtree size 111

Chapter 1

Introduction

With natural language text evolved as an efficient medium for communication among people, it has also become a prominent form of representing information and knowledge. Huge volumes of natural language text are available electronically. Emails, news-group messages, web pages, research papers, books, news, etc. are almost entirely authored in human readable natural languages. Such data is also a major source of information for many applications including search engines, Question Answering (QA) systems, and analytical tools built on more ad-hoc basis.

1.1 Searching Beyond the Document Level

Natural language text is a major feed to a wide range of search applications. Many of these systems consider natural language text as an arbitrary bag or sequence of words (or characters). However, this view of text imposes a few limitations on searching; e.g. it ignores the regularities and inherent structure that exist in finer granularity portions of text such as clauses and sentences. In this section, we briefly review a few search systems that use natural language text and discuss their approach in modeling it.

1.1.1 Document-level Search Systems

Information Retrieval (IR) systems including search engines mostly provide access to a ranked list of documents, sorted based on their relevance to a keyword query. This document-level retrieval model has a few limitations. First, the unit of expressing factual information (e.g. named entities, facts and relations) is often smaller than a document. However, the storage model and access methods supported by IR engines are not optimized for answering text units smaller than a document. Harvesting fine granularity information from the documents will therefore require extra processing or manual work. Second, keyword-based queries used by IR engines, are expressed as flat sequences of words. This representation is not expressive enough to capture the relationships between the words or to filter the matches based on their part of speech tags (e.g. adjective or noun-phrase) or grammatical role (e.g. subject or verb). Finally, the term-document inverted indexing model used by IR systems, does not support queries that ask for syntactic relationships (e.g. parent-child or ancestor-descendant). These relationships can be obtained by processing natural language text sentences using a syntactic parser (See [77] for more on syntactic parsing).

1.1.2 Question Answering Systems

Question Answering (QA) systems leverage the syntactic and semantic properties of natural language text to find relevant facts and meaningful answers to questions. QA systems address the problem of finding answers to arbitrary questions such as `‘Who is the mayor of New York city?’` or `‘Which state is Chicago located in?’`. In order to answer the above questions, QA systems translate the questions into one or more queries. A common practice is to translate the question into a keyword query that is supported by a search engine and scan the returned documents for the answers.However, this approach can be inefficient when scaling for larger sets of documents and queries. Also, documents are returned based on their relevance to a keyword sequence representation which does not capture all the information that can be obtained from a question. As we discuss in Section 3.3, many question answering systems improve the accuracy of their answers by using augmented text such as POS-tagged or syntactically parsed text. Syntactic parsers for example are used for question typing, answer typing [66] and parsing snippets of text returned from posing keyword queries to a search engine.

1.1.3 Information Extraction Systems

Information Extraction (IE) systems focus on extracting fine granularity information such as entities and relationships mainly from sources on the Web. The idea here is to gather or tag these entities and relationships which can later be queried in a structured fashion or be integrated with structured data (e.g. in a database). Two examples of information extraction systems are KnowItAll [34] and TextRunner [8]; they both rely on learning patterns from the huge number of instances found over the Web, and are reported extracting a large numbers of facts in a short time. However, it is also reported that the quality of extraction can be improved greatly if the structural information of the text is taken into consideration. For instance, a body of recent work [25, 115] focuses on improving the precision and recall of IE systems by attending to the inherent structure underlying natural language text. They use the output of a syntactic parser or a semantic role labeler to learn the weight of features for their extractions and achieve up to 70% improvement in terms of the F-measure compared to TextRunner extractions. However, their extractions are slowed down by orders of magnitude.

The aforementioned systems suffer from one or more of the following. (1) The retrieval unit is large, making the search for finer granularity elements cumbersome and costly. (2) The inherent structure in natural language text is ignored, resulting in inaccurate extractions or limited search functionalities. (3) The structural information within sentences is taken into account, but a high price is paid for such information due to lack of efficient storage mechanisms and access methods. Our goal in this thesis is to address or alleviate those limitations. The following section provides a few examples that demonstrate how the structure in natural language text can be exploited to achieve more accurate answers.

1.2 Motivation

Example 1.2.1. Assume we are interested in finding the answer to the question, `‘Who is the mayor of New York city?’`. Without any knowledge about the correct answer, we can conclude that the answer is the name of a person, because we have a who question, and that the answer is in the class of mayor names. Thus, if we have access to a knowledge base that contains a list of mayor names we can effectively prune many of the potential matches that have been found.

The question discussed in Example 1.2.1 is a factoid question. It is relatively easier to find candidate matches for this class of questions compared to other types of questions such as *definition*, *reasoning* and *how-to* questions. The reason is that answers to factoid questions are in most cases short entity names that can be tagged by a Part Of Speech (POS) tagger as nouns or noun-phrases.

If we have access to a document-level search system that receives as input a keyword query and returns a relevance-ranked list of documents (as most web search engines do), we can convert the question to queries such as `Q1=‘mayor of New York city’` or `Q2=‘is the mayor of New-York city’` and evaluate them using the search engine. The answer is often found in the first few matching documents. However, even for factoid questions, it is not efficient to scan one or multiple documents per query and join the candidate matches with the knowledge base. The problem will be more severe if we want to automate the question answering process for a large number of questions. Therefore, more efficient access methods to potential matches could improve the performance of question answering.

Wild card queries [99] (as discussed in Section 2.1.1) may help writing those queries, but we would need efficient storage mechanisms and access methods that would more efficiently support wild card queries.

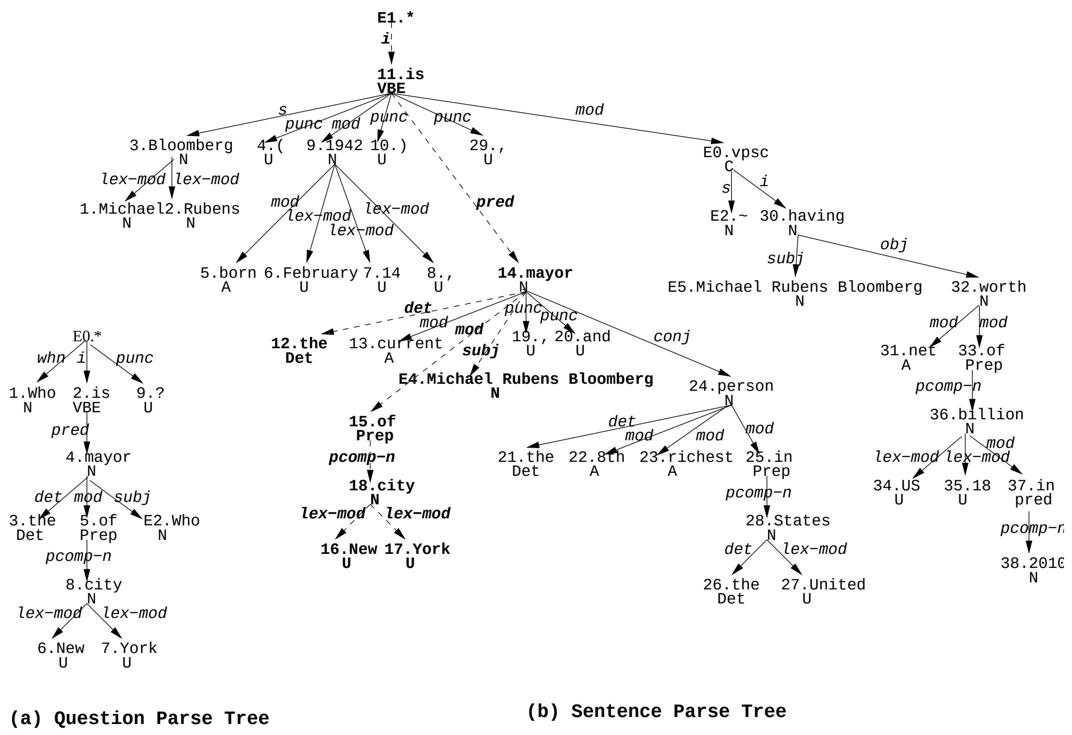


Figure 1.1: (a) Parse tree of a sample question, (b) parse tree of a sample match. The bold labels and dashed edges indicate the match. Edge labels indicate role of the child relative to its parent and node labels include offset, word and POS tags.

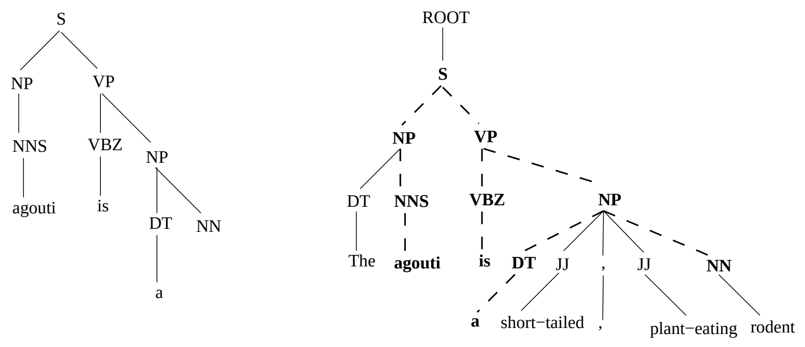
Example 1.2.2. If we pass the question in Example 1.2.1 to a dependency parser, such as Minipar [82], more information about the syntax of the answer can be found. Figures 1.1(a),(b) show the Minipar parse trees of the question, and the parse tree of a sample sentence that contains the answer[1], respectively. In these parse trees, the edges are labelled by the role of the child, relative to its parent and nodes are labelled using the word offset in the sentence, the word itself and its part of speech tag.

Looking at these two parse trees we conclude that although the answer, Michael Rubens

[1]The first sentence from the Wikipedia page of Michael Bloomberg on September 22, 2010. `''Michael Rubens Bloomberg (born February 14, 1942) is the current Mayor of New York City, and the 8th richest person in the United States, having net worth of US$18 billion in 2010''`

Bloomberg, and the identifying question terms, mayor of New York city, are far apart from each other in the original sentence, the parser manages to find their syntactic relationship correctly and associate them using edges in the tree. Moreover, the corresponding terms in the question and the sentence have the same POS tags, roles and syntactic relationships.

Example 1.2.3. In this example we use constituency parsing to answer a factoid question. Suppose we want to find the answer to the question `What kind of animal is agouti?`, chosen from the TREC-2004 question answering track [110]. Using a keyword-based search engine, we can send a query such as `agouti` and skim through the returned pages for the desired answer. Alternatively, if a corpus of syntactically parsed sentences that contain the answer exists, we can parse the text snippet `agouti is a kind of`, using Stanford Parser [62] and match against the database of parsed sentences. As Figure 1.2 shows, a `NN` node is added to the query parse tree to indicate that the sought word is a noun and thus achieving a more accurate result set. This snippet correctly matches the parse of the sentence `The agouti is a short-tailed, plant eating rodent`. The matched subtree has been marked with dashed lines for better visualization. As Figure 1.2 shows, there are a few words between the answer, `rodent`, and the words of the query, `agouti is a`, in the matched sentence. However, the parser correctly identifies the relationships between the corresponding words and a structural match can extract the answer.



(a) parse tree of a sample query (b) parse tree of a matching sentence

Figure 1.2: Constituency Parse trees a sample query and a sample sentence containing a match. The bold labels and dashed edges indicate a match. Internal nodes indicate the syntactic role of its subtree and leaves indicate words in a sentence.

Examples 1.2.2, 1.2.3 show how answers to natural language questions can be more effectively found using a parse tree. Syntactically annotated trees may also be used in for finding or verifying some linguistic phenomena in text, checking the accuracy of parsers or gathering some statistical evidence. More generally, we can state that searching over text can be significantly improved in terms of effectiveness and usability provided that we have an efficient and scalable access to parsed text. With the current state of the art parsers, it is possible to parse text fairly accurately, and since one can run them on a large number of machines at the same time, it is not far-fetched to think about the whole Web being parsed and indexed (see [71] for a web-scale part-of-speech annotated example).

Supporting the above applications boils down to searching for patterns and relationships over parse trees. There exist query systems that support a rich set of queries over syntactic trees (see Section 3.2.1 for a survey). While these systems work on clean and often small corpora, there has been little focus on the performance of these systems or any reports on their scalability over large text collections.

The given examples demonstrate some of the areas where more expressive and direct queries can improve the effectiveness of searching over natural language text. In this thesis we study the problem of efficiently supporting queries over natural language text. We focus on the two scenarios where natural language text is either modelled as sequences of words or as syntactic trees containing word labels and relationships. Next section discusses more details of the problems we address, the challenges involved and the areas we cover.

1.3 Problem

With all the benefits of searching over linguistic relationships accessible in a syntactic parse tree or other forms of fine-grained text, there are also some challenges, and an important one is the efficiency of the searches; this is also the challenge we take on in this thesis. In particular, we study the storage structures and access methods under two scenarios: (1) when a sentence is modelled as a flat sequence of words and (2) when a sentence is modelled as a syntactically annotated tree.

In the scenario where sentences are modelled as flat sequences of words, we look at the problem of efficiently supporting wild card queries over large-scale text. The syntax and semantics of wild card queries are defined in more detail in Section 2.1.1. We study index structures that directly support word-level matchings required for answering wild card queries. Moreover, we study the performance gain over the scenario where a match is a document and wild card matching requires post-processing. The querying algorithms and access methods are developed with regard to the underlying index structure and with respect to the extraction needs. The challenges involved include minimizing the response time of queries while keeping the index size and index construction time at a reasonable level.

In the second scenario where sentences are parsed and modelled as syntactic trees, we study the problem of answering a set of unordered, node-labelled tree queries. The syntax of such queries and the semantics of matchings are defined in Section 2.2.2. In order to improve the efficiency of query matching, we propose a novel index structure over the set of unique subtrees of the input parse tree corpus. A large number of challenges are involved with regards to query evaluation and storage schemes over the proposed subtree index. We study the problem of query splitting both theoretically and experimentally and investigate existence of optimal query splits, in terms of the number of joins required for evaluating the query. Another challenge in developing a subtree index is often the enormous index size. We study ways of storing structural subtree information concisely and the effect of different coding schemes on the index size, index construction time and the number of joins involved for query evaluation.

1.4 Contributions

In this thesis we make the following contributions.

Contribution 1.4.1. We introduce Word Permuterm Index (WPI) [26] as an efficient method for evaluating wild card queries over natural language text. WPI extends Permuterm Index (PI) [41] in several aspects. (1) By construction, WPI supports pattern matching over keywords rather than characters, (2) WPI supports a wider range of queries than PI, adding support for queries that are more frequently used over natural language text, and (3) WPI returns the actual keywords that match a wild card query whereas PI is mostly used to find the range of elements that match a pattern. Thus, WPI goes one more step toward matching the keywords after finding the range of matching elements.

Contribution 1.4.2. We propose a Subtree Index (SI) over syntactically parsed corpora of natural language text. SI stores all unique subtrees (up to a certain size) from the corpora as index keys. It also stores the structural information of each subtree in a set of posting lists which can be used to perform exact tree query matchings. We show theoretically and experimentally that SI can achieve a large query run-time speedup compared to the scenario where only structural information of nodes are stored (see [9] for an example). To the best of our knowledge, SI is the first work on indexing tree structured data that stores the set of unique subtrees as index keys.

Contribution 1.4.3. A novel root-split coding scheme and corresponding query splitting and evaluation algorithms are proposed. We discuss that root-split coding concisely stores the structural information of subtrees within SI, making it possible to perform exact matching, while reducing the index size, index construction time and query response times. Root-split coding and our baseline coding schemes are discussed in Chapter 5.

Contribution 1.4.4. We develop efficient query splitting algorithms over root-split and our baseline methods, subtree interval and filter-based codings, in the scenario where query matching is not injective (See Section 2.2.2 for a discussion of injective matching). Over subtree interval and filter-based codings, we propose a query splitting algorithm that achieves optimality in terms of the number of joins required to evaluate the query. Over root-split coding, we present an efficient query splitting algorithm that decomposes the query into smallest number of subtrees possible for a root-split query evaluation.

Contribution 1.4.5. In the scenario where query matching over SI is required to be injective, we propose novel pruning techniques to achieve efficient query splitting and query evaluation algorithms, reducing the overall number of joins required compared a naive approach.

Contribution 1.4.6. The last contribution of this thesis is the broad set of experiments and analyses. (1) We compare the performance of WPI to alternative baseline methods. Our performance comparison includes cases where WPI is given a limited memory and is forced to do paging. To the best of our knowledge this is the first work that experimentally compares traditional inverted file indexes with more recent succinct full-text self indexes (See the survey in [90]). (2) We experimentally show that SI storing subtrees larger than one node can outperform the node approach. Moreover, we show that our root-split coding outperforms our baseline coding schemes in terms of the query response times, is highly scalable and has a reasonably small index size and index construction time.

1.5 Organization of the Thesis

The organization of this thesis is as follows. In the next chapter, we present some background information required for understanding the discussions in the rest of thesis, specifically the material presented in Chapters 4 and 5. Next chapter also defines more formally the preliminary concepts used of this thesis such as the syntax and semantics of the queries we support, our data models and the metrics we use to evaluate our algorithms and solutions.

In Chapter 3, a comprehensive review of the literature around querying and indexing natural language text is presented. This chapter includes related work on topics such as query systems over syntactically annotated corpora, question answering over natural language text, and querying over XML documents, to name a few. We further discuss how these systems and solutions relate to our work and discuss the advantages and drawbacks of each approach.

Chapter 4 discusses the problems associated with querying natural language text in its sequential representation. We propose Word Permuterm Index (WPI), and study its architecture and data structures. We also present a set of query processing algorithms over WPI and asymptotically analyze the time complexity of such algorithms.

In Chapter 5 we propose a novel Subtree Index (SI) structure that improves the performance of querying over syntactically annotated trees at the price of a larger index size. We theoretically analyze the properties of SI, and propose baseline algorithms for splitting and evaluating queries over SI. We also introduce root-split coding and its corresponding query evaluation algorithms and discuss some of the benefits and limitations.

In Chapter 6 we present a broad range of experiments to verify the effectiveness of our index structures and querying algorithms. We use different hardware and software settings and parameters in order to check the robustness of our algorithms in different scenarios. We also perform several experiments to show the scalability of our methods to large text corpora.

Finally, in Chapter 7, we conclude with a discussion of the achievements and the limitations of the thesis. Moreover, we present a set of ideas and future avenues that can lead to interesting new problems or that potentially could improve the results presented in this thesis.

Chapter 2

Preliminaries and Background

This chapter provides a more formal definition of some of the concepts, terminology and notations relevant to the scope of the problem that we study in this thesis. Since we consider two different settings where text is modelled as (1) flat sequences of words or (2) syntactically annotated trees, our discussion in this section also treats them separately.

2.1 Sequential Model

Under a sequential model, a text search often refers to finding pieces in text that match a sequential pattern. More formally, consider a text collection containing a sequence of elements, $C =< e_1, \cdots, e_n >$, where each element is a sequence of terms[1] taken from an alphabet Σ; i.e. $e_i =< t_{i1} \cdots t_{ik} >$, where $t_{ij} \in \Sigma$. The following definitions more precisely describe text pattern matching over natural language text.

Definition 2.1.1. (Match) A match of a query $Q =< q_1 \cdots q_m >$ is an element $e =< t_1 \cdots t_p >$, where for every q_i in Q, there is a t_j in e, such that $q_i \sim t_j$, meaning that q_i and t_j are the same alphabet tokens.

More intuitively, the above definition requires every match (a document, sentence, paragraph, etc.) to contain all query elements (usually keywords). In other words, all the query terms have to appear in every match at least once.

Definition 2.1.2. (Order-Preserving Match) An order-preserving match of a query $Q =< q_1 \cdots q_m >$ is a match $e =< t_1 \cdots t_p >$, where for every $q_i \sim t_j$ and $q_k \sim t_l$ if $i < k$ then $j < l$.

[1]words, phrases and perhaps punctuation

Definition 2.1.3. (Phrasal Match) A phrasal match of a query $Q =< q_1 \cdots q_m >$ is an order-preserving match $e =< t_1 \cdots t_p >$, where for every $q_i \sim t_j$ and $q_k \sim t_l$ if $k = i + 1$ then $l = j + 1$. In this setting, Q is referred to as a **phrasal query**.

Definition 2.1.1 treats text as a bag of words whereas Definitions 2.1.2 and 2.1.3 treat it as a sequence of words and add some constraints on order and adjacency of terms within a match. Next we describe a class of queries, called wild card queries, that further expands word-level matchings.

2.1.1 Wild Card Queries

In contrast with the traditional (usually information retrieval based) definition of a match, where a keyword query matches whole elements of the collection, wild card queries contain place-holders, called wild cards, which match single text pieces within collection elements. A more formal definition of a wild card query follows.

Definition 2.1.4. (Wild Card Query) A wild card query $Q =< q_1 \cdots q_m >$, is a phrasal query, where each $q_i \in \Sigma \cup \{\%\}$, and $\%$ is an extractor wild card. A match for a wild card query is a tuple $e_i =< t_{i_1} \cdots t_{i_m} >$ of terms such that there is an assignment of terms in e_i to wild cards $\%$ in Q such that the assignment would make e_i a phrasal match for Q.

Wild card queries supporting word-level extractions are particularly important for several reasons. First, a large class of natural language questions, known as factoid questions, can intuitively be translated into one or more wild card queries (See Table 2.1 for a set of examples). Second, the results of such queries can easily be joined with data that may reside in a database. For example, candidate answers for the *which* question in Table 2.1 can be further refined by looking up the values returned by the first query in a database populated with a list of city names (second query). Finally, question answering systems often rely on NLP components that may directly or indirectly use wild card queries. Examples are taxonomy construction, fact extraction, named entity recognition and query expansion.

Information extraction can also be enriched using wild card queries. Rafiei and Li [99] present a data extraction system using wild card queries over web text. They discuss several techniques such as query expansion and relevance ranking to increase the precision and recall of extractions.

Table 2.1: Samples of natural language questions and their corresponding wild card queries

Question	**Translation**
Who invented the light bulb?	% invented the light bulb
What is glass made of?	glass is made of %
Which city hosted 1988 Olympics?	% hosted 1988 Olympics % is a city
Where is Grand Canyon located?	Grand Canyon is located in %
How tall is the Empire State Building?	Empire State Building is % tall
How many electrons are there in a sodium atom?	There are % electrons in a sodium atom

2.2 Structural Model

In this scenario, natural language text is modelled as collections of syntactically annotated trees. Such collections can be generated automatically (using syntactic parsers), manually or semi-automatically. Regardless of the approach used for generating such corpora of syntactic trees, the text here is often modelled as collections of unranked node-labelled trees. Such trees are unranked, meaning that every node can have any arbitrary number of children.

In this section, we briefly study different types of corpora and how we model them for our querying task. Further we describe our query and matching models, and touch on relevant background on binary axes matching, interval coding schemes and structural joins.

2.2.1 Data Model and Corpora

Linguistic corpora can exist in several different form; e.g. there exist text and speech corpora. Text corpora are often represented as hierarchical structures, and can exist as constituency-based or dependency-based, each with a different set of tags, semantics of relationships and annotation levels. In the following, we describe in more detail, some of the variations in linguistic corpora.

Annotation Grammar

Two main classes of syntactically annotated text corpora (also known as treebanks) can be identified, based on the grammar that is used for parsing and annotating the text; these are the dependency-based and constituency-based syntactically annotated corpora. Dependency-based treebanks (e.g. Prague Dependency Treebank [85]), mark the relationships between individual words in a sentence with their dependents. Constituency-Based treebanks (e.g. Penn Treebank of English text [78]) are often obtained by constituency parsers and mark the nested structure of constituents within a sentence, such as verbs and noun phrases. As such, there are constituents that contain other constituents. For example, a verb phrase containing a verb and perhaps a noun.

Annotation Levels

Annotations can happen at many different levels. Depending on the type of grammar used, constituency or dependency, the annotation levels might be different. Dependency-based corpora are concerned with the dependencies between words in a sentence. Some of the supported annotations are as follows. The *morphological* level stores information about morphemes. The result of a morphological tagging is a flat structure with annotations on individual words in a sentence. The analytical or syntactic level has the structural relationships between dependents in a sentence and their governors. The *analytical* annotation generates a single-rooted tree structure, where the order of siblings can be defined to be the order of their words in the sentence. The *tectogrammatical* or semantic level is a more complex annotation, marking the semantic relatedness of the words within a sentence. Similar to the analytical level, the tectogrammatical level generates a tree structure. However, the mapping between the semantic level nodes and the analytical level nodes is not necessarily one-to-one. For an example of these annotation levels see [84].

The annotations developed over a constituency grammar mark the constituents of the sentence and do not deal with the dependencies within the sentences. Some of the annotation levels available are lexical, topological, phrasal and clausal (e.g. [55]). The *lexical* level almost corresponds to the morphological level of dependency corpora, marking the individual words with their lexical information. The *topological* annotations are descriptive annotations about the constituents in a word, such as affixes. The *phrasal* annotations, identify the phrase structures in a sentence, such as noun-phrases or propositional-phrases. Finally, *clausal* annotations mark the boundaries of clauses.

2.2.2 Query Model and Matching

As discussed in Section 2.2.1, syntactic trees are modelled using unranked node-labelled trees. At the abstract level, queries over syntactic trees are represented using trees whose nodes represent the annotations to be matched, and the edges represent the binary relationships that govern between the corresponding annotations. We call such queries Syntactically Annotated Tree Queries or SAT-Q. Two scenarios are possible for generating queries over syntactic trees. First is the scenario where the queries are generated from natural language questions (especially factoid questions) using the same method for generating syntactic trees. This approach is perhaps the most favorable for the user, as natural language questions are an easy and intuitive way to express the information need for humans. In this approach queries can be represented using unranked node-labelled trees, with nodes marking the labels to be matched and edges denoting a parent-child relationship. Second is the scenario where query trees are generated by an expert user, familiar with the grammar of the language, syntax of trees in the treebank and the range and semantics of annotations. Such queries vary greatly in terms of their expressive power, syntax and even semantics across different querying systems over syntactically annotated trees (See Section 3.2.1 for a brief survey on such systems). However, a majority of queries in this category can also be represented using unordered unranked trees whose edge could denote a wider range of navigational axes. The focus in this thesis is mostly on the first type of queries, while a subset of the second category can also be supported.

Query matching happens by mapping query nodes to syntactic tree nodes that have the same label. A formal definition of a match over syntactically annotated trees follows.

Definition 2.2.1. (SAT-Q Matching) Given a query Q and a tree T, a matching is a mapping function $f : V(Q) \mapsto V(T)$ that maps nodes of Q to nodes of V, such that (1) for every query node $v \in V(Q)$ we have $label(v) = label(f(v))$ and (2) for each query edge uv connecting nodes u and v of Q, then $f(u)$ and $f(v)$ have the same *relationship* in T as marked by uv.

In the above definition, $label(v)$ denotes the annotation or label used on node v. We call T a match of Q, if and only if there exists a matching from Q to T. The relationships over query edges are often represented using a set of navigational axes. The syntax and semantics of these axes will be covered in Section 2.2.3.

Query nodes might have the same labels. In such a case, it is reasonable to assume that nodes with the same label are mapped to distinct nodes of the data tree. Without such an assumption, a few common queries cannot be expressed, where two or more nodes with the same label occur in the same relationship; e.g. (1) two adjectives modifying a single noun or (2) two noun phrases such as subject and object as children of a verb phrase. The assumption on the distinctness of the matching labels over the data tree requires the mapping function to be injective. In general, an injective matching is more costly than the non-injective counter-part. We call such a matching, an injective matching and define it as follows.

Definition 2.2.2. (SAT-Q Injective Match) Given a query Q and a syntactically annotated tree T, an injective match is a match whose mapping function $f : V(Q) \mapsto V(T)$ is injective. I.e. for all $u, v \in V(Q)$ we have $f(u) = f(v)$ if and only if $u = v$.

2.2.3 Navigational Axes

Navigational axes are binary structural relationships between pairs of nodes in a query tree. These axes have widely been used in query matching over tree structured data such as eXtensible Markup Language (XML) documents and syntactically annotated trees.

The full set of navigational axes, expressed with respect to a context node are as follows. (1) **basic axes** are single forward location steps including child, immediate-following and immediate-following-sibling. (2) **reverse axes** are single backward location steps including parent, immediate-preceding and immediate-preceding-sibling, (3) **transitive closure axes** are multiple forward location steps including descendant, following and following-sibling. Finally, (4) **reverse transitive closure axes** are multiple backward location steps which include ancestor, preceding and preceding-sibling.

Based on their querying needs, different systems might support parts or all of the above axes. XPath [27] for example, which is a path query language over XML documents supports all but the ones prefixed with *immediate* word. LPath [9] which is an adaptation of XPath over syntactically annotated trees, supports all navigational axes. Finally twig queries, support only the parent-child and ancestor-descendant axes (See [47] for a survey).

2.2.4 Structural Indexes

Over large corpora, scanning individual trees for finding potential matches is not efficient. As a result indexes are commonly used over tree structured data that store structural information of the nodes. The structural information of nodes are usually represented by a set of numbers which uniquely identify a node within its hierarchical structure. These numbering schemes have widely been used over XML documents. In this section, we briefly review a few of such numbering schemes and a set of structural join approaches that utilize the discussed numbering schemes to provide efficient access method over tree structured data.

Numbering Schemes

Assume we are given a query $A//B$ (A is an ancestor of B) and we would like to find the matches for A and B in a data tree. A naive approach to evaluate the query is to start from every occurrence of A in the data tree and traverse its entire subtree to check if there are any descendents labelled as B. In order to avoid such costly traversals, a set of numbering schemes have been developed to support $/$ (parent-child) and $//$ (ancestor-descendant) axes, efficiently. Using these numbering schemes, any pair of nodes over a data tree can be checked in constant-time to find out if they match the given axis or not. The most commonly used numbering schemes are *interval coding* and *Dewey coding* [16]. Numbering schemes have been first proposed by Dietz [32] over trees and have been used extensively for querying XML trees.

Interval coding [120] assigns each node a pair of *pre* and *post* numbers. These numbers indicate the interval spanned by the subtree rooted at the given node and correspond to the pre-visit and post-visit numbers assigned to nodes while traversing the tree in any depth-first search (DFS) traversal. As an example, the (pre, post) numbers for the nodes in Figure 3.3(b) are $a_1 : (1, 7)$, $b_1 : (2, 5)$, $b_2 : (3, 3)$, $c_1 : (4, 4)$ and $c_2 : (6, 6)$. The (pre, post) numbers are also referred to as (left, right) and (begin, end) in different querying systems. Axes evaluation using the PrePost coding is performed as follows. If node a is an ancestor of node b, then $a//b \Leftrightarrow a.pre < b.pre < a.post$. The parent-child axis can be checked by adding the level (depth) number to the interval coding. Thus, $a/b \Leftrightarrow a.pre < b.pre < a.post$ and $a.level + 1 = b.level$. By adding a parent id, the interval coding can be extended to support all other axes supported by XPath [47].

Interval coding is very efficient for answering containment queries. Moreover, it is very space efficient. However, with interval coding, updates in the data tree will be very expensive, requiring a large number of node numbers to be updated. Dewey Coding, first introduced for coding XML by Tatarinov et al. [106], can help reduce the cost of updates. In Dewey coding, each node label is prefixed by the label of its parent and the siblings have labels in the same order as they would appear in a depth-first traversal. As an example, the labels for the nodes in Figure 3.3(b) are $a_1 : 1$, $b_1 : 1.1$, $b_2 : 1.1.1$, $c_1 : 1.1.2$ and $c_2 : 1.2$. Thus, any containment query will be converted to a prefix matching of the corresponding node labels.

For the subject area of this thesis, the interval coding is more relevant as we are mostly dealing with static data that almost never changes once stored and indexed. In the interval coding, each interval is denoted by two values, *left* and *right* and each axis match is checked by evaluating a set of inequalities on both numbers. Thus, the common practice is to sort the data on the *left* values and use structural joins for efficiently matching queries. In the next Section, we discuss some specialized join techniques that are developed for joining the structural information.

Structural Joins

The coding schemes introduced above capture the nested structure of trees when indexing tree structured data, such as XML documents and syntactically annotated trees. A large body of work on XML query processing consider the scenario where each element of the document is represented using a document id, and the numbers *left*, *right* and *level*, with the elements sorted on the document id and the *left* values (See [47] for a survey). In particular, the interval coding has been utilized in the Multi-predicate merge join (MPMGJN) of Zhang et al. [120], and this join method is shown to outperform regular database join techniques. StackTree [1] introduces stack-based processing of structural joins, avoiding many of the extra axis checks especially those for parent-child axis. A drawback of both these approaches is that query trees have to be decomposed into individual binary axis matches and the partial results need to be joined to form the final output. Since the size of intermediate results could grow very large, thus reducing the performance, a large body of work in this area has been devoted to holistic twig query processing. More on these topics will be discussed in Section 3.2.2.

Chapter 3

Related Work

In this chapter, we review the literature around indexing and querying over natural language text. Similar to the structure of the rest of the thesis, this chapter is divided mainly into two distinct topics. We differentiate between approaches that consider text as flat sequences of words with the ones that take into consideration the syntactic structure of natural language text.

The first set of works include works around indexing and querying text in the scenario where text is modelled as sequences of words. We review a few indexing techniques including those on handling wild card queries. Self-indexes are also reviewed as they succinctly store text and are amenable for text compression.

The second set of approaches on indexing and querying natural language text benefits from the grammatical structure of text obtained through syntactic parsing. Such approaches introduce indexing and querying data that is represented by unranked node-labelled trees. We study a body of work on querying systems over natural language text, mainly from the linguistics and NLP literatures. Such systems mostly deal with an expressive set of queries on a small, clean and often hand-tagged corpora. Performance of querying over syntactically annotated corpora has rarely been addressed in such systems. Further, we study indexing and querying systems over tree structured data such as XML documents mainly from a database perspective. In this latter set of works, the focus is on improving the performance. In each case we compare our problem and algorithms with the works that are reviewed.

Finally, we study natural language question answering as an NLP application which has received considerable attention in the past decades. We discuss how QA over relatively smaller corpora has benefited from syntactic parsing, and how this is desirable for larger text collections.

3.1 Natural Language Text as Sequences of Words

Natural language text data exists in large volumes in electronic format as long sequences of words, delimiters and punctuations. The common approach in the literature for storing text has been to build inverted lists of (non-stop) words that include the indexed elements, occurrence frequencies, offsets, etc. In this section we review some of the query types over flat text and discuss the advantages of wild card queries for word-level extractions over natural language text. We further study some of the prominent works on indexing and querying natural language text with more focus on indexing techniques that support wild card queries.

3.1.1 Discussion of the Query Types

Question answering on a large corpus is a challenging task mainly because it is difficult to analyze the whole (or a good portion of) data and to retrieve candidate answers. On open-domain QA applications, such as question answering over the web (See OpenEphyra [91] for an example), it would be very difficult to build a general QA system with high accuracy. In those cases, a reasonable approach is to convert natural language questions into queries and benefit from available querying engines to enhance the performance of the search. The choice of queries can further affect the efficiency of searching, ease of translating questions to queries and the relevance of the results.

Table 3.1 gives a list of a few query types and some of the contexts where each query type is used. Although our focus in this thesis is on wild card queries, other types of queries or a combination of them might be interesting in question answering or other linguistic applications.

From the list of queries in Table 3.1, this thesis mostly focuses on the queries presented in rows two and four. Multi-keyword queries can improve the performance of keyword-based querying by pre-materializing a few joins and reducing the size of the posting lists. They are interesting because wild card queries can benefit from a multi-keyword index structure, as each key may contain neighbor terms and wild card queries might be answered without referring to the original data.

Note that by wild card queries, we specifically mean word level wild card queries. Full-text search supports wild cards in the form of regular expressions, but they are not the focus of this thesis. For instance, Lucene [74] supports $*$ and $?$ wild cards, which match *'zero or more'* and *'zero or one'* characters, respectively. However, it does not differentiate between

Table 3.1: A list of flat query types in the literature and their result sets

Query Type	**Query Elements**	**Result Set**	**References**
Keyword queries	keywords, Boolean operators	documents	AltaVista [2], Google [44], Yahoo [116]
Multi-keyword queries	keywords, phrases	documents	[18], Nextword and Phrase index [7, 114]
Proximity queries	keywords, proximity radius	documents	Lucene [74], INDRI [57], [17]
Wild card queries	queries, wild cards(%)	list of keywords	Dewild [99], BE [13], KnowItAll [34]
Structured queries	SQL, text predicates	relations	Oracle InterMedia Text[58], DB2 Text Extender[75], [14]
Full-text search	characters, regular expressions	strings	Lucene [74]

word level matches and matches within words. Moreover, Lucene has to scan the entire data collections for the prefix wild card queries such as `'*oors'`, which can potentially match `'doors'` and `'floors'`. For this reason, earlier versions of Lucene did not support prefix wild card matchings. In Section 3.1.2 we study a permuterm index which supports a class of full-text search queries having wild cards and forms a baseline for our Word Permuterm Index discussed in Chapter 4.

3.1.2 Supporting Wild Card Queries

Wild card queries, defined in Section 2.1.1, are fill-in-the-blank queries that are appropriate for word level extractions. Supporting them efficiently brings some interesting challenges. In this section, we describe how state of the art indexing schemes support wild card queries.

Inverted Index

There has been a great deal of activity around increasing the efficiency of keyword-based queries. However, the same structures and algorithms would not necessarily be useful or efficient for evaluating wild card queries. Assume we are given an inverted index structure, such as the one depicted in Figure 3.1 with four terms and three documents. Each term t in the index has a list of postings, each posting in the form of a triplet $< d, f_{t,d}, [o_1, \cdots, o_{f_{t,d}}] >$ where d is a document id, $f_{t,d}$ is the frequency of t in d and $o_1 \cdots o_{f_{t,d}}$ are offsets in d where t appears.

Given a keyword query `Q1:'world population'` and a wild card query `Q2:'world population is %'`, the algorithm for evaluating `Q1` involves only intersecting the posting lists of terms `'world'` and `'population'`, and finding the list of matching documents. However, for `Q2`, each matching document has to further be scanned in order to find the keywords that match the wild card. In the above example, `Q1` matches <1,2,[37,56]>, <2,1,[124]> and <3,1,[7]>, and `Q2` matches `'6706993152'` which is located on offset 10 of document 3. Although `Q2` matches its answer in fewer documents than `Q1`, the query response time for `Q2` using inverted indexes in one of our experiments was 12 times larger. This indicates that inverted indexes are not appropriate for evaluating wild card queries.

6706993152	→<3,1,[10]>
is	→<1,4,[12,154,184,190]>, <2,4,[379,401,427,503]>, <3,1,[9]>
population	→<1,7,[8,30,38,57,153,170,194]>, <2,2,[125,155]>, <3,1,[8]>
world	→<1,3,[11,37,56]>, <2,2,[29,124]>, <3,1,[7]>

Figure 3.1: Architecture of an inverted index

Solutions on multi-keyword queries such as phrase and nextword indexes [7, 114] can help reduce the time it takes to intersect the posting lists, but won't help in the keyword matching step, which is in most cases the dominant process. Therefore, development of solutions for efficient retrieval of keyword matches from text seems essential.

Neighbor Index

Neighbor index, as proposed by Cafarella and Etzioni [13], is an inverted index that is more suitable for queries over natural language text data. The index stores for each term both its left and right neighbors. As shown in Figure 3.2 for our running example (given in Figure 3.1), the inverted lists have grown significantly larger, but the answers to wild card matches are stored within the index and can be found by looking at the appropriate neighbors of a query literal. For example, to find the matches for `Q2` in the neighbor index, the search is conducted in the inverted index until offset $o = 10$ in document $d = 1$ is identified as an answer. To obtain the actual answer, it is sufficient to look at the right neighbor of the term at offset 9 in the index without retrieving the document. This can speed up the evaluation of wild card queries by 1-2 orders of magnitude compared to inverted index, as reported by the authors and confirmed in some of our experiments in Chapter 6.

Permuterm Index and Self-indexes

Recently, there has been an evolving trend in developing index structures supporting fast full-text searches over large text corpora. These systems study the theoretical and practical

6706993152 → <3,1,[(10,is,<DBM[1]>)]>
is → <1,4,[(12,world,estimated),(154,population,expected),(184,Earth,experiencing),(190,consensus,that)]>,
<2,4,[(379,sector,equally),(401,there,a),(427,there,a),(503,action,not)]>,
<3,1,[(9,population,6706993152)]>
population → <1,7,[(8,human,of),(30,human,to),(38,world,has),(57,world,growth),(153,world's,is),(170,human,over),
(194,current,expansion)]>, <2,2,[(125,world,and),(155,a,set)]>, <3,1,[(8,world,is)]>
world → <1,3,[(11,the,is),(37,The,population),(56,of,population)]>
, <2,2,[(29,and,population),(124,fastgrowing,population)]>, <3,1,[(7,the,population)]>

[1] Document Boundary Marker

Figure 3.2: Architecture of a neighbor index

aspects of index succinctness, search efficiency and compression. As a result, succinct indexes have been developed and many interesting problems associated with them have been studied [90]. A succinct index, is an index that is able to store text in size proportional to the information-theoretic lower bound of the text, while maintaining search efficiency.

One of the key ideas that led to the development of such indexes have been the idea of the Permuterm Index by Garfield [41]. A permuterm index, computes all the sorted cyclic rotations of the text, reducing the pattern matching into prefix searches. As a result, permuterm index requires $O(N^2)$ space, where N is the size of the dataset, which is prohibitive. Motivated by the idea of a permuterm index, Burrows-Wheeler transformation (BWT) [12], achieves a text transformation that is more amenable for compression. BWT, discussed in Section 4.2.1, is thus used in building succinct indexes. Ferragina and Venturini [39] use BWT to build a compressed permuterm index (CPI) that supports wild card queries. CPI benefits from a rich set of previous work on efficient pattern matching over BWT and strings.

Despite being highly efficient, CPI supports only a limited number of wild card queries and is tuned to answer full-text searches over strings rather than natural language text queries. A word-level adaption of the permuterm index has been developed by Chubak and Rafiei in [26] which improves upon CPI in a few directions. Compared to CPI, WPI supports a wider range of wild card queries. Moreover WPI supports word-level extractions and variable size alphabets. WPI is covered in detail in chapter 4.

Other related work to WPI are the keyword-based generalizations of text index structures such as word suffix arrays [37] and word suffix trees [4]. Finally, Manning et al. [76] propose solutions for extending CPI to support more than one wild cards. They propose materializing the range of matching rotations for one substring of query literals and intersecting with the results obtained from the prefix range returned by the rest of the query literals.

Other Sequential Text Indexes

Querying over natural language text is often addressed in the literature by indexes that are based on inverted lists. For large text corpora, these indexes run into the problem of high costs of intersecting long posting lists. As a result, solutions for multiple keywords have been proposed that materialize posting lists for more than one keyword. Examples are the works on phrase index and nextword index [7, 114]. Phrase index extracts natural language phrases from a query log and stores inverted lists for such phrases. A nextword index, for each term, keeps a list of high frequency terms that follow it in the text and the pair's corresponding inverted list. Chaudhuri et al. [18] propose breaking long posting lists into smaller ones by storing lists for multiple keywords. As a result they can guarantee an upper bound for the worst case running time of the queries. However, the above works have no support for wild card queries.

3.2 Natural Language Text as Linguistically Annotated Trees

3.2.1 Querying over Linguistically Annotated Trees

There has been numerous systems developed in the past decade for querying over linguistically annotated trees. These systems differ widely in terms of the expressiveness of the queries they support, the types of corpora they address (e.g. treebank, parallel, time or word-aligned), performance, architecture and last but not least the syntax of their queries. Some of these differences for a handful of published systems have been studied (e.g. [67]). In this section we present a more comprehensive study of the systems for querying linguistically annotated corpora. Moreover, we classify and compare such querying systems in terms of their performance.

Tgrep [97], or tree grep, is one of the earliest systems developed for querying syntactically annotated trees. Tgrep and its successor Tgrep2 [101] support sequential and hierarchical querying over parse trees in a treebank. It operates over its own corpus file format, but sentences parsed in penn-treebank [98] style or Combinatory Categorial Grammar (CCG) style can also be converted to Tgrep format and queried. Tgrep2 has support for the full set of axes defined in Section 2.2.3, and boolean operators. However, it does not support quantifiers and thus tgrep2 is not First Order (FO) complete. The syntax of its language is simple and can easily be used by non-expert users. Unlike our approach on querying syntactic trees, Tgrep2 does not benefit from any index for efficient access to the parsed data. For any query, it scans the whole corpus; hence, as reported in [42], it does not scale on large corpora.

LPath [9] is a more recent system which extends the axes supported by XPath [27] to a wider range of linguistic queries. It is at least as expressive as Tgrep2 and is more expressive than XPath [69]. LPath uses a numbering scheme similar to the PrePost Coding [120] of XML for efficient access to the navigational axes. Moreover, it uses a relational database to store structural information about nodes in parse trees and creates several indexes for efficient access. Performance analysis in [9] shows that LPath outperforms tgrep2 and CorpusSearch [100] in most cases and has comparable efficiency to XPath when developed under an engine using the labeling scheme proposed in [31]. In contrast, our system stores the structural information of subtrees rather than nodes. In the case where the size of subtrees stored is at most 1, our system is very similar to LPath. Moreover, our focus in this thesis has been both on improving the performance of querying and on the conciseness of the index.

Other querying systems over syntactically annotated corpora either focus on increasing the expressiveness of the queries or focus on particular types of treebanks. Kepser developed Finite Structure Query (FSQ) [60] which is a lisp-like language for querying syntactically annotated trees. Despite this query language being very expressive (full first order logic) supporting extensive use of quantification, it is very inefficient and has a difficult syntax. MONASearch, a later work by Kepser et al. [61, 79], adds Monadic Second Order (MSO) elements to the querying language, adding even more to the expressiveness. Some queries such as counting are only expressible in MSO. MONASearch uses tree automata for its matchings, which results in an inefficient linear-time scan of the whole corpus for every query. The reported performance of MonaSearch in [79] shows that it outperforms FSQ, but has a worse query time than TIGERSearch [64] on simple queries. TIGERSearch is a tool for querying different treebanks, originally developed for querying a German newspaper treebank.

Table 3.2 gives a comprehensive summary of different querying systems over syntactically annotated trees, sorted based on the year these systems have been developed. In this table, we compare some of the major query languages in terms of their expressiveness, querying approach, target corpus and architecture. We also indicate if there has been any report of the performance over these query systems and provide references. Unfortunately, due to differences in the types of queries and corpora supported, we cannot provide a fair comparison of all the querying systems in terms of performance. However, we will discuss such comparisons when possible in the rest of this Section.

Table 3.2: Summary of the literature on query languages over syntactically annotated trees. Refer to the text for the meaning of abbreviations.

Year	**Query**	**Express-**	**Target Corpus**		**Querying**		**Architecture**	**Performance**	**Ref.**
Started	**System**	**iveness**	**Language**	**Type**	**Method**	**Medium**		**Reported**	
1994	Tgrep	Axxx	English	SW	Scan	Disk	Native	N/A	[97]
1998	NetGraph	AxxV	Czech	MW	Scan	Disk	Native	N/A	[83, 84, 86]
1999	ICECUP	ABxx	English	MW+MT	N/A	N/A	N/A	N/A	[111, 112]
2000	CorpusSearch	AxxV	English[1]	SW	Scan	Disk	Native	[9]	[100]
2000	NXT Search[2]	ABQV	German	SW+ST	Scan	Memory	Native	[80]	[54, 35, 80]
2001	Tgrep2	ABxV	English	SW	Scan	Disk	Native	[9, 42, 24]	[101]
2001	Emu	xxxx	English	SW+ST	N/A	Memory	N/A	N/A	[15]
2001	TIGERSearch	ABxV	German	SW	Index	Memory	Native	[79, 24]	[64, 63]
2002	Emdros	xxxV	generic corpora		Index	Disk	Relational	[96]	[95, 96]
2002	VIQTORYA	xBxV	German	SW	Index	Disk	Relational	N/A	[105]
2003	FSQ	FOL	German	SW	Scan	Disk	Native	[79]	[60]
2004	MONASearch	MSO	German	SW	Scan	Disk	Native	[79]	[61, 79]
2004	LPath	ABxx	English	SW	Index	Disk	Relational	[9]	[9, 69, 67]
2005	LPath$^+$	FOL	English	SW	N/A	N/A	N/A	N/A	[68, 69]
2005	LQL	xBxV	English	MW	Index	Disk	Relational	[89]	[89]
2006	DDDQuery	AxxV	German	MW	Index	Disk	Relational	N/A	[36]
2006	Tregex	ABxV	English	SW	Scan	Disk	Native	[24]	[70]
2007	ANNIS2	xxxx	Multilingual[3]	MW+MT	Index	Disk	Relational	N/A	[65]
2007	TreeAligner SM	ABxV	Multilingual[4]	MW	Index	Memory	Native	N/A	[109]
2009	PML-TQ	AxxV	Czech	MW	N/A	Disk	Relational	N/A	[92]
2010	LPath-IR	Axxx	English	SW	Index	Disk	Native	[42]	[42]
2010	TPE	Axxx	English[5]	SW	Scan	Disk	Native	[24]	[24]

[1] Also Portuguese and French, see `http://corpussearch.sourceforge.net/CS-manual/Corpora.html`
[2] The query language is called NiteQL or NQL
[3] Mostly German but also other languages such as Hindi and African languages
[4] English, German and Swedish
[5] Text abstracts from PUBMED containing protein names

Expressiveness of Query Languages

Expressiveness, or the expressive power, of a query language measures the range of elements expressible in that query languages. In Table 3.2 we use a raw metric for reporting the expressiveness of the query languages. Aside from languages which are as expressive as First-Order Logic, denoted by FOL, or as expressive as Monadic Second-Order Logic, denoted by MSO, we use four characters to roughly describe the expressiveness of the other query languages. Note that FOL is strictly more expressive than non-FOL languages and MSO is strictly more expressive than FOL.

In the third column of Table 3.2, the first letter, A, refers to the language supporting the navigational axes. For an A to appear in the expressiveness column of a query language, it is usually enough if it supports parent (or child), ancestor (or descendant), immediate following (or immediate preceding), following (or preceding) and any of the sibling axes. If the language does not support enough axes or if we cannot find enough clues in the querying system manual or references, we mark it by an x in the respective column. The second letter, B, refers to supporting boolean operators over expressions. We require that the language supports negation, and either of conjunction or disjunction. Q refers to the support of universal quantifier ($\forall$) and existential quantifier ($\exists$). Languages which support both of these quantifiers usually explicitly define the notation of implication ($\rightarrow$) as well. Moreover, nodes or variables are often existentially quantified, thus Q usually reduces to finding if the query language supports universal quantifiers. Finally, V refers to the support for defining variables over expressions. Using variables not only makes the task of composing the queries easier, but also allows the definition of cycles in the query pattern.

If any of the the above four fields is marked with an x, it means that the language does not support the given feature or we could not find a clue. Note that not all the querying features over syntactically annotated trees can be expressed using the above four letters. For example edge alignment, scoping and regular expressions are not accounted for in our formulation, while are supported by a large number of reviewed systems. Also note that the semantics of operators and matchings might vary from system to system. As a result, the same query might be evaluated differently by two querying systems which support it.

Table 3.2 also reports on the properties of the *Target Corpus* for each query system, including the language and the type of the corpus. *SW* refers to a Single-layer Word-aligned corpus, hence a treebank, and *MW* refers to a Multi-layer Word-aligned parallel corpus. Finally *ST* and *MT* refer to a Single-layer or a Multi-layer Time-aligned parallel corpus.

Our work in this thesis has an expressive power of $Axxx$, given the above formulation, and focuses more on subtree matching. Our solutions are not bound to a specific language or a parser, however, they only operate on single-layer word-aligned corpora.

Performance

A large number of the query languages discussed above lack greatly on addressing the issue of performance. There are only a few systems that report the runtime of some queries on their system, and even those systems do not report on the other performance measures such as the size of the indexes, memory requirements or index construction time. The sizes of the corpora used are usually small, ranging from a few thousand sentences to a few million sentences.

There are a large number of query systems whose performance is linear on the size of the corpus. Examples of such systems, as denoted in Table 3.2 by Scan under querying method, are tgrep2, FSQ (Finite Structured Query), MONASearch, Tregex and TPE (Tree

Pattern Expression). In order to do their matchings, they have to compare each query with every tree in the corpus. Their query execution time is thus determined by both the corpus size and the time it takes to evaluate a query against a data tree.

The approaches that scale well are usually the index-based approaches. Among the few reported in Table 3.2 only four have their performance reported either by themselves or as benchmarks for other methods. These four are LPath, LPath-IR, LQL and Emdros. LPath-IR [42] uses a large datasets of a few million sentences and reports on the performance analysis with different types and sizes of queries. However, all the queries it reports the performance for are path queries. Moreover, they only return the trees which match a given query, without finding the individual matches within the same sentences.

LQL [89] very briefly reports the performance of its queries over a corpus of 10 million sentences to be in the order of a few minutes. However, it does not elaborate on the experimentation setup and the details of their access methods. Emdros [95, 96] supports querying on generic corpora and has a relatively extensive study of its performance. Unfortunately, the type of corpora and the queries reported makes it difficult to compare the performance of Emdros with other systems. Finally, LPath [9] compares its performance with tgrep2 and CorpusSearch as its baseline methods and outperforms both for most queries.

In comparison with our Subtree Index in this thesis, LPath and LPath-IR have the most similar approach as they both develop indexes over structural information of the trees. Our Subtree Index is similar to LPath in the case where the size of the largest subtree stored is 1, thus storing only nodes of the trees. However, LPath and LPath-IR do not consider the injective matching assumption. As a result, the result set of queries could be different when there are multiple nodes with the same label within the query tree.

3.2.2 Query Languages over XML Documents

EXtensible Markup Language (XML) is extensively used for storing semi-structured data. Similar to syntactically annotated trees, XML documents can be modeled as unranked node-labelled trees. Unlike syntactically annotated trees which could be modelled as ordered or not ordered, XML documents are often modelled as ordered trees. Query languages such as XPath [27] and XQuery [29] over XML documents, support similar functionalities as the query systems presented in Section 3.2.1. XPath [27] for example, supports querying over navigational axes such as child, following and self. XQuery uses XPath as its navigational building block and adds FLWOR[1] expressions. In this section, we briefly discuss the se-

[1]For-Let-Where-Order By-Return

mantics of XQuery and XPath queries and the related work on efficiently processing such queries.

Index Graph Schemes

As an alternative to numbering schemes which assign unique set of numbers to data tree nodes, a set of approaches have been developed whose main idea are to summarize XML data such that structural information is preserved. Examples are the work on *Strong DataGuide* by Goldman and Widom [43] and *1-Index* by Milo and Suciu [81]. These approaches use a similar approach used for converting a non-deterministic finite automaton (NFA) to a deterministic one (DFA). As a result, the structure of all XML trees are preserved within the index graph scheme and at each state or particular node label, it would be clear which other nodes/labels are reachable. Query processing is done by navigating these summaries, which are in practice much smaller than the actual data, specifically for tree structured XML data.

XPath

XPath is a path query language. Every query consists of a context node (from where the search begins) and a set of axis steps which narrow down the result of the query. Hence, the result of an XPath query is a set of nodes matching the context node in the query tree.

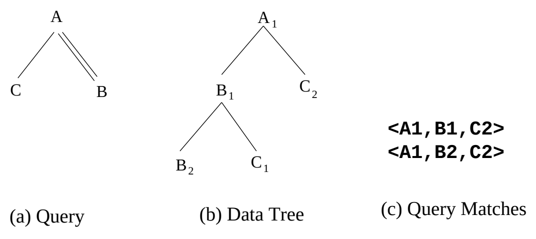
Most of the literature around XPath is due to the work of Grust [50] and Gottlob [45]. This line of work mostly focuses on using the querying power of a relational engine to support XPath axis steps. For example, Grust [50] presents an encoding that maps each node to a point in a two-dimensional plane of the node pre-order and postorder ranks and converts each XPath axis query into a range query over this plane. He suggests using an R-Tree, in order to efficiently support the range queries. Staircase join [51] is another relational approach that stores a set of structural information of XML trees within a database to achieve a better performance for XPath axes over a sequence of context nodes. It performs three additional optimizations over Grust's XPath Accelerator [50], namely pruning, partitioning and skipping. The basic idea behind these optimizations are to avoid redundant checks and evaluations, when performing an axis evaluation over a sequence of context nodes.

XQuery

XQuery [29] is a Turing-complete general purpose language supporting a broad set of querying functionalities such as variable and function definitions, recursion and loops. As a result, it is far more expressive than XPath. A complete review of the work on efficiently supporting different aspects of XQuery is out of the scope of this thesis. As a result, we only cover a few closely related to our work. As in XPath, relational databases have been the backbone for supporting XQuery. DeHaan et al. [31] propose an implementation strategy based on *dynamic intervals* coding for storing XML in relations and mapping XQuery to SQL. Their strategy supports function definitions and nested FLWR expressions. Chen et al. [23] introduce *Generalized Tree Patterns* (GTPs) which augments tree patterns with optional edges and boolean operators over nodes. These GTPs are shown to support significant XQuery features such as nesting, aggregation, quantifiers and joins. Finally, Boncz et al. [10] introduce loop-lifted staircase join for efficiently supporting XPath expressions within XQuery nested `For` loops.

Twig Queries

The most commonly used axes in XPath and XQuery are the child axis and the descendant axis, denoted as $/$ and $//$ (also `p-c` and `a-d` axes or `pc` and `ad` axes), respectively [47]. A twig query is an (unordered) unranked node-labelled tree, whose edges indicate either a child or a descendent relationship, denoted by a single or a double line, respectively. There has been an extensive line of research focusing on the *Twig Pattern Matching*, which looks at the problem of finding matches for nodes of a query tree within a set of XML trees. Thus, a match is a set of data nodes having the relationship given in the query tree. An example of a twig query, a tree representation of an XML document and the corresponding query matches are illustrated in Figures 3.3(a),(b) and (c), respectively. Note that in this figure A_1 belongs to the class of nodes that have label of type A. Similarly, B_1, B_2 belong to label type B and C_1, C_2 belong to label type C. We denote query nodes by only the label types, which can match any of the data tree nodes belonging to the same class.



```
<A1,B1,C2>
<A1,B2,C2>
```

Figure 3.3: Sample query, data tree and match results for a twig matching problem.

In the rest of this section, we will discuss two major approaches to solving twig queries, relational and native. Most of these approaches use the numbering schemes and structural joins discussed in Section 2.2.4 as their building blocks. We further discuss how our subtree index over syntactically annotated trees might overlap with or benefit from the approaches reviewed in this section, and clarify our contributions compared to these approaches.

Relational Approaches

The relational approaches for tackling twig matching benefit from the rich storage mechanisms, query optimization and concurrency control, developed over relational engines. In these approaches, the structural information of the trees are stored in relational tables and twig queries are mapped into SQL queries that can retrieve the matches. In general, while relational approaches are easier to implement and they benefit from existing development in relational databases, they cannot address certain performance issues related to twig queries. For instance, unless the structural join is integrated within a database engine, the size of intermediate results from partial solutions might be very large while evaluating twig queries.

In general, native join approaches are more efficient and favorable for matching twig queries.

The common relational approaches for storing XML are (1) the edge approach, (2) the node approach, (3) path materialization, and (4) the DTD approach. In the edge approach [40], an XML document is considered as an edge-labelled tree. Edges are stored in a relational table with (Source, Target) pairs as keys, indicating the source and target nodes of an edge. The edge approach could be very inefficient for long queries and queries containing $//$ axis, as many joins might be necessary. The node approach [120] stores node information such as the interval codes, parent id and the node tag in a relational table. It is more efficient for queries with $//$ axis than the edge approach. The path materialization approach avoids expensive joins by pre-materializing paths, either from the root to nodes as in [118] and using the interval coding or from nodes to the root as in [93] and using a variant of Dewey coding. The twig queries are decomposed into path queries and the SQL *LIKE* function is used to perform path matchings. The path materialization might be inefficient for queries with multiple $//$'s and might produce incorrect results for trees with recursion [47]. The DTD approach uses the relationships and data types defined in the XML Document Type Descriptors (DTDs) to design the architecture of the relational tables, hence reducing the number of joins [102].

Of the above relational join approaches, the first three could be considered as special cases of our subtree index, ignoring the injective matching assumption. The node approach is similar to the case where we only store subtrees of size 1 and the edge approach can be represented with the scenario where we only store subtrees of size 2. The path materialization would be the scenario where we only keep subtrees that consist of unary branches rather than all subtrees. As we show in our experiments in Chapter 6, subtree index performs better for larger sizes of subtrees, outperforming the node and edge approaches. The path materialization approach would not be suitable for our matching tasks as it does not distinguish between paths that have nodes in common and distinct paths. For instance trees such as `T1:A(B(C)(C))` and `T2:A(B(C))(B(C))` will both be decomposed into two `A(B(C))` paths.

Native Twig Join Approaches

Given the interval codes over XML nodes, evaluating twig queries requires the use of θ-joins (joins involving inequalities) as a fundamental operation. These joins can be expensive in relational engines as large intermediate results might be generated. Native twig join approaches try to perform structural joins as efficiently as possible. These approaches assume that the structural information of nodes, i.e. *DocId, Pre, Post, Level* values are stored in an inverted list, sorted on *DocId* and *Pre* values.

Multi-Predicate MerGe JoiN (MPMGJN) [120] is an adaptation of the merge-join for parent-child and ancestor-descendant queries. Given a query A/B (or $A//B$), the algorithm instantiates two cursors at the beginnings of the sorted lists for A and B and for each A posting, iterates through all B postings, where $A.DocId = B.DocId$ and $A.Pre \leq B.Pre \leq A.Post$. One problem with this approach is that an A node is compared with all its B descendents, even for A/B queries. Motivated by this, StackTree [1] uses a global stack to push A nodes as they are seen in the sequence of the input stream. Thus, every B node is only compared to the top of the stack for parent-child axes, reducing the overall number of comparisons.

Both algorithms operate on a single query axis at a time and the partial results from these axes have to be computed and joined, which could be costly if the size of the intermediate results is large. PathStack [11] reduces the size of the intermediate results by decomposing twig queries into several root to leaf paths. It then solves for these paths and joins the results. Instead of using one global stack, PathStack uses one stack for each label in the query. Nodes are pushed into their corresponding stacks once they are seen in the input stream with a pointer to the top of their parent stack. Once a leaf is observed, solutions are enumerated and sent to the output. TwigStack [11] is the first holistic twig join algorithm. Similar to PathStack, it uses multiple stacks for labels in the query and uses a *getNext()* function to filter intermediate results that will not be part of a final match. *getNext()* guarantees that no data node, say A_1 in Figure 3.3, is pushed over its corresponding stack unless all its descendents in the query tree exist in the subtree rooted by A_1 and its children have this property recursively. TwigStack is shown to be optimal for queries with only $//$ edges. By optimality we mean no redundant intermediate results are pushed over the stack. Thus, as for TwigStack, every node that is pushed over a stack in a $//$ only query, contributes to a solution. Thus, an optimal solution is linear on the sum of the input and output sizes and is independent of the intermediate results generated. TwigStack is sub-optimal when used for general twig queries.

The problem of optimality has been the focus of interest for other twig join algorithms. TwigStackList [73] uses lists to cache limited number of nodes in the memory and obtains optimality for twig queries in which $/$ edges are under non-branching nodes only. iTwigJoin [20] extends the TwigStack by adding several inverted sublists based on the level of the nodes in the data trees, achieving optimality for queries with only $/$ axis as well as $//$ only queries. $Twig^2Stack$ [19] is the first optimal algorithm for addressing the twig join problem. It uses a complicated list of trees of stacks data structure for storing the intermediate results and enumerating the solutions. Recently, Grimsmo et al. [48] proposed TJStrictPre and TJStrictPost, which are both optimal and fast.

The subtree index developed in this thesis uses a native approach and therefore it could benefit from the above join approaches. However, it should be noted that certain tweaks have to be applied for such algorithms to work with subtree index. First, twig pattern matching does not make the injective matching assumption and the matches obtained for a twig match have to be further pruned. Moreover, stack based approaches such as StackTree and TwigStack require some modifications over non root-split coding scenarios as the posting list of subtrees are sorted on the *Pre* values of the subtree roots. However, joins could be performed over non-root nodes of subtrees, whose lists might not be sorted on their *Pre* values.

3.2.3 Querying over Trees

In this section we briefly review some of the related works on querying over general tree structured data. These include Unordered Tree Pattern Matching (UTPM), Ordered Tree Pattern Matching (OTPM) and Approximate Subtree Matching (ASM).

Unordered Tree Pattern Matching

Chen and Cooke study the problems of Unordered Tree Pattern Matching (UTPM) and strict UTPM [21]. UTPM and strict UTPM study the problem of pattern matching over general trees using the same mapping functions as the ones we define in definitions 2.2.1 and 2.2.2, respectively. They show that strict UTPM is NP-complete and propose an algorithm for UTPM that runs in $O(|D||Q|)$ time, where D and Q are data and pattern trees, respectively. Götz et al. [46] study the problem of *tree homeomorphism*. In their approach all query edges will be mapped to ancestor-descendant axes over the data tree.

Similar to our approach, there are works that use index elements larger than nodes or edges to improve the performance of querying. Shasha et al. [103, 104] proposed ATreeGrep, which facilitates approximate and exact matching over unordered trees. ATreeGrep stores all paths of the set of input trees into a suffix array. It also uses a hash index over all nodes and edges to filter a set of candidate trees and improve overall querying performance. Tree matching is done by decomposing the query tree into its root to leaf paths, and evaluating them against the suffix array. In contrast to our subtree index, ATreeGrep does not support distinct labels over different children of a node. It also does not support single node queries. Moreover, our subtree interval and root-split codings remove the need for post-validations. As a result, as confirmed by our experiments, our subtree index using root-split coding performs orders of magnitude faster than ATreeGrep.

Ordered Tree Pattern Matching

The problem of Ordered Tree Pattern Matching (OTPM) refers to finding exact matches of an ordered query tree Q over a a node-labelled ordered data tree D. More specifically, the task is to find a mapping between nodes in Q and nodes in D, such that the same parent-child and sibling order relationship that exist between nodes in Q exist between mapped nodes in D. The obvious approach to solving OTPM is to try matching every subtree of D with Q which takes $O(mn)$ time [56], where n is the size of the data tree and m is the size of the query tree. Faster approaches [33] use suffix trees and solve the problem in $O\left(n\sqrt{m}polylog(m)\right)$, achieving a more efficient solution to this problem. To the best of our knowledge the fastest approach proposed so far achieves $O(n \log^3 m)$ [28].

Approximate Subtree Matching

Given a pattern tree Q, (Top-k) Approximate Subtree Matching ((T)ASM) refers to finding subtrees of a large data tree or a forest D that are similar to Q. The similarity is usually defined as the tree edit distance [121] which is the minimum number of tree operations to convert one tree to another. Two subtrees are considered similar when their distance is smaller than a certain threshold. The output is usually a sorted set of subtrees, ranked based on how similar they are to the query pattern. Guha et al. [52] address the problem of approximate joins over XML documents. They introduce the notion of a reference set, which is a projection of the join sets into the metric space to reduce the size of the joins. They also propose inexpensive computation of upper and lower bounds on the distance between two XML documents that effectively prunes a large number of join candidates. Similar to the idea of relevance ranking in information retrieval, query trees can be scored and ranked. Amer-Yahia et al. [3] propose path and twig scoring over twig queries that capture structure and content of the trees. Finally, recently Augsten et al. [6] proposed an algorithm to compute an upper-bound on the size of the subtrees in the data tree based on the query tree. Then, they proposed a prefix ring buffer to prune the subtrees that are above this size in one scan of the data tree. Approximate Subtree Matching can be of interest as a future work on querying linguistic trees. As an example, given the non-uniform distribution of labels over the data and query trees and the localities that exist among the labels, heuristics can be developed to lead the search into regions of the data tree more prone for matches in order to find the top-k matches faster.

3.2.4 Querying over Graphs

There exist annotations over natural language text that cannot be modelled using trees and require to be represented using DAGs or general graphs. Moreover, Indexing and querying graphs have lots in common with trees and are relevant. In this section we review the major recent works regarding the index structures and query algorithms over graphs. Graph indexing can be divided into two major categories. Exact subgraph matching (subgraph isomorphism) and approximate (similarity) subgraph matching. Both problems are NP-hard in their general form and as a result most solutions are either computationally prohibitive for large graphs or use heuristics for reducing the problem size.

Exact Matching

GraphGrep [103] addresses the problem of exact subgraph matching over undirected node-labelled graphs. It stores all paths of sizes less than a threshold (usually a small number) found in the graph database into hash tables. The query graph is traversed in depth-first order and shredded into multiple paths, which are evaluated against the hash tables for matches. GIndex [117] discusses why path-based indexing schemes for graphs could produce incorrect results or be inefficient and proposes a frequent sub-graph mining algorithm in order to store frequent sub-graph structures instead of paths. They use a larger support threshold for larger frequent sub-graphs in order to avoid storing exponentially many subgraphs and therefore they are able to reduce the index size. GIndex is shown to outperform GraphGrep by an order of magnitude. The work by Williams et al.i [113] on subgraph isomorphism, stores the canonical forms of all subgraphs into an index. However, they assume that the input graphs are very small, making it possible to compute and store the exponentially many subgraphs of all sizes. Their approach cannot be used over parse trees as computing subtrees of all sizes takes tremendous amount of time to complete for a single parse tree. TreePi [123] uses frequent subtrees as elements of its index for subgraph isomorphism problem. TreePi prunes the search space of candidate graphs, and finds the set of matches using post validations. Compared to TreePi, our approach stores all subtrees up to a certain size and performs exact matching over the index. Moreover, our root-split and subtree interval codings do not require any post validations. As we discuss in Chapter 6, an adaptation of TreePi to indexing parse trees results in smaller index sizes, but a worse querying performance compared to our root-split coding.

Approximate Matching

He and Singh [53] propose C-Tree, an R-tree based index that supports both exact and approximate subgraph queries. C-Tree stores a hierarchy of graphs in a tree. For similarity queries, they use the edit distance as their measure of similarity and support k-nearest neighbor and range queries. SAGA [107] introduces an approximate subgraph matching technique that accounts for node gaps, structural mis-matches and node mis-matches. SAGA uses a similarity measure based on the distance of two subgraphs. It prunes and stores subgraphs having nodes up to a certain threshold in an index for fast look-ups. The matching algorithm works by shredding the query into small fragments, finding the matches to the small fragments and joining the results. TALE [108] proposes a heuristic approach to efficiently support approximate subgraph matching over large graphs. It introduces neighborhood index which is a disk-based augmented index that stores certain properties about the adjacent nodes of each node in the graph. Matching is done based on certain properties holding for the node and its neighborhood. TALE does not guarantee to find all matches or that matches don't have false positives, however, it reports high precisions and recalls in the experiments.

3.3 Natural Language Question Answering

In this section, we review some of the works in the literature around question answering over natural language text that could potentially benefit from our index structures and access methods in this thesis.

Question Answering (QA) refers to the task of finding “correct” responses to questions expressed in a human readable natural language. QA in open domains is a challenging task as questions and answers can be expressed using various syntaxes and wordings. Many different types of questions have been studied, but recent QA systems have made more progress with factoid, list and definition questions mostly due to their relative simplicity. Factoid questions are questions whose answer is a fact or a set of facts and list questions are an extension of factoid questions specifically denoting the output to be a list of facts.

These three types of questions have been the main focus of the Text REtrieval Conference (TREC) QA track. The state-of-the-art systems are able to find a correct answer to around 70% of the factoid questions within their first try [30] on a corpus of blogs and a corpus of news. During the several years in which TREC held the QA tracks, some of the best systems achieving very high accuracies such as PowerAnswer for TREC-2002 [87] have used syntactic parsing to obtain more accurate results. Moldovan et al. in [88] show that using the linguistic information in QA such as part of speech tags and syntactic parsing significantly improves the recall of the answers. However, there is a trade-off between the accuracy obtained and the performance of these systems. Finally, Ittycheriah et al. [59] propose techniques for constructing syntactic queries that can lead to effective answers. They model the answer selection as a classification task which maximizes the conditional probability that the query has matches, and thus try to maximize the number of matches for each query.

Recently, automated QA systems have focused mostly on question answering over the world wide web (See [72] for a survey of techniques). Most techniques developed over the web benefit from statistical methods and redundancy of answers to compensate for the low accuracy of a large scale open domain question answering task. Due to the large volume of pages over the web, most QA systems over the web consist of (1) an IR engine which extracts the relevant documents that might contain the answers, (2) a question analyzer, which based on the question type, generates an appropriate query and sends it to the IR engine, and (3) the answer extractor which finds the answers in the retrieved documents. MULDER [66] is one of the earliest question answering systems over the web. It extensively uses syntactic parsing in several components of its system such as finding the question type and parsing the text snippets returned from Google searches. They report that the recall of their QA system is around 60% for the top 1000 answers for the TREC-08 questions. With efficient querying systems available over large text collections, QA systems can benefit from more efficient access methods and querying algorithms over syntactic relationships.

Chapter 4

Sequential Indexing and Querying of Natural Language Text

In this chapter, we discuss the problem of indexing and querying natural language text in the scenario where text is modelled as a sequence of words. Specifically we study index structures and querying algorithms for handling word-level wild card queries. Our study includes a review of a few baseline methods and a discussion of how they handle wild card queries. Further, we introduce Word Permuterm Index (WPI) as a novel and efficient indexing scheme, study its architecture and analyze its performance.

4.1 Baseline Methods

Without loss of generality, we consider phrase queries that have exactly one wild card and any number of non-wild card terms, referred to as literals. For queries with multiple wild cards, one can find the matches for query sub-sequences that have only one wild card, substitute the wild card with its matches and look for further matches.

Next, we introduce a few baseline access methods that are used within natural language applications and study their performance. Regardless of the access strategy, a wild card query evaluation can be often divided into two phases: (1) *Binding phase*, where the indexed elements (e.g. sentences, paragraphs or documents) are filtered based on the query literals that are present and maybe their order, and (2) *Matching phase* in which filtered elements are scanned and the keywords that match the wild card are retrieved.

4.1.1 Full Scan

A straightforward approach for answering wild card queries is to scan the dataset elements one by one and check every element for possible query matches. If the dataset fits in main memory, a full scan may not be a bad idea given that the initial cost of loading is negligible when amortized over a reasonable-sized set of queries.

4.1.2 Inverted Index

As illustrated in Figure 3.1, our implementation of an inverted index stores a linear vector of posting triplets $< d, f_{t,d}, [o_1, \cdots, o_{f_{t,d}}] >$. Wild card query evaluation over inverted index can be easily adapted from the standard implementations of keyword queries. Keyword queries are evaluated by intersecting the posting lists of query literals and finding the matching documents and corresponding offsets. The key idea behind wild card query evaluation is to sequentially scan these documents and to find and extract the wild card matches. Thus, in order to do the wild card matching we need to store and access the text dataset as well.

The complexity of wild card matching over an inverted index is $O(\sum_{i=1,q_i \neq \%}^{m} \|q_i\|) + O(\|Q\| \cdot |d_{avg}|)$, where $\|P\|$ is the number of bindings of a pattern P[1]. The first expression gives the cost of matching query keywords and retrieving partial matches. Since we have to go through all the matching elements in order to find the wild card matches, this cost is $\|Q\| \cdot |d_{avg}|$, where d_{avg} is the average size of an indexed element.

4.1.3 Neighbor Index

As introduced in Section 3.1.2, a neighbor index is an inverted index that stores for each term both its left and right neighbor terms within its postings. The original implementation of the neighbor index stores for each neighbor in addition to the term, both its part of speech (e.g. noun-phrase) and its role (e.g. term). In our sequential modeling of text, tags are not explicitly used in the queries, thus we implement a simplified version of the neighbor index, where for each offset, only one left neighbor and one right neighbor are stored with no further information. Therefore, the structure of a posting in our implementation of the neighbor index looks like $< d, f_{t,d}, [(o_1, l_1, r_1), \cdots, (o_{f_{t,d}}, l_{f_{t,d}}, r_{f_{t,d}})] >$, where l_i and r_i are the left and right neighbors of the i'th occurrence of t in d, respectively.

Given that neighbor index is an inverted index, the algorithm for evaluating wild card queries over neighbor index follows the same bind-and-match process of any inverted index, except that the *matching phase* is much less costly. Once the matching documents and offsets are found, the wild card matches can be extracted in without referring to the documents. Thus, the running time of wild card query evaluation over a neighbor index will be

[1]The number of documents matching P

$O(\sum_{i=1,q_i \neq \%}^{m} \|q_i\|) + O(\|Q\|)$.

4.2 Permuterm Index over Natural Language Text

This section presents our Word Permuterm Index (WPI) as an efficient access method that supports wild card querying over natural language text. WPI is an adaptation of the permuterm index [41, 39] and as such it has the following three components. (1) A word level Burrows-Wheeler (BW) transformation of text [12], (2) an efficient mechanism to store and access the alphabet, and (3) an efficient mechanism to access the ranks. Next, we discuss these components in more detail.

4.2.1 Word Level Burrows-Wheeler transformation

Burrows-Wheeler transformation (BWT) is a reversible transformation that is used in well-known compression algorithms such as bzip2 and is believed to give a permutation that is more amenable to compression. The transformation, when applied to a character string, can change the ordering of the characters in the string but not their values. Our work applies BWT to words instead of characters; a word-level transformation has some interesting properties especially in answering wild card queries.

Assume we are given a dataset containing three sentences `S1:'Rome is a city'`, `S2:'countries such as Italy'` and `S3:'Rome is the capital of Italy'`, and we would like to index them using WPI. Adapting the ideas proposed by Manning et al. [76] and Ferragina and Venturini [39], we sort this dataset lexicographically[2] and use the \$ symbol, to mark the sentence boundaries and the ~ symbol, to mark the end of the dataset. This results in our dataset to look like `T:'$ Rome is a city $ Rome is the capital of Italy $ countries such as Italy $ ~'`.

A word-level BWT is obtained by (1) computing all the cyclic rotations of the words, (2) sorting the rotations, and (3) finding the vector that contains the last word in the rotations in the same order after the sorting. Figure 4.1 depicts the result of applying these three steps to T in the given example. Note that the set of sentences are rotated by one word at each level. We denote the vector of last words, BW-transformation, by L and the sorted vector of first words, by F.

BWT has some very interesting properties. First, for any word in T, the j'th occurrence of the word in L and the j'th occurrence of the word in F correspond to the same word of the sequence. For instance, the second occurrence of the word `'Italy'` in L is preceded

[2]Sorting guarantees nice properties on BWT, See Section 4.2.4

i	F		L
1	$	Rome is a city $ Rome is the capital of Italy $ countries such as Italy $	~
2	$	Rome is the capital of Italy $ countries such as Italy $ ~ $ Rome is a	city
3	$	countries such as Italy $ ~ $ Rome is a city $ Rome is the capital of	Italy
4	$	~ $ Rome is a city $ Rome is the capital of Italy $ countries such as	Italy
5	Italy	$ countries such as Italy $ ~ $ Rome is a city $ Rome is the capital	of
6	Italy	$ ~ $ Rome is a city $ Rome is the capital of Italy $ countries such	as
7	Rome	is a city $ Rome is the capital of Italy $ countries such as Italy $ ~	$
8	Rome	is the capital of Italy $ countries such as Italy $ ~ $ Rome is a city	$
9	a	city $ Rome is the capital of Italy $ countries such as Italy $ ~ $ Rome	is
10	as	Italy $ ~ $ Rome is a city $ Rome is the capital of Italy $ countries	such
11	capital	of Italy $ countries such as Italy $ ~ $ Rome is a city $ Rome is	the
12	city	$ Rome is the capital of Italy $ countries such as Italy $ ~ $ Rome is	a
13	countries	such as Italy $ ~ $ Rome is a city $ Rome is the capital of Italy	$
14	is	a city $ Rome is the capital of Italy $ countries such as Italy $ ~ $	Rome
15	is	the capital of Italy $ countries such as Italy $ ~ $ Rome is a city $	Rome
16	of	Italy $ countries such as Italy $ ~ $ Rome is a city $ Rome is the	capital
17	such	as Italy $ ~ $ Rome is a city $ Rome is the capital of Italy $	countries
18	the	capital of Italy $ countries such as Italy $ ~ $ Rome is a city $ Rome	is
19	~	$ Rome is a city $ Rome is the capital of Italy $ countries such as Italy	$

Figure 4.1: Sorted permutations of a sample set of sentences and the first and last word lists, F and L.

by `'as'`, and so is the second `'Italy'` in F; hence, $L(4) = F(6)$. Second, for every row, $L(i)$ precedes $F(i)$ in T. Given these two properties, Ferragina and Manzini [38] propose the following function for traversing L in backward order:

$$LF(i) = C[L[i]] + Rank_{L[i]}(L, i)$$

where $C[L[i]]$ is the number of words smaller than $L[i]$ and $Rank_{L[i]}(L, i)$ is the number of times $L[i]$ appears in the sub-sequence $L[1..i]$. $LF(i)$ tells where the element preceding $L[i]$ in T is located in L. E.g. $LF(6) = C['as'] + Rank_{'as'}(L, 6) = 9 + 1 = 10$ and $L(10)$ is `'such'` and is the word preceding $L(6) =$`'as'` in T. Since T is sorted, one can start from $L(1) = F(n)$ and repeatedly call LF to find $L(n) = F(1)$, traversing the whole text in backward order. Therefore, L is reversible, meaning that given L, any sub-sequence of words in T can be re-constructed. We can use this property to turn L into an index that can support searches over word sequences. The challenges would be to support a wide range of wild card queries and to efficiently support access to C and $Rank$, required for traversing L in backward order. Next, we discuss these challenges and the proposed solutions.

4.2.2 Maintaining the Alphabet

A major difference between the permuterm index and WPI is in the size of their alphabets. The alphabet in permuterm index typically consists of ASCII characters and symbols which are small in size and are not required to be explicitly stored. However, the alphabet size in WPI grows with the size of the text dataset almost linearly. When $|\Sigma|$ is in the order of millions, efficient access to alphabet elements, their ordering and their frequency is crucial.

In order to provide efficient access to Σ, we built one array and one hash table. The array stores the elements of Σ in ascending order, therefore the first element is always \$ and the last is ~. The array helps to find which alphabet element is represented by which numerical code, which is its index in the array. Coding the alphabet is essential for efficient implementation of algorithms such as `backwardSearch` and `Rank`. Without coding, we will not be able to achieve the time complexities we later report for these algorithms. Moreover, coding reduces the index size, replacing a keyword and a delimiter by a code which uses smaller number of bits.

The hash table stores the same information in the reverse order; given an alphabet element, the hash returns the code of the element, together with its frequency and cumulative frequency, C. Thus, $C[t]$ counts the number of alphabet elements in the whole dataset that are smaller than t. In the above example, $|\Sigma| = 13$ and the hash table provides constant-time access to C values for all the alphabet elements.

4.2.3 Rank Data Structures

$Rank_c(L, i)$ returns the number of occurrences of $c \in \Sigma$ in the prefix $1 \cdots i$ of array L. In order to evaluate queries over WPI, we make frequent accesses to $Rank$ and therefore, quick access would be required. Naive baseline solutions to the rank problem are as follows. (1) Start from the first element in L and compute rank by counting, which has space complexity and average search complexity of $O(n)$. (2) Keep a matrix of all the alphabet elements and all the locations in L and pre-compute all the values. This approach has the optimal constant search time but a space requirement of $O(n|\Sigma|)$, which is too much given the fact that $|\Sigma|$ grows relative to the size of the dataset. Given the large size of our alphabet, we chose a combination of a wavelet tree [49] and a three level architecture to support constant time rank operation over a bit sequence [90].

A wavelet tree is a perfect binary tree, with a bit sequence at each node representing the occurrences of a sequence of alphabet elements. The root represents Σ over L and each leaf represents one of the alphabet elements. A non-leaf node v represents alphabet elements $\Sigma_v = \{e_i \cdots e_j\}$ and contains a bit sequence $B_v = b_i \cdots b_j$. For each $i \leq k \leq j$ we have $b_k = 0$, if $L[k] \in \{e_i \cdots e_{(i+j)/2}\}$ and $b_k = 1$, if $L[k] \in \{e_{(i+j)/2+1} \cdots e_j\}$. The bit sequence at the left child of v will represent elements of Σ in $\{e_i \cdots e_{(i+j)/2}\}$ and the right

```
Rank(Node,f,l,e,i)
1  if $i = j$
2    return nodeRank(Node,i)
   else
3    if $e \leq \lfloor \frac{f+l}{2} \rfloor$
4      return Rank(Node→left, f, $\lfloor \frac{f+l}{2} \rfloor$, e, i - nodeRank(Node,i))
     else
5      return Rank(Node→right,$\lfloor \frac{f+l}{2} \rfloor$+1,l, e,nodeRank(Node,i))
```

Figure 4.2: Rank function computes the occurrences of e in prefix $1 \ldots i$ of L

child represents alphabet elements $\{e_{(i+j)/2+1} \cdots e_j\}$, recursively. Thus, the algorithm for computing rank of an alphabet element $e \in \Sigma$ in prefix $1 \ldots i$ of L, using the wavelet tree, would be as shown in Figure 4.2.

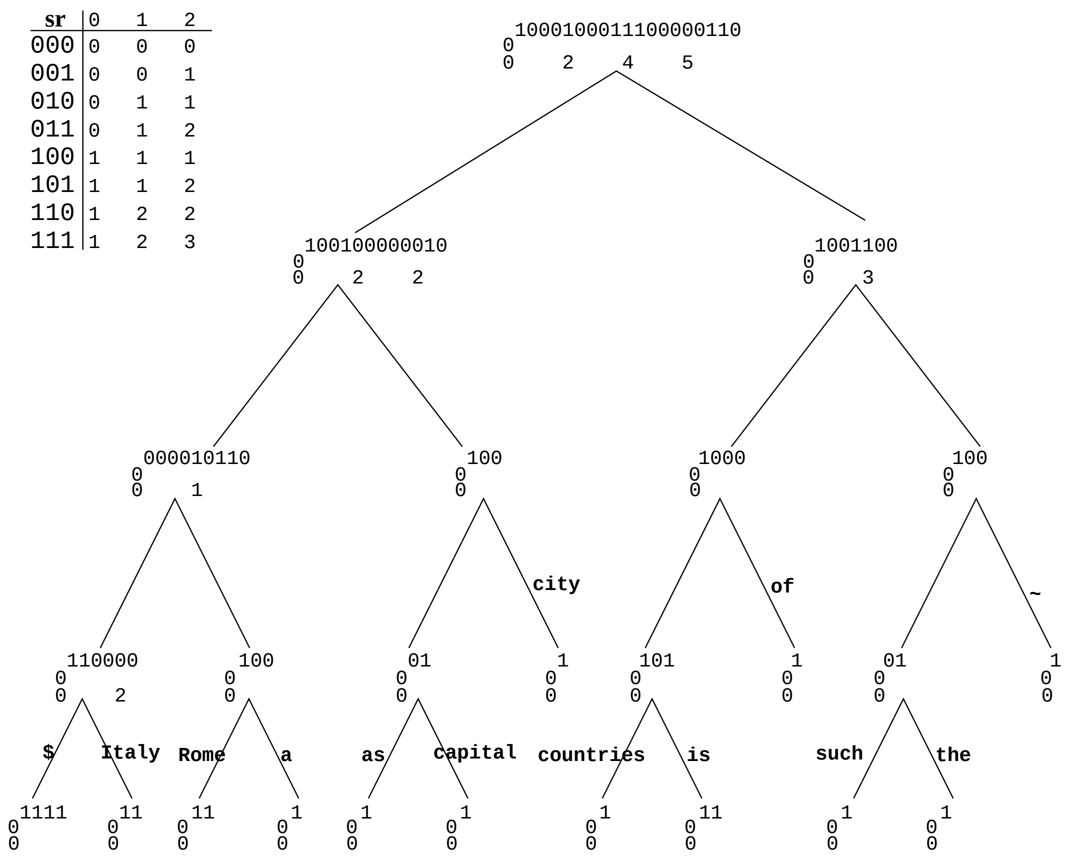
In Figure 4.2, $nodeRank(Node, e, i)$ counts the number of 1's in the prefix $1 \ldots i$ at node $Node$. The count of 0's can be obtained by $i - nodeRank(Node, e, i)$. Counting the number of 1's in each node by sequential scanning is very in-efficient. There are a few solutions that provide constant-time access to binary rank values over a bit sequence [90]. In our work, we use a solution which uses $n + o(n)$ bits of additional storage at each node, where n is the length of the bit sequence in the node. Figure 4.3 depicts our wavelet tree solution over L for the example of Section 4.2.1. For the $nodeRank$ to operate in constant time, two arrays are maintained at each node, namely `sbr` and `br`[3]. For each node, $sbr[i]$ stores the count of 1's in the range $[b_0 \ldots b_{i \times S_b^2 - 1}]$, where $S_b = \lceil \log n \rceil$ and $i \in \{0 \ldots \lfloor \frac{n}{S_b^2} \rfloor\}$. $br[i]$ stores the count of 1's for the range $[b_{\lfloor \frac{i \times S_b}{S_b^2} \rfloor \times S_b^2} \ldots b_{i \times S_b - 1}]$. Finally, a table called `Small Rank (sr)` is pre-populated, which stores the binary rank values for bit sequences of size $t = \lfloor S_b/2 \rfloor + 1$.

Recall that $nodeRank$ function returns the rank of a prefix of the bit string stored at a given node. As depicted in Figure 4.4, $nodeRank$ uses `sbr`, `br` and `sr` arrays to compute the rank in constant time. $nodeRank$ is computed as shown in Figure 4.4. In this figure, $b2d(bs, p, len)$ returns the decimal equivalent of the bit sub-sequence $bs_p \ldots bs_{p+len}$.

4.2.4 Algorithms and Analysis

Ferragina and Manzini in [38] benefit from the properties of the Burrows-Wheeler transformation discussed in Section 4.2.1 and propose `backwardSearch` algorithm, which searches for a pattern over PI in backward order and returns the range of matching strings. A term-level adaptation of `backwardSearch` over WPI is depicted in Figure 4.5. Given

[3]These stand for `super block rank` and `block rank`, respectively



sr	0	1	2
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Figure 4.3: A sample wavelet tree. In each node a bit string and two arrays, `super block rank` and `block rank`, are stored.

a sequence of natural language words $P = p_1 \cdots p_q$, `backwardSearch` finds the range $[first, last]$ of the sorted cyclic rotations prefixed by P. For the example provided in Figure 4.1, `backwardSearch` returns the range $[7, 8]$ for the pattern `P = 'Rome is'`, which is the range of cyclic rotations prefixed by P.

`backwardSearch` makes $O(|P|)$ accesses to C and $Rank$. We adjust the hash table size so that it provides constant time access to hash elements. The wavelet tree access for $Rank$ requires traversing from the root to one of the leaves which requires $O(\log |\Sigma|)$ accesses to the tree nodes. Thus, `backwardSearch` has a complexity of $O(|P| \log |\Sigma|)$.

After adding delimiters and sorting strings as discussed in Section 4.2.1, permuterm index supports wild-card pattern matching over dictionary strings. More specifically, it supports (1) Prefix ($\$\alpha\%$), (2) Suffix ($\%\beta\$$), (3) Substring (γ) and (4) PrefixSuffix ($\$\alpha\%\beta\$$) queries where α, β and γ are arbitrary sequences of characters [39]. We use the above four queries to express our wild card keyword matching over natural language text. Thus in our queries α, β and γ are sequences of natural language text words.

```
nodeRank(Node,e,i)
1  $R_{sbr} = sbr\left[\lfloor\frac{i}{S_b^2}\rfloor\right], R_{br} = br\left[\lfloor\frac{i}{S_b}\rfloor\right]$
2  if $\left(i - S_b \times \lfloor\frac{i}{S_b}\rfloor\right) < t$
3     $R_{sr} = sr\left[b2d\left(Node \to bs, S_b \times \lfloor\frac{i}{S_b}\rfloor, t\right)\right]\left[i - S_b \times \lfloor\frac{i}{S_b}\rfloor - 1\right]$
4  else if $\left(i - S_b \times \lfloor\frac{i}{S_b}\rfloor\right) = t$
5     $R_{sr} = sr\left[b2d\left(Node \to bs, S_b \times \lfloor\frac{i}{S_b}\rfloor, t\right)\right][t - 1]$
6  else if $\left(i - S_b \times \lfloor\frac{i}{S_b}\rfloor\right) > t$
7     $R_{sr} = R_{sr} + sr\left[b2d\left(Node \to bs, S_b \times \lfloor\frac{i}{S_b}\rfloor + t, t\right)\right]\left[i - S_b \times \lfloor\frac{i}{S_b}\rfloor - t - 1\right]$
```

Figure 4.4: A constant-time $nodeRank$, returning binary rank at each node

```
backwardSearch($P$)
1  $i = |P|, c = P[i], first = C[c] + 1, last = C[c+1]$
2  while $((first \leq last)\, and\, (i \geq 2))$ do
3     $c = P[i]$
4     if $1 \leq first \leq M$, increment $first$ by 1
5     if $1 \leq last \leq M$, increment $last$ by 1
6     $first = C[c] + Rank_c\,(L, first - 1) + 1$
7     $last = C[c] + Rank_c\,(L, last)$
8     $i = i - 1$
9  return the range $[first, last]$
```

Figure 4.5: backwardSearch algorithm for traversing L in backward order

Over WPI, we add support for queries such as (5) $\alpha\%$, (6) $\%\beta$, (7) $\alpha\%\beta$, (8) $\alpha\%\beta\$$ and (9) $\$\alpha\%\beta$ where α or β can be in arbitrary places in the document. The set of queries supported by PI are very limited and we often need to search for natural language patterns that are neither a prefix nor a suffix in a document. One other advantage of WPI over PI is that it makes word-level extractions possible. Therefore, even for the first four queries, PI provides character level matches, which are not desired when querying over natural language text. Finally, since WPI performs word-level rotations, it requires fewer number of backward searches compared to PI for the same set of queries.

The key idea behind supporting wild card queries using `backwardSearch` is to convert them into prefix searches over rotations. Table 4.1 gives a summary of how to evaluate wild card queries using `backwardSearch`. In this table, the columns from left to right displays the different types of queries supported by WPI, the pattern(s) to invoke `backwardSearch` with and the range of wild card keyword matches, respectively. The time complexity of the queries are depicted in Table 4.2. As displayed in these tables, the

Table 4.1: Different wild card query patterns over WPI and their corresponding range of matches

Q	**P**	**Wild Card Match**
(1)$\$\alpha\%$	$\$\alpha$	$T\left[I_F\left[first\right] + \lvert\$\alpha\rvert\right] \ldots T\left[I_F\left[last\right] + \lvert\$\alpha\rvert\right]$
(2)$\%\beta\$$	$\beta\$$	$L\left[first \ldots last\right]$
(3)γ	γ	matches documents[1]
(4)$\$\alpha\%\beta\$$	$\beta\$\alpha$	$L\left[first \ldots last\right]$
(5)$\alpha\%$	α	$T\left[I_F\left[first\right] + \lvert\alpha\rvert\right] \ldots T\left[I_F\left[last\right] + \lvert\alpha\rvert\right]$
(6)$\%\beta$	β	$L\left[first \ldots last\right]$
(7)$\alpha\%\beta$	α, β	$T\left[I_F\left[first_\alpha\right] + \lvert\alpha\rvert\right] \ldots T\left[I_F\left[last_\alpha\right] + \lvert\alpha\rvert\right]$ if $\lVert\alpha\rVert \leq \lVert\beta\rVert$ $T\left[I_F\left[first_\beta\right] - 1\right] \ldots T\left[I_F\left[last_\beta\right] - 1\right]$ if $\lVert\alpha\rVert > \lVert\beta\rVert$
(8)$\alpha\%\beta\$$	α, $\beta\$$	$T\left[I_F\left[first_\alpha\right] + \lvert\alpha\rvert\right] \ldots T\left[I_F\left[last_\alpha\right] + \lvert\alpha\rvert\right]$ if $\lVert\alpha\rVert \leq \lVert\beta\$\rVert$ $T\left[I_F\left[first_\beta\right] - 1\right] \ldots T\left[I_F\left[last_\beta\right] - 1\right]$ if $\lVert\alpha\rVert > \lVert\beta\$\rVert$
(9)$\$\alpha\%\beta$	$\$\alpha$, β	$T\left[I_F\left[first_\alpha\right] + \lvert\$\alpha\rvert\right] \ldots T\left[I_F\left[last_\alpha\right] + \lvert\$\alpha\rvert\right]$ if $\lVert\$\alpha\rVert \leq \lVert\beta\rVert$ $T\left[I_F\left[first_\beta\right] - 1\right] \ldots T\left[I_F\left[last_\beta\right] - 1\right]$ if $\lVert\$\alpha\rVert > \lVert\beta\rVert$

[1] See `displayString` in [39] for details

first six queries could be matched with only one call to `backwardSearch`, while the last three require two invocations of `backwardSearch` as the sequence of words are separated by a wild card. For these queries, $first_\alpha$ and $last_\alpha$ are respectively the beginning and the end of the range returned by `backwardSearch` when invoked by α. Recall that `backwardSearch` returns only a range of matching rotations, prefixed by a given pattern. Therefore, it does not provide any efficient support for extracting keyword matches for a wild card. We solved this problem by storing two additional lists, T and I_F, where I_F is the list of locations of elements of F over T; hence $T[I_F[i]] = F[i]$. These lists require $O(n)$ extra space. However, since the overall space consumption of the index is $O(n \log |\Sigma|)$, storing these additional lists will not change the space complexity of WPI.

In Chapter 6 we provide experiments that compare WPI with the baseline methods introduced in Section 4.1 in terms of the query runtimes, index construction time and index size. The next Chapter discusses indexing and querying over structural natural language text.

Table 4.2: The running time complexity analysis of queries in Table 4.1

Q	**P**	**Runtime Complexity**
(1)$\$\alpha\%$	$\$\alpha$	$O(\|\$\alpha\| \log \|\Sigma\|)$
(2)$\%\beta\$$	$\beta\$$	$O(\|\beta\$\| \log \|\Sigma\|)$
(3)γ	γ	$O(\|\gamma\| \log \|\Sigma\|)$
(4)$\$\alpha\%\beta\$$	$\beta\$\alpha$	$O(\|\beta\$\alpha\| \log \|\Sigma\|)$
(5)$\alpha\%$	α	$O(\|\alpha\| \log \|\Sigma\|)$
(6)$\%\beta$	β	$O(\|\beta\| \log \|\Sigma\|)$
(7)$\alpha\%\beta$	α, β	$O[(\|\alpha\|+\|\beta\|) \log \|\Sigma\|]$ + $O[min(\\|\alpha\\|\|\beta\|, \\|\beta\\|\|\alpha\|)]$
(8)$\alpha\%\beta\$$	α, $\beta\$$	$O[(\|\alpha\|+\|\beta\$\|) \log \|\Sigma\|]$ + $O[min(\\|\alpha\\|\|\beta\$\|, \\|\beta\$\\|\|\alpha\|)]$
(9)$\$\alpha\%\beta$	$\$\alpha$, β	$O[(\|\$\alpha\|+\|\beta\|) \log \|\Sigma\|]$ + $O[min(\\|\$\alpha\\|\|\beta\|, \\|\beta\\|\|\$\alpha\|)]$

Chapter 5

Structural Indexing and Querying of Natural Language Text

In this chapter, we consider the scenario where text is available as a collection of syntactically annotated trees, and study the problems associated with indexing and querying. We propose a novel subtree index and a few storage and querying techniques over this index. We present some structural properties of the index and an analytical study of its performance. Specifically, we study how interval coding can be adapted to represent the structural information of subtrees and introduce our novel root-split coding; the root-split coding leads to a more concise index, which further reduces the response time of queries as well as index construction time.

5.1 Subtree Index

Given a set of syntactically annotated trees S and a size mss, consider the set of all unique subtrees of sizes $1, 2, \ldots, mss$ that can be extracted from trees in S, and associate to each subtree a posting list consisting of the ids of trees in S where the subtree appears. We want to organize the pairs of subtrees and posting lists in an index, referred to as *Subtree Index* (or *SI* for short), such that our queries can benefit from this structuring.

There has been work on indexing nodes, edges and paths in XML documents (as reviewed in Section 3.2.2), and similar approaches have been used over syntactically annotated trees (e.g. LPath [9]). Since the subtree index stores all unique subtrees of sizes $1, 2, \ldots, mss$, it generalizes node- and edge-based indexes. In the scenario where $mss = 1$, the index only stores information about individual nodes, and this is very similar to the relational *Node* approach. For Larger values of mss, a subtree index can offer the following benefits: (a) Reduced number of joins compared to relational *Node* and *Edge* approaches, by pre-materializing subtrees of larger sizes. (b) Better preserving the structure of trees which can lead to better query response time. Bird et al. [9] show that node approach outperforms TGrep2 [101] (which uses sequential scan) in terms of the querying performance and scalability. In this thesis, we show that SI with larger mss values perform better than the case with $mss = 1$ (or the node approach). We also provide results on the performance and the scalability of SI over a baseline that uses indexing to filter candidate matches and tree scans to find the exact set of matches. This baseline, which we call filter-based coding, is described in Section 5.1.4.

In the rest of this chapter we discuss some of the challenges involved in using SI for querying syntactically annotated trees. Specifically, we study the following problems over subtree indexes. (1) Coding structural information of a tree into an inverted list of subtrees, where we adapt previous approaches such as navigational and interval coding in order to come up with two baseline approaches, referred to as filter-based coding and subtree interval coding. (2) Efficiently decomposing query trees into smaller subtrees, for which we study the properties of a "good" cover over a query tree. (3) Query matching over SI for which we study both the scenarios where matching requires injective mapping and when it does not (See Section 2.2.2 for a discussion on matchings with injective mapping functions).

5.1.1 Subtree Indexes over Syntactically Parsed Trees

In this section, we study the properties of syntactically annotated trees, which make subtree indexes practical and scalable over them. Subtree Indexes can improve the performance of querying over natural language queries by (1) pre-materializing partial subtree solutions, thus reducing the number of joins required for evaluating a tree query, and (2) reducing sizes of posting lists in most cases[1] by grouping together smaller subtrees and building larger subtrees.

One drawback of subtree indexes is that their size could potentially grow dramatically as mss increases. Two factors that affect the size of a subtree index are (1) the number of unique subtrees or index keys, and (2) the total number of extracted subtrees. This latter number gives an upper bound on the total number of postings[2]. Next we discuss some of the issues that play a role in the growth of these two factors and a discussion of why building SI can be practical and scalable for syntactically annotated trees.

[1]In Section 5.2 we discuss that sizes of posting lists might not be monotonically reduced for larger subtrees if the index uses subtree interval coding

[2]Number of postings could be smaller for the scenarios where subtree interval coding is not used

Number of Index Keys

One nice property of a subtree index is that the number of index keys (unique subtrees) grows linearly with the size of the input, for different values of mss. As a result, the body of the index does not grow dramatically as more data is being indexed, regardless of the value of mss. One reason for this is that similar structures are abundant throughout the corpus of parsed trees. This is based on the observation that there is only a finite and relatively small set of grammatical structures used in natural languages, and the number of such unique structures does not grow dramatically even considering differences in writing styles and parsing errors.

Figure 5.1 shows the number of unique subtrees as a function of the input size, for different values of mss, over collections of parse trees containing 1 to 10^6 sentences from a news corpus. The figure shows approximately the same rate of growth in the number of keys, for different values of mss. Moreover, the number of index keys grows almost linearly with the size of the indexed data.

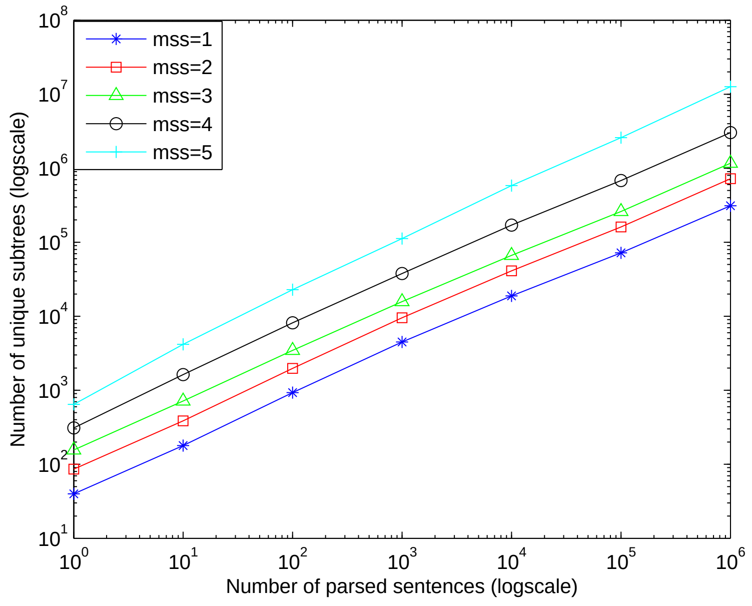


Figure 5.1: Number of index keys (unique subtrees) as a function of the input size in terms of the number of sentences

Number of Extracted Subtrees

For a parse tree of size n, the number of subtrees of size $1 \leq m \leq mss$ could range from $n - m + 1$ to $\binom{n-1}{m-1}$. The former belongs to the case where the tree is a unary branch of height n, and the latter demonstrates the case where the parse tree is of height 2 and consists of a root with $n - 1$ leaf children. Note that the number of subtrees of sizes $1, \ldots, mss$ of a tree gives an upper bound on the number of postings stored for it in the index. Therefore, for large values of mss and n, the number of postings stored in the index could be very large, resulting in a huge index. As we show later, the number of subtrees is in practice orders of magnitude smaller than the worst case scenario, making it possible to build SI for small values of m (e.g. $1 \leq m \leq 5$).

To investigate how the number of extracted subtrees changes over syntactically annotated trees, we conducted an experiment on more than $50,000$ nodes from a (constituency) parsed corpus of news. Over each node, we extracted every possible subtree of sizes 2 to 5, and counted the number of such subtrees. Figure 5.2 depicts how the number of subtrees changes with the branching factor of the nodes, for this dataset. In this figure, the x-axis displays the branching factor of nodes, and the y-axis shows the average number of subtrees extracted from nodes with the given branching factor. As displayed in this figure, nodes with higher branching factors, lead to a larger number of subtrees, on average. We also present the non-aggregated results for the same set of nodes in Figure A.2 in Appendix A, which displays the number of distinct subtrees in terms of the branching factor of the node over which such subtrees are constructed. Motivated by these two figures, in the rest of this section, we elaborate on some of the important characteristics of syntactically annotated trees that distinguishes them from other data types modelled as trees.

Rare Nodes with Large Branching Factors. In syntactically annotated trees, we expect to see a few nodes with relatively high branching factors. In what follows, we provide some supporting experimental results and a discussion of why such nodes are expected to be rare in English. In Figure A.2 we see only two nodes that have a branching factor larger than 10. For these two nodes, the branching factors of 19 and 23 are still small compared to XML documents, which could have branching factors of a few hundreds or even larger. Reasons for such a characteristic in syntactically annotated trees could be the following.

(i) Parse trees are relatively small trees. The number of nodes in a dependency tree and the number of leaves in a constituency tree are equal to the number of words in the underlying parsed sentence. As a result, the total number of nodes in a parse tree is in the range of tens to at most a few hundreds of nodes.

(ii) High branching factor nodes in parse trees are due to repetitive structures. In well-written and clean natural language corpora, such repetitive structures are rarely too

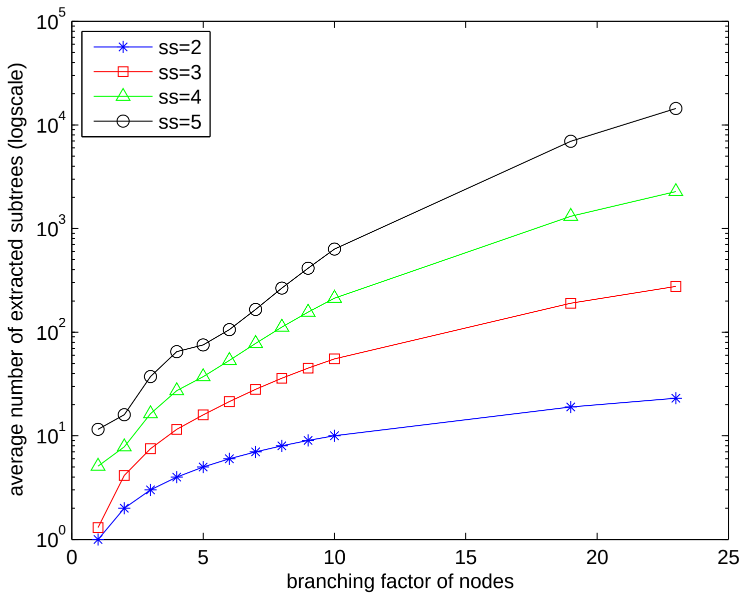


Figure 5.2: Average number of subtrees extracted in terms of the branching factor of roots of subtrees

long, as they can create difficult sentences to read and understand.

Example 5.1.1. Examples of repetitive structures in syntactically annotated trees can be (a) `(NN)* VB` as in a parse of `Tom, Sarah, Alex and Mary attended the tea party` and (b) a structure like `DT (JJ)* NN` as in `It is an amazing warm sunny day`. Parse trees of these examples, parsed using the Stanford parser, are depicted in Appendix A in Figure A.1. As this figure shows, the above examples lead to parse trees with maximum branching factors of 7 and 5, respectively.

Small Average Branching Factor. Syntactically annotated trees have a small average branching factor. The total average branching factor for the above dataset is 0.98. The average branching factor for internal nodes only is 1.52. Thus, on average, each internal node has less than two children, which makes syntactically annotated trees very suitable for indexing.

In the next section, we provide details on constructing subtree indexes including our subtree extraction algorithm and a few tweaks that can help reducing the index size.

5.1.2 SI construction

A subtree index is parameterized by mss, the maximum subtree size. Given mss and a set of parsed trees, SI is constructed by extracting all unique subtrees, then flattening and encoding them in the index.

Subtree Extraction

For each input tree t, the algorithm in Figure 5.3 extracts all unique subtrees of size $1, \ldots, mss$. In this algorithm, $|t|$ is the size of the subtree rooted at t in terms of the number of nodes. The process starts at the root of the input tree t and recursively descend into its descendant in a pre-order traversal, and for each node t calls $subtrees(t, i)$, where i ranges between 1 and $\min(|t|, mss)$. The call to $subtrees(t, i)$ computes and returns every possible subtree of size i rooted at t. This algorithm is depicted in Figure C.1 in Appendix C. As an example of how *extract* algorithm works, Figure 5.4(b) depicts the set of all unique subtrees of size 3 originating from the root of the tree in Figure 5.4(a).

```
$\mathbf{extract}(\mathbf{t})$
1  $res \leftarrow \emptyset, m = \min(|t|, mss)$
2  **for** $i \in \{1, \ldots, m\}$
3      $res \leftarrow res \cup \mathbf{subtrees}(\mathbf{t}, \mathbf{i})$
4  **for** $c \in t.children$
5      $res \leftarrow res \cup \mathbf{extract}(\mathbf{c})$
6  **return** $res$
```

Figure 5.3: Algorithm for extracting all unique subtrees of sizes 1 to mss from a tree t

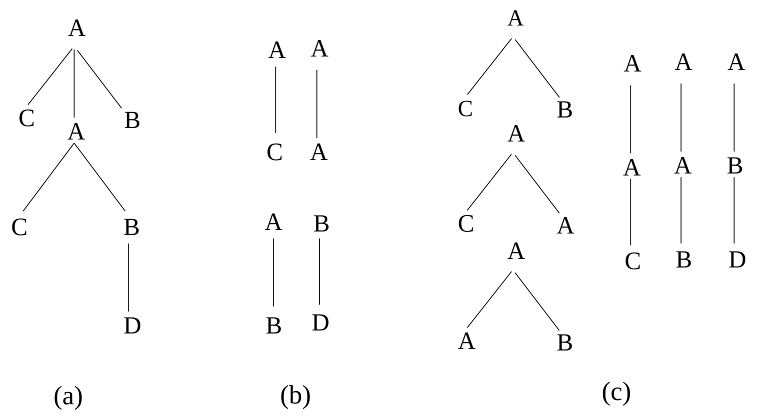


Figure 5.4: An example of how unique subtrees are extracted, (a) input tree, (b) unique subtrees of size 2, (c) unique subtrees of size 3

Flattening and Encoding

In order to store the unique subtrees as keys into the index, they have to be flattened. We do the flattening by traversing each subtree in a pre-order traversal and for each node capturing its label and size. Given a pre-order traversal, the original subtree can be re-constructed. Alternatively, a well known approach is to use a DFS traversal together with delimiters to convert the structure of a tree into a flat representation, as in [122]. With this flattening, the tree in Figure 5.4(a) could be flattened as `(A(C)(A(C)(B(D)))(B))` or more concisely as `AC)A)BD)))B))`. To further reduce the key size and tune coding to parse trees which are generally small in size, we choose to store the size of nodes instead of delimiters. A tree can be encoded concisely by exactly $mss(\lceil\log_2(mss+1)\rceil + \lceil\log_2|\Sigma|\rceil)$ bits, where Σ is the alphabet of node labels. In this formula, the first term is the number of bits required to encode the size of each node and the second term is the number of bits required to encode the label. Notice that sizes of nodes are orders of magnitudes smaller than size of the alphabet for index keys and for that reason, encoding using node sizes requires much less space than using delimiters.

5.1.3 Query Matching Over Subtree Indexes

Query matching over a subtree index happens in two phases, (1) the *split phase* in which the queries are shredded into smaller subtrees, where each subtree size is at most mss, and the posting lists of subtrees are fetched from the index, and (2) the *join phase* in which result set of subtrees are joined to evaluate the final results. In this section we provide a brief overview of these two phases for our proposed coding schemes. In Section 5.2 we provide a thorough study of the first query matching phase, i.e. *split phase*. Section 5.3 provides an in depth study of the *join phase* and includes a discussion of query matching in the scenario when the matching is injective.

Definition 5.1.2. For two trees T and T', we say that T is a subtree of T' and denote it by $T \precsim T'$ if and only if (1) $V(T) \subseteq V(T')$ and (2) $E(T) \subseteq E(T')$.

In the above definition, $V(T)$ and $E(T)$ are the set of nodes and set of edges of tree T, respectively.

Definition 5.1.3. A set $C = \{c_1, \ldots, c_k\}$ of trees is a *node-cover* of tree T, if and only if (1) for all $c_i \in C$, we have $c_i \precsim T$ and (2) for all $v \in V(T)$ there exists at least one $c_i \in C$ such that $v \in V(c_i)$.

Intuitively, a *node-cover* of a tree T is a set of subtrees of T such that every node of T appears on at least one of the subtrees of the node-cover.

Definition 5.1.4. A set $C = \{c_1, \ldots, c_k\}$ of trees is a *full-cover* of T, if and only if (1) C is a *node-cover* of T, and (2) for all $e \in E(T)$ there exists at least one $c_i \in C$ such that $e \in E(c_i)$

According to the above definition, a *full-cover* C of tree T, covers both nodes and edges of T. Hereafter in the thesis, we refer to both *full-covers* and *node-covers* simply as *covers*, when the meaning is clear from the context.

Definition 5.1.5. Given a query Q and a parameter mss, $C = \{c_1, \ldots, c_k\}$ is a valid cover of Q with respect to mss if and only if there does not exist a subtree $c_i \in C$ such that $|c_i| > mss$, for all $1 \leq i \leq k$.

In the rest of the thesis, we assume that all covers are valid, unless otherwise noted.

Our goal in the *split phase* is to find a "good" cover for a given query. A "good" cover can be informally defined in terms of its closeness to a cover that results in the least query execution cost. A query could have a large number of covers, and the choice of which cover to pick can have a large effect on the query evaluation cost. While query optimization is not the topic of interest in this thesis, in Section 5.2 we study a few properties of covers over coding schemes that help us prune the search space for "good" covers.

In the *join phase*, we have a cover for a query and we want to intersect the posting lists of the subtrees in the cover to find the set of matching parse trees. Next we discuss our coding schemes over subtree indexes.

5.1.4 Coding Schemes

In this section we propose three coding schemes for describing the structural information of subtrees stored as keys in a subtree index. The first two coding schemes are adaptations of current coding schemes over text or xml documents; they are mainly used as baseline methods. The third coding scheme, root-split coding, is a novel approach we propose to store the structural information of subtrees more concisely.

Filter-Based Coding

The filter-based coding is a minimal coding scheme which does not store any structural information about the keys being indexed. Similar to any inverted index structure, the filter-based coding stores a sorted list of unique tree identifiers, tids, of the trees that contain the indexed subtrees.

A query matching for the filter-based coding starts by finding a query cover and fetching the posting lists of the subtrees in the cover. The join phase of the query matching includes the pairwise intersection of the sorted lists of tids to obtain the list of candidate tids. Unlike the other two coding schemes, Query matching for the filter-based coding has a third phase, called the *filtering phase*. In the filtering phase, the parse trees corresponding to candidate tids are all fetched and checked if they actually match. Note that since the structural information of the subtrees are not stored, exact matching over the posting lists of the subtree index is not possible and a final (usually costly) filtering phase is required to find the set of matching trees. Our *filtering phase* in the case of filter-based coding only checks if a tree matches a query, and it does not report all query matches but only the first match. This is unlike our next two coding schemes for which the matching algorithm finds all matches.

Subtree Interval Coding

As noted in Section 2.2.4, the node interval coding stores for each node a pair of *left* and *right* values to handle reachability queries and a *level* value to answer parent-child axes queries. A subtree interval coding generalizes the node interval coding and stores in addition the structural information of individual nodes in an indexed subtree.

As for the structural information, we keep the order in which each node is visited in the DFS traversal used for flattening the subtrees. Thus keeping an *order* value for each subtree node is required because we assume that the indexed subtrees are not ordered, and two subtrees such as `A(B)(C)` and `A(C)(B)` are indexed under the same entry. For instance, assume that both subtrees are represented as `A(B)(C)`. In such a scenario, postings which represent `A(C)(B)` subtrees, have order equal to 0, 2 and 1, for A, C and B, respectively. As a result, a query such as `A(C(D)(E))(B)`, requiring an equality join of `A(C)(B)` and `C(D)(E)` on the C node, can be correctly evaluated, by selecting the second set of numbers when C's order is 2 and the third set of numbers when C's order is 1.

The structure of a posting describing a subtree of size m is therefore as follows

$$\{tid, m, < l_1, r_1, v_1, o_1 >, \ldots, < l_m, r_m, v_m, o_m >\}$$

where tid is an identifier of the tree that contain the subtree and $< l_i, r_i, v_i, o_i >$ values are the left, right, level and order numbers, respectively.

The query matching for a query Q is performed by computing a cover of Q, fetching its posting lists and joining them. In Section 5.2 we discuss how to compute an efficient query cover over a subtree index with subtree interval coding. In Section 5.3, we discuss in more detail the matching algorithms over SI with subtree interval coding.

Root-Split Interval Coding

The idea behind root-split (interval) coding is to avoid storing unnecessary structural information and to represent each subtree as concise as possible. Root-split coding stores for each subtree only the tree identifier and *left*, *right* and *level* values of its root. Compared to subtree interval coding, root-split coding reduces the size of each posting by a factor larger than m, where m is the size of the subtree being indexed.

Similar to the previous two coding methods, query matching over root-split coding also consists of a *split* phase and a *join* phase. Note that since the structural information of individual nodes are not stored in a root-split coding, the queries cannot be arbitrarily split and joined. In the following, we define the types of covers required over root-split coding.

Definition 5.1.6. Given a query Q, $C = \{c_1, \ldots, c_k\}$ is a *root-split* cover of Q if and only if either $C = \{Q\}$ or for every subtree c_i, there exists a subtree c_j, $1 \leq i, j \leq k$, such that one of the following hold: (1) both c_i and c_j are rooted at the same node in Q, (2) c_i is rooted at the parent of c_j in Q, or (3) c_j is rooted at the parent of c_i in Q.

Intuitively, a root-split cover is a cover which can be evaluated only by performing joins over the roots of its subtrees. Such a cover would be useful for our root-split coding as we only store structural information over roots of index keys.

Lemma 5.1.7. *Every query Q has at least one root-split cover.*

Proof. The proof can be achieved by simply building a cover C as a set containing individual nodes of Q. C is a (valid) root-split cover and the lemma is proved. □

In Figure 5.5, we present a naive algorithm for generating a root-split cover, which is more practical than a cover of all query nodes and better exploits the power of subtree indexes and creates a foundation for more efficient algorithms that follow in the next section.

Lemma 5.1.8. *The algorithm in Figure 5.5 computes a root-split cover.*

Proof. Proof is by induction.
Base. For $|Q| \leq mss$, $naiveRC$ adds Q to the cover and returns. A cover that contains only Q is root-split, thus the base case holds.
Induction. Assume $naiveRC$ generates root-split covers for individual children of Q by calls in line 6 of the algorithm. We would like to prove that the final cover, which is

```
naiveRC(Q)
1  $C \leftarrow \emptyset$
2  if $|Q| \leq mss$ then
3     $C \leftarrow C \cup Q$, return $C$
4  pick any subtree from $\mathbf{subtrees(Q, mss)}$, add it to $C$
5  for $c \in Q.children$
6     $C \leftarrow C \cup \mathbf{naiveRC(c)}$
7  return $C$
```

Figure 5.5: A naive algorithm that guarantees a root-split cover

the union of all such root-split covers, plus the subtree s_Q generated at a call to line 4 of algorithm is root-split ($subtrees$ algorithm is covered in Figure C.1 in Appendix C). By definition of a root-split cover, s_Q is rooted at the parent of the subtrees covering its children and therefore, the final cover is root-split. Thus, lemma is proved. □

In the next Section, we analyze root-split coding and show that it can be used for both correctly and efficiently evaluating queries. We compare the three proposed coding schemes in terms of their performance analytically.

5.2 Query Splitting Strategies

In this section, we study the theoretical properties of the root-split coding and compare it in terms of applicability and optimality with the subtree interval and filter-based codings. As will be discussed in this section, root-split coding reduces the size of posting lists, by reducing the number of postings and the size of each posting. As a result, the size of SI with root-split coding is smaller than its corresponding SI using subtree interval coding, by a large factor.

5.2.1 Monotonicity of Posting List Sizes

In the context of relational query optimization, intersection of the posting lists of subtrees indexed in SI, maps to select-project-join queries, with selections using index scan and joins using merge joins over sorted data streams (posting lists). In such a context a query optimizer over a subtree index, often generates query execution plans in the form of left-deep (or right-deep) trees resulting in a linear order of joins. Given a query Q, an efficient query plan can be obtained by (1) picking a “good” cover of Q whose subtrees serve as data streams over leaves of the query plan, and (2) searching the space of available plans for the selected cover and finding an efficient or optimal query execution plan. The second step is the task of a query optimizer and we do not study it in this thesis. However, in this section we study what properties of a cover make it more amenable for query optimization.

The first property we study is how the size of the posting lists change for subtrees of different sizes, for our proposed coding schemes.

Lemma 5.2.1. *For any two index keys s_1 and s_2 over a given SI, where $s_1 \precsim s_2$, we have*

(i) The posting list of s_2 is always a subset of the posting list of s_1 for filter-based coding.

(ii) The posting list of s_2 is a subset of the posting list of s_1 for root-split coding if and only if s_1 and s_2 share the same root.

(iii) The posting list of s_2 is not guaranteed to be a subset of the posting list of s_1 for subtree interval coding.

Proof. Appears in the Appendix B. □

Lemma 5.2.2. *For any two index keys s_1 and s_2 of a SI with root-split coding, where $s_1 \precsim s_2$ and s_1's root has a different label from s_2's root, then for each posting in the posting list of s_1 there is at most one posting in the posting list of s_2 associated with it.*

Proof. Given the conditions of this lemma, s_1 must be a descendant of s_2's root. Since ancestor-descendant relationship is a one to many relationship, there must be only one posting in the posting list of s_2 for any number of its descendants, hence the lemma is proved. □

The direct conclusion from Lemmata 5.2.1 and 5.2.2 is that the size of the posting lists monotonically decrease as subtrees grow for filter-based and root-split codings, while we do not have such a guarantee for the subtree interval coding. This is a very useful property and we will discuss some of the interesting conclusions that this monotonicity will provide.

Theorem 5.2.3. *Given a query Q and a subtree index with root-split coding and maximum subtree size mss, an optimal query plan for Q cannot have a subtree of size less than mss.*

Proof. Appears in the Appendix B. □

The above theorem is true for filter-based coding as well, but not for subtree interval coding. Therefore, we can conclude that the search space of query optimizers for the optimal query execution plan using filter-based and root-split codings are smaller compared to subtree interval coding, which is an advantage. Finding optimal query execution plan requires building histograms of subtree selectivities or estimating the selectivities of subtrees (See [22] for an example). However, the focus of this thesis is not on addressing the problem of query plan optimization over subtree indexes, therefore we assume any cover in which all subtrees have size mss would be a good enough cover for our query execution task.

5.2.2 Join Optimality

As discussed earlier, root-split coding constrains query splitting to covers in which subtrees can be joined over their roots only. In this Section, we investigate the ramifications of such a constraint on the size of the root-split covers. We study the number of joins required to evaluate a cover as a measure of its efficiency. Moreover, we study the problem of join optimality for root-split and non-root split covers.

Max Covers

We showed earlier by Lemma 5.1.7 that for every query Q there exists at least one root-split cover C. Depending on how Q is structured, C might have subtrees ranging in size from 1 to mss. One interesting problem is to investigate if there exists an algorithm that can always generate a root-split cover, where the size of every split is equal to mss. We call such a cover a max-cover. According to the discussion in the previous section, such a cover would achieve an efficient query evaluation plan.

Theorem 5.2.4. *For every query Q and size mss such that $|Q| \geq mss$, there exists a root-split max-cover C; i.e. for every subtree $c \in C$ we have $|c| = mss$.*

Proof. Appears in the Appendix B. □

Among all max-covers of Q, only a few are root-split, and among such max-covers, those with the smallest size, in terms of the number of subtrees, are desirable as they lead to our definition of a join-optimal cover.

Definition 5.2.5. For a given query Q, a join-optimal cover of Q, is a max-cover over Q that has the smallest size in terms of the number of subtrees among all covers of Q.

Note that for any cover C, there exists a max-cover which has size smaller than or equal to C. As a result, we do not need to worry about non max-covers that might be join-optimal. Selecting covers among max-covers is also desirable for filter-based coding, but not necessarily for subtree interval coding.

According to Lemma 5.2.1, it is not always desirable to select covers over a subtree interval coding from max-covers. As that lemma shows, larger subtrees in the covers do not necessarily lead to smaller posting lists, when subtree interval coding is used. To study the effects of the non-monotonicity of the posting list sizes on the performance of subtree interval coding if max-covers are always selected, we conducted an experiment on a sample dataset of around 1000 sentences from a news corpus. The dataset contained more than 112 thousand index keys of sizes 1 to 5. Over this dataset, we exhaustively compared every key k_i with every other key k_j and counted the cases where k_i is a subtree of k_j, but the size of the posting list of k_j is larger than that of k_i. Out of more than 6 billion pair-wise comparisons made, only 26495 cases met the condition, and this accounts to less than 0.0005% of such comparisons. This confirms that max-cover may be a good heuristic for evaluating queries over the subtree interval coding scheme. In the rest of this thesis and for all of our coding schemes, we only consider max-covers, ignoring any non-max covers. In the rest of this section, we study join-optimality over max-covers (hereafter referred to as simply as covers).

Join Optimal Covers

In this section, we study the problem of finding join optimal covers, in the scenario where matchings are not required to be injective. Query matching under the injective matching assumption is discussed in Section 5.3.

Definition 5.2.6. Given a query Q, and a cover C over it, we say that C has *deep branching* anomaly, if there exist subtrees s_i and s_j in C such that (1) s_i and s_j share at least one node of Q, say $v \in V(Q)$, such that v is not root of s_i, and v is not root of s_j, and (2) v has at least two children u and u', such that $u \in V(s_i)$, $u \notin V(s_j)$ and $u' \in V(s_j)$ and $u' \notin V(s_i)$.

Deep branching anomaly, as defined in Definition 5.2.6, describes a situation where two subtrees in a given cover cannot represent the structure of part of the query they cover, uniquely. Deep branching anomaly can result in an incorrect set of matches for root-split codings, due to extra matches. As a result of a deep branching anomaly, extra subtrees might be required to be added to the root-split covers to fix this anomaly.

For non-root-split codings, deep branching anomaly can be dealt with, efficiently. Such an anomaly does not cause any problems for filter-based codings as the final set of matches is obtained by scanning over the candidate parse trees. For subtree interval coding this anomaly can be dealt with by joining on the deepest shared branching node of the two subtrees. In the example that follows, we demonstrate how it is possible to handle deep branching for subtree interval coding.

Example 5.2.7. Consider the query in Figure 5.6.(a) and let $mss = 4$. A join-optimal root-split cover of the query is C_1={`A(B(C(D)))`, `B(C(E)(F))`} Figure 5.6.(b) shows multiple tree structures that match the given root-split cover. The result set obtained by an anomalous join over roots of the subtrees of C_1, i.e. `A` and `B` nodes, thus might result in extra incorrect matches. For subtree interval coding, this situation can be dealt with by an equality join on the deepest branching node, i.e. `C`. By sacrificing join optimality, we can obtain root-split covers that do not have the deep branching anomaly, as in the following cover C_2={`A(B(C(D)))`, `B(C(E)(F))`, `C(D)(E)(F)`}.

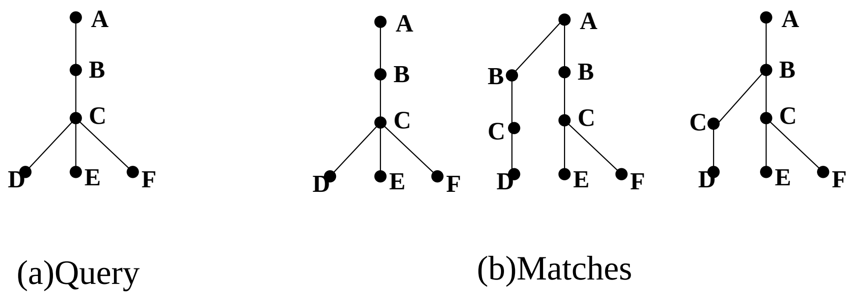


Figure 5.6: Example of a query having deep branching anomaly for $mss = 4$ for a root-split join optimal cover

Figure 5.7 displays every possible query structure of size 5, and the minimum number of root-split joins required for different values of mss. In this figure, the first column displays subtree structures and columns $2, 3$ and 4 display the number of joins required for evaluating the best possible root-split cover when mss is equal to $2, 3$ and 4, respectively. The values in bold display the cases where the best possible root-split joins do not achieve join optimality. In such cases, the number of extra joins required is indicated in front of the number of joins in brackets; e.g. (+1).

Note that in most cases in Figure 5.7, there exists a join optimal root-split cover for the given query structures. From a total of 14 tree structures listed, 13 have a join optimal root-split cover when $mss = 2$, and 9 have a join optimal root-split cover when $mss = 3$. The number of extra joins required in each case for non-optimal covers are only 1, however, there could be cases where more extra joins are required.

Proposition 5.2.8. *The number of extra joins required for evaluating a root-split cover of a query Q is at most $|Q| - \lceil \frac{|Q|}{mss} \rceil - mss + 1$, compared to a join optimal cover.*

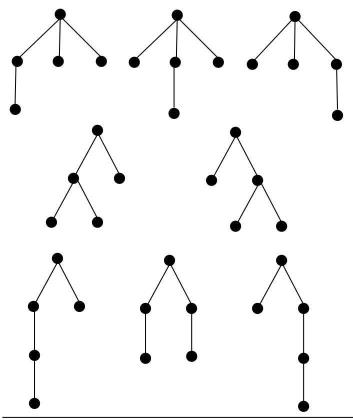
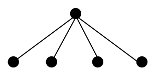
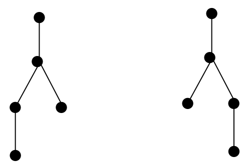
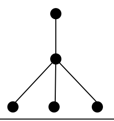
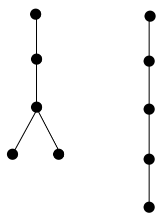
mss	2	3	4
	2	1	1
	3	1	1
	2	**2(+1)**	1
	3	**2(+1)**	1
	**3(+1)**	**2(+1)**	1

Figure 5.7: Join optimality on all possible queries of size 5 with mss values of $2, 3$ and 4 from left to right columns.

Proof. The worst case happens when the tree is structured as a unary branch of height $|Q|$. In this case, the number of subtrees for a root-split cover is given by $|Q| - mss + 1$, while the number of subtrees in a join optimal cover is given by $\lceil \frac{|Q|}{mss} \rceil$. The difference of these two terms gives our proposition bound. $\square$

The above proposition provides an untight upper bound on the number of extra joins required to evaluate a root-split cover. In practice however, as we will show in Section 6.2.3, the actual number of extra joins is much smaller than this bound. In the rest of this section, we provide algorithms for computing covers for both root-split and non-root-split codings.

Our algorithm $optimalCover$, as shown in Figure 5.8, generates for each query a join optimal cover, with the size of every covered subtree equal to mss. The algorithm starts with an empty cover C, and in each step either adds a subtree of size mss or calls $optimalCover$ on larger subtrees.

Thus, at the base of the recursion, $optimalCover$ handles only children of Q having size less than or equal to mss. Any child of Q with size equal to mss is added immediately to the cover C. Children with sizes smaller than mss are handled by calling $assign$ until the total number of unassigned nodes in the subtree of Q and including Q is less than mss. At this point if Q is not the root of the original query, Q and its unassigned nodes can be part of a subtree originating from parent of Q, and thus the $optimalCover$. Otherwise, if Q is the root of the original pattern, all that is left to do is to cover the last set of unassigned nodes, whose number is less than mss. This is achieved by one last call to $assign$ in lines $9 - 10$. The algorithm $assign$ is also presented in the same figure. Intuitively, a call of $assign(t)$ computes a subtree of size mss, rooted at t, which has the most possible set of unassigned nodes. It starts by picking larger unassigned children of t and once it runs out of unassigned nodes, adds assigned nodes until the size of subtree is mss.

Example 5.2.9. Consider the tree shown in Figure 1.2.(a) and suppose we run the algorithm $optimalCover$ on this tree with $mss = 3$. The first child of `S` is `NP(NNS(agouti))` of size 3 and this child is added to C immediately. The second child of `S`, `VP`, is of size 7, so $optimalCover($`VP`$)$ is called, which in turn calls $optimalCover$ on `NP` of size 4. Since `DT(a)` and `NN` both have size less than mss, $assign($`NP`$)$ is called; the call returns `NP(DT(a))` which is added to C and sets $|NP| = 2$. Since `NP` is not a root (line 9 of $optimalCover$), C is returned to the caller. The next steps of the algorithm will add `VP(VBZ(is))`, `VP(NP(NN))` and `S(NP(NNS))` to the cover. Note that a join of subtrees `VP(NP(NN))` and `NP(DT(a))` must be in the form of an equality join on node `NP`, to avoid erroneous results due to deep branching anomaly.

Lemma 5.2.10. *Given a parameter $mss \leq 6$ and a tree t, where $|t| > mss$ and all children of t have size less than mss, repeated calls of $assign$ over $r(t)$ partitions t into a join optimal cover.*

Proof. Since children of t all have size less than mss, any subtree that covers them have to be rooted at $r(t)$. Thus, the partitioning problem reduces to the integer bin packing problem, where the bin capacity is $mss - 1$ and children sizes are the volumes of the items to be stored. The objective is to minimize the number of bins (subtrees in our problem). Our $assign$ algorithm sorts children in a non-increasing order of their sizes, which maps to the *fit first decreasing (FFD)* approximation algorithm for bin packing. FFD in general

```
optimalCover(Q)
1  $C \leftarrow \emptyset$
2  for $c \in Q.children$
3    if $|c| = mss$
4      $C \leftarrow C \cup c$, $|Q| = |Q| - |c|$, $c.assigned = true$
5    else if $|c| > mss$
6      $C \leftarrow C \cup$ optimalCover(c)
7  while $|Q| \geq mss$
8    $C \leftarrow C \cup$ assign(Q)
9  if $|Q| > 0$ and $isRoot(Q)$
10   $C \leftarrow C \cup$ assign(Q)
11 return $C$
```

```
assign(Q)
1  $cnt = 1$, $t.root = Q.root$, $Q.assigned = true$
2  sort $Q.children$ on size, descending
3  for $c \in Q.children$
4    if $c.assigned = false$
5      if $(mss - cnt - |c|) \geq 0$
6        $c.assigned = true$, $t.children \leftarrow t.children \cup c$
7        $|Q| = |Q| - |c|$, $cnt = cnt + |c|$
8    if $cnt = mss$ then return $t$
9  if $cnt < mss$
10   for $c \in (Q.children - t.children)$
11     if $(mss - cnt - |c|) \geq 0$ then
12       $t.children \leftarrow t.children \cup c$, $cnt = cnt + |c|$
13     else
14       add any subtree from subtrees(c, mss − cnt) to $t.children$
15 return $t$
```

Figure 5.8: Algorithm that computes a join optimal cover of size mss

gives approximation ratio of $\frac{11}{9}OPT + 1$ [119] and is shown to be optimal for integer bin packing with bin sizes less than or equal to 6, which proves our lemma. □

The above lemma proves that for small values of mss, assign provides an optimal partitioning and for general mss, it achieves a good approximation ratio. As discussed earlier, the number of extracted subtrees could grow dramatically as mss increases and therefore, in practice we will not be dealing with mss values larger than 6. In our experiments, we limited mss to be at most 5.

Theorem 5.2.11. *optimalCover returns a join optimal cover if (1) $mss \leq 6$ and (2) injective matching is not assumed.*

Proof. Appears in Appendix B. □

Through some modifications of the $optimalCover$ algorithm, we can develop an algorithm that obtains the smallest root-split cover in terms of size. This algorithm, referred to as $minRC$, is presented in Figure 5.9. This new algorithm takes a bottom-up approach and descends into subtrees of smaller sizes until children have size less than or equal to mss. Then, it covers the given subtree entirely, before moving up to higher levels. This guarantees that every child of a given node v is covered, before v is covered and as a result deep branching anomaly cannot occur.

```
$\mathbf{minRC}(\mathbf{Q})$
1  $C \leftarrow \emptyset$
2  for $c \in Q.children$
3     if $|c| = mss$
4        $C \leftarrow C \cup c$, $|Q| = |Q| - |c|$, $c.assigned = true$
5     else if $|c| > mss$
6        $C \leftarrow C \cup \mathbf{minRC}(\mathbf{c})$
7  while $|Q| \geq 0$
8     $C \leftarrow C \cup \mathbf{assign}(\mathbf{Q})$
9  return $C$
```

Figure 5.9: Algorithm that computes the best root-split cover of size mss

Example 5.2.12. The $minRC$ algorithm generates the following cover over the query in Figure 1.2.(a). $C = \{$`NP(NNS(agouti)), NP(DT(a)), NP(DT)(NN), VP(VBZ(is)), S(NP(NNS))`$\}$. The subtree ordering shown is the same as the order by which $minRC$ adds subtrees to C. C is join optimal, and it has the same number of subtrees as an optimal cover, given in Example 5.2.9.

Theorem 5.2.13. *$minRC$ returns the smallest root-split cover possible if (1) $mss \leq 6$ and (2) injective matching is not assumed.*

Proof. Appears in Appendix B. □

5.3 Join Approaches over SI

In this section we study some of the problems associated with joining structural information stored under the root-split and the subtree interval codings. As for the filter-based coding, the problem does not arise since no structural information is stored and the filtering is done through scanning candidate parse trees.

5.3.1 Joins

Join Types

Given two subtrees in a cover C of a query Q, we often want to join them to obtain our matches for Q. Two major types of joins are possible, and these are equality joins and structural joins. Structural joins were briefly discussed in Sections 2.2.4 and 3.2.2. A large number of structural join algorithms have been proposed in the literature. In this thesis we use the MPMGJN algorithm [120] for its easy adaptation to our problem.

An equality join happens when two subtrees share a node of Q and are joined on the equality condition over that node. As a result, only equal postings in the corresponding posting lists are returned. Equality joins can be performed very efficiently, as all the matches can be obtained by a linear scan of the sorted posting lists of the corresponding subtrees. Thus, if two subtrees can be joined in several ways, as might occur in subtree interval coding, equality joins are preferred.

Join Selection

Any two subtrees in a root-split cover have only one way to be joined, and that is a join over their roots. However, subtree interval codings might have multiple ways to be joined. Given two subtrees s_i and s_j of a query Q, the order of priority for selecting joins over subtree interval covers is as follows.

(i) If s_i and s_j suffer from deep branching anomaly, select the equality join on the deepest shared node of Q.

(ii) If there are any shared nodes between s_i and s_j, select the equality join on the shared node.

(3) Select a structural join over nodes of s_i and s_j.

5.3.2 Injective Matching

As discussed in Chapters 2 and 3, it is often desirable for queries over syntactically annotated trees to have an injective matching. Our next example shows some of the problems that can arise when a matching is not injective.

Example 5.3.1. Figure 5.10(a) shows a query that looks for the set of noun phrases which have one `DT` child, two `JJ` children and three `NN` children. Given the subtrees from a join optimal cover shown in Figure 5.10(b), the matching can lead to some false matches, one of which is depicted in Figure 5.10(c). Therefore, for certain queries, the result set using an injective matching and a non-injective matching would be different.

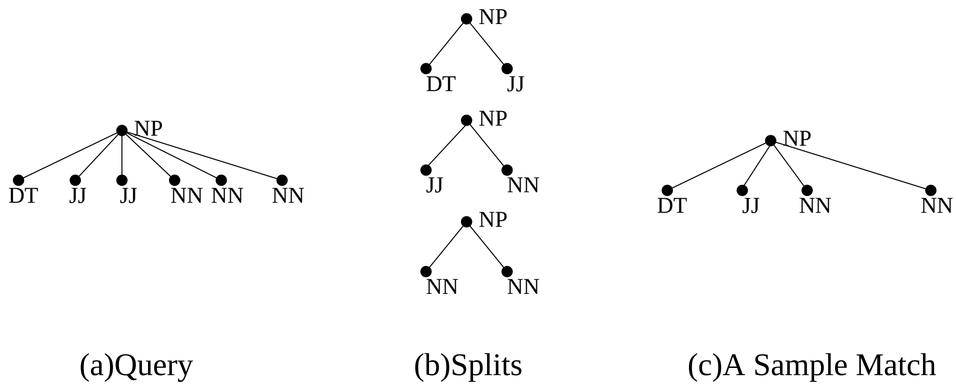


Figure 5.10: Example of a query and its corresponding cover that can lead to false positive matches

The query in Figure 5.10(a) is one example where extra effort needs to be made in order for the matching to be injective. We want to find more general cases where our regular splitting and query matching algorithms discussed so far might fail in providing a guarantee on the injectivity of the matching.

Definition 5.3.2. Given a query Q, if there exists subtrees s_B and s_P of Q such that (1) s_B and s_P share the same parent, (2) s_B and s_P have the same root label, and (3) s_B is a subtree of s_P, then we say that s_P *hides* s_B, as every match for s_B is included in s_P.

Theorem 5.3.3. *Given a query Q, if there exist subtrees s_B and s_P of Q such that s_P hides s_B, then there exist covers over Q which do not guarantee a correct set of results if an injective matching is required.*

Proof. Appears in Appendix B. □

Definition 5.3.4. Given a query Q, if there exists a node v such that one subtree child of v, say s_B, is hidden by one or more other subtree children of v, say $s_{P_1}, \ldots, s_{P_k}$, we call the subtree spanned by v, s_B and $s_{P_1}, \ldots, s_{P_k}$ a *recurrence* subtree of degree $k+1$ of Q, or a $k+1$-recurrence subtree of Q as there are exactly $k+1$ occurrences of s_B in children of v.

We use the notations of s_B and s_P used in Definition 5.3.2 in the rest of this section to refer to the subtree and the supertree that can lead to a violation of the injective matching property. Note that s_B and s_P could be equal subtrees, and s_B does not have to be a proper subtree of s_P. In order to guarantee that injective matching property holds for all matches, we need to guarantee that every node of s_B matches distinct postings, compared to the posting matched by its corresponding node in s_P. In Example 5.3.1, each `JJ` node can hide the other `JJ` node and each `NN` node can hide the other two `NN` nodes. Thus, `NP(JJ)(JJ)` is a *2-recurrence* and `NP(NN)(NN)(NN)` is a *3-recurrence* subtree of the query.

Definition 5.3.5. The *granularity* of a join over a parent-child axis v/u, where v and u are subtrees of a given query Q, is the minimum number of distinct instances of u that must participate in the join with v, in order for u and v to form a match.

Definition 5.3.5 can be used to define a constraint on a parent-child join requiring a match to have more than one instance of the child per parent. This constraint will be helpful in forcing injective matching in the scenarios where the query explicitly requires more than one match for a child, and the matching is injective.

The idea behind solving the injective matching is that for each hidden subtree s_B under a *recurrence* subtree S of degree d, there must be at least d distinct occurrences of s_B in a parent-child relationship with S. Thus, adding a join with granularity equal to d will guarantee that all the matches for s_B are distinct. In order to guarantee that a query Q is matched injectively, all such *recurrence* subtrees have to be found and the corresponding joins have to be added to the list of joins (which include those required for the cover of Q, and obtained using one of our previous algorithms).

In Figure 5.11 we propose an algorithm that adds additional joins or updates current ones in order to guarantee that matching is injective. As this figure shows, in lines $4-5$ of the $addRCJoins$ algorithm, for each child c_i of Q, we count the number of children of Q that hide c_i, and store the count in hs. The $isASubtree$ algorithm returns true if $c_i \precsim c_j$ and false otherwise. In lines $6-7$, if $hs > 1$, meaning that there is at least another child of Q that hides c_i, we update the set of joins by a parent-child join with parent equal to Q and child equal to c_i. The $updateJoins$ algorithm checks the list of joins in which p is a parent and if finds a join in which c is a child, updates its join granularity. If there is no such a join, it adds the join to the list of p's joins and returns.

In the next chapter, we experimentally study the performance of the index structures and access methods proposed in Chapters 4 and 5.

```
$\mathbf{addRCJoins(Q)}$
for $c_i \in Q.children$
  $hs = 1$
  for $c_j \in Q.children$
    if $\mathbf{isASubtree(c_j, c_i)}$ and $c_i \neq c_j$
      $hs = hs + 1$
  if $hs > 1$
    $\mathbf{updateJoins(Q, c_i, hs)}$
for $c \in Q.children$
  $\mathbf{addRCJoins(c)}$
```

```
$\mathbf{updateJoins(p, c, g)}$
for $j \in p.joins$
  if $c = j.child$ and $j.type = ParentChild$
    $j.granularity = g$
    return
$p.joins \leftarrow p.joins \cup Join(p, c, g, ParentChild)$
```

Figure 5.11: The algorithm that computes extra joins that guarantee injective matching

Chapter 6

Experimental Results

In this chapter, we experimentally study the performance of our proposed solutions. We first consider the scenario where text is represented as sequences of words, and study the performance of Word Permuterm Index compared to the baseline methods in terms of measures such as query response time and index construction time. Next we study the performance of our subtree index under our proposed coding schemes. We report the performance in terms of the number of joins involved in evaluating each query, the runtime of queries and size of the index under different coding schemes.

6.1 Natural Language Text as Sequences of Words

6.1.1 Experimental Setup

For our experiments, we used all or parts of the following two text collections. (1) *News Dataset* is the AQUAINT corpus of English News Text [5], which we processed and extracted the sentences to be indexed. It contains around 18 million sentences and its size is more than 2 GBs. (2) *Web Dataset* is our crawl of the web done on May 2008, which contains around 2 million documents and is around 8 GBs in size.

We created three sets of wild card queries for our experiments. (1) *WHQ query-set* was created by replacing the *wh* keywords in *who* and *what* questions from AOL query log [94] with a wild card. (2) *SVO query-set* was generated by randomly replacing the subject or the object of a Subject-Verb-Object relation with a wild card. We obtained the SVO relationships using the Minipar dependency parser [82]. Finally, (3) *n-gram query-set* was generated by randomly replacing a keyword with a wild card in an n-gram, with $n = 1..5$. These n-grams were selected according to their number of bindings in our datasets, in an attempt to cover a wide range of bindings.

WPI is a memory-based index, hence to be fair to other indexes we assigned in our

Table 6.1: Summary of the performance of the indexes in terms of the running time in seconds

		News data 10M sentences			Web data 1M documents		
		n-gram	WHQ	SVO	n-gram	WHQ	SVO
Avg Bindings		2.5e+5	0.4	3.3e+3	5.4e+5	5.1	220
Min	**WPI**	**2e-6**	**1e-6**	**4e-6**	**3e-6**	**2e-6**	**4e-6**
	Neighbor	1e-4	0.008	0.005	1.34e-4	0.274	0.013
	Inverted	0.03	0.007	0.028	0.01	0.064	0.022
	Memscan	82.0	86.6	83.5	30.8	29.4	29.7
Max	**WPI**	**0.03**	**2.5e-4**	**2.6e-4**	**0.06**	**0.01**	**0.02**
	Neighbor	24.0	10.0	4.30	194	8.98	10.2
	Inverted	1.6e+4	8.99	493	4.6e+4	4.03	35.8
	Memscan	431	432.9	424.8	219.8	116	33.2
Avg	**WPI**	**3.5e-4**	**6.6e-5**	**1.2e-4**	**6.8e-4**	**2.5e-4**	**3e-4**
	Neighbor	1.37	0.93	1.20	5.42	1.44	0.77
	Inverted	373	0.73	9.71	1.2e+3	0.75	1.66
	Memscan	87	90.5	87.47	44.9	31.3	30.7

experiments as much cache to the inverted and the neighbor indexes as the memory used by WPI. We ran each query multiple times and only considered the last running time, in order to make sure cache is being utilized by the querying engine. Neighbor and inverted indexes were implemented over Berkeley DB, with the terms as the keys and posting lists as values.

6.1.2 Performance of Querying

Our first set of experiments compared the performance of the indexes under different settings, in terms of the average running time of queries in seconds. Table 6.1 gives a summary of the performance of each index over 10 million sentences of news data and 1 million documents of the web data and all the query sets.

As Table 6.1 suggests, WPI performs the best among all indexes on any combination of data and query sets. The third row of the table shows the average number of bindings per query for each query and data set used. Neighbor index performs relatively good when the number of bindings is high. Inverted index performs very poorly on queries that match a large number of documents. MemScan performs relatively slow regardless of what type of query is given. The statistical correlation of the running time of queries over indexes is largest for inverted index and smallest for WPI. These correlations reflect how the indexes perform when the number of bindings grow. Figures 6.1 and 6.2 depict the behavior of these four methods with respect to the number of bindings of a query, plotted over 100 n-gram queries over 10 million sentences of news data and 1 million documents of web data, respectively. As these figures show, the running time of WPI is almost entirely independent of the number of bindings of the query. For the data presented in these figures, on average WPI is 5 orders of magnitude faster than the neighbor index. The worst case performance of WPI is still an order of magnitude faster than the neighbor index whereas in its best case, WPI is 6-7 orders of magnitude faster. The worst case, observed as a spike in Figures 6.1 and 6.2 for the running time of WPI, belongs to the query `'the % of'`. The running time of WPI on this particular query is relatively higher because the query is of type $\alpha\%\beta$ whose running time complexity is decided by $\|\alpha\|$ and $\|\beta\|$ according to Table 4.1. Since α='the' and β='of' are the two highest selective words in the alphabet, we observe the spike in these two figures.

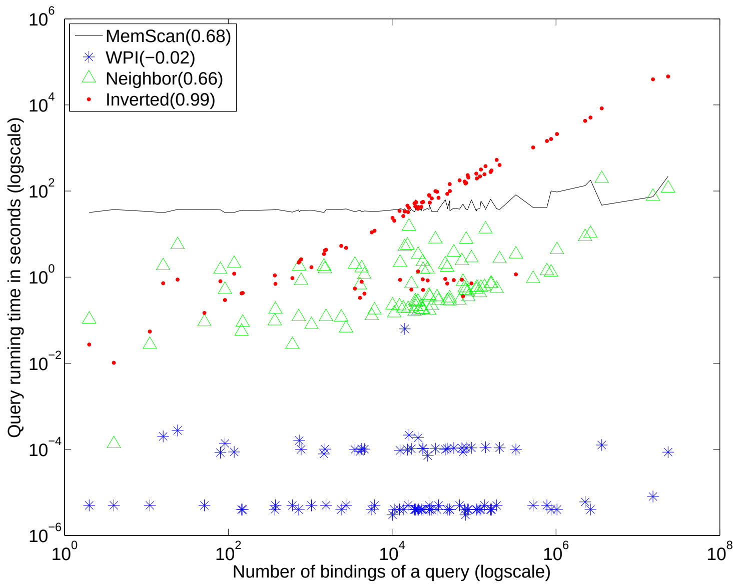


Figure 6.1: The performance of the indexes based on the number of bindings of queries over 10 million sentences of news data

In order to compare the scalability of the indexes we conducted another experiment to compare how the indexes perform as the dataset size grows. Figure 6.3 shows the total querying time of the four indexes over 1000 *SVO* queries computed over web datasets of sizes 0.4, 0.8, 1.2, 1.6 and 2 million documents. As this figure shows, the running time of WPI stays almost constant. Starting as low as 0.095 seconds for 0.4 million documents and going up to at most 0.118 seconds for 2 million documents, WPI shows only 24% growth in the overall querying time. The running times of the neighbor and the inverted indexes grow almost linearly with the dataset size. The minimum (maximum) running times are

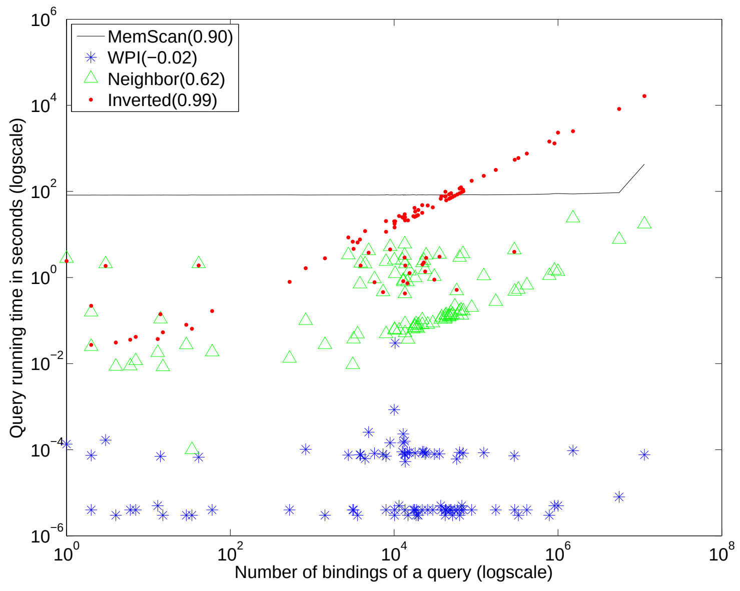


Figure 6.2: The performance of the indexes based on the number of bindings of queries over 1 million documents of web data

370 (1693) and 600 (2211) for the neighbor and the inverted indexes, respectively. Finally, MemScan shows an exponential growth with respect to the dataset size. The maximum running time (for 2 million documents) shows almost two orders of magnitude growth with respect to the minimum running time of MemScan.

6.1.3 WPI Performance with Limited Physical Memory

Given that WPI is a memory-based index, it is important to evaluate its performance in settings where the space consumption of WPI exceeds the available system physical memory. This is a worst-case scenario for WPI whereas the inverted and the neighbor indexes are not expected to be affected much by limitations on the size of memory. A straight-forward solution would be to use disk as a supplementary storage and allocate more memory than available and let the operating system do the paging[1] (i.e. decide which memory blocks to swap out to disk). In an attempt to push WPI to do paging, we ran a set of experiments on the news data of sizes 4, 6, 8, 10, 12, 14, 16 and 18 million sentences and the web data of sizes 0.4, 0.8, 1.2, 1.6 and 2 million documents. We used a machine with 4 GBs of physical memory, around 0.8 GB of which was reserved by a distribution of the Linux operating system for kernel and other system processes. We report here the amount of memory that was required for storing all data structures required by WPI, as a percentage of the available

[1]the terms swapping and paging are used interchangeably in this paper

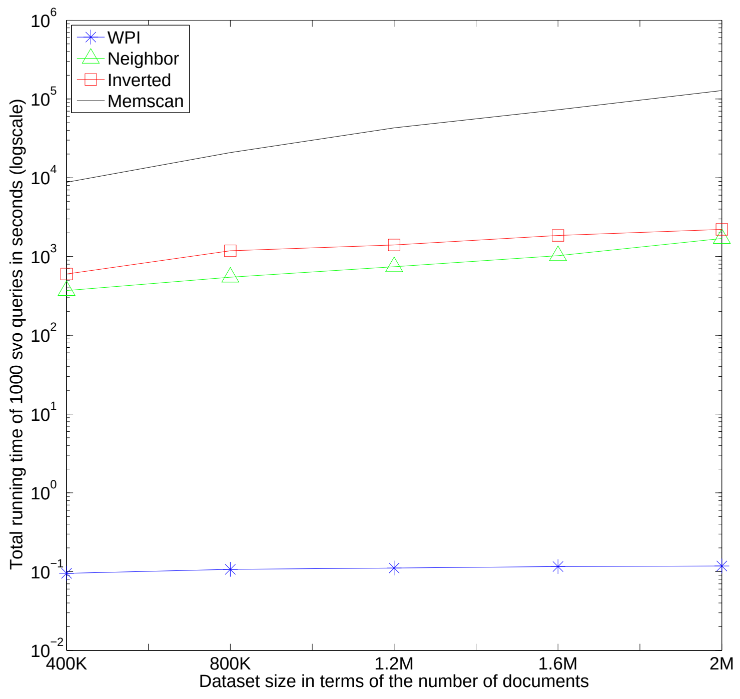


Figure 6.3: Scalability of the indexes over web data of growing sizes.

system physical memory. These memory requirements are depicted on the horizontal axis of Figures 6.4 and 6.5 for different sizes of data. The reported values are not the peak memory usage of operating system for WPI process as the process needed additional memory for code, stack and other static and dynamic data items. Hence, the amount of memory the process required exceeded the above figures, and paging could happen for the smaller datasets as well.

Figures 6.4 and 6.5 show the total running time of 1000 *SVO* queries over WPI and the neighbor index as the datasets vary in size. As Figure 6.4 shows, WPI's running time grows dramatically as its size grows to 80% of the memory size. This shows the effect of paging on the WPI process. Moreover, as the figures show, even when paging happens, the running time of WPI is still much lower than the neighbor index. By increasing the swap size, we were able to run WPI over datasets that required memory equal to approximately 10 times that of the available system memory. For large datasets, a major part of the index resides over disk and increasing the dataset size, as our results suggest, does not dramatically change the running time of the queries. Even with such a naive disk-based solution to WPI, it performs pretty well and can scale up well with limited available memory.

The total running times of queries for the inverted index and MemScan exceed those of the neighbor index in Figures 6.4 and 6.5 and have been omitted for brevity.

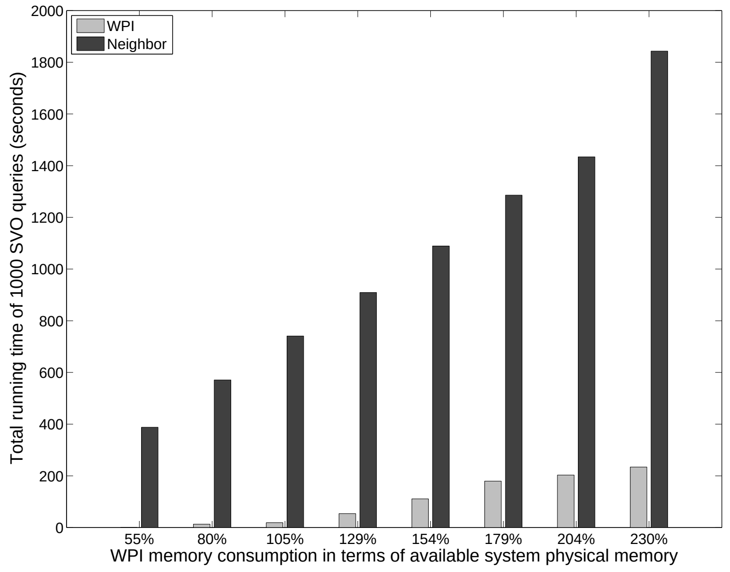


Figure 6.4: The performance of WPI vs. the neighbor index using paging on News Data of sizes 4, 6, 8, 10,12, 14, 16 and 18 million sentences

6.1.4 Index Construction Time

Table 6.2 shows the time required to construct WPI compared to the neighbor index for our experiment in Section 6.1.3. As this table suggests, the construction time of WPI is smaller than the neighbor index for the given sets of data. In most cases, inverted index has a slightly lower construction time than WPI and memory scan can be considered as having no construction time except loading the dataset once into the main memory.

Table 6.2: Index construction time of WPI compared to the neighbor index in seconds

	News dataset sentences					
	4M	6M	8M	10M	12M	14M
WPI	457	689	796	1172	2191	2246
Neighbor	871	1439	2028	2265	3170	3853

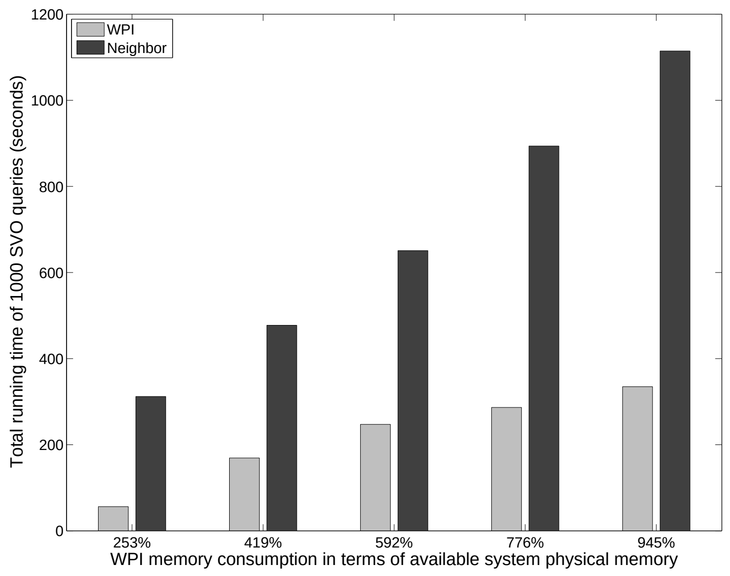


Figure 6.5: The performance of WPI vs. the neighbor index using paging on Web Data of sizes 0.4, 0.8, 1.2, 1.6 and 2 million documents

6.2 Natural Language Text as Syntactically Annotated Trees

6.2.1 Experimental Setup

For our experiments in this section, we parsed a collection of sentences from the AQUAINT corpus of English News Text [5], using Stanford Parser [62], and used this dataset or a part of it in our experiments.

We further processed each parsed tree obtained from the parser and assigned ids and structural tags to individual nodes of the subtrees. With this tagging, each node is described as a tuple consisting of *treeId, nodeId, parentId, left, right, level* and *label.* The treeId value points to the corresponding tree that contains the node. The *nodeId* is a numeric value that uniquely identifies each node within a tree, and *parentId* is the *nodeID* of the parent node. The *left, right* and *level* are the structural information of an individual node, as discussed under numbering schemes in Chapter 2.

We constructed two sets of queries over syntactically annotated text for our experiments. The first set, *WH query-set* was created by a third person who was asked to select 48 of the questions extracted from AOL query log [94], 12 questions from each of *what, which, where* and *who* questions. She was then asked to rewrite the questions in the form of sentences that have the same patterns as a sentence with a potential match. For instance a question such as *who is the mayor of New York city?* is converted to *mayor of New York city is %match%.*

Finally, we parsed these sentences using Stanford parser and removed for each sentence the leaves that contain terms from the sentence, leaving only the sentence structure. The list of all these questions and their corresponding query structures are presented in Table A.1 in Appendix A.

Our second query set was constructed by extracting subtrees from a set of parsed sentences which were not included in our indexes. The extracted subtrees were selected according to the frequencies of their nodes. To account for differences in the selectivities of queries that are posed to our indexes, we constructed the following classes of subtrees consisting of (1) all high frequency nodes denoted as H, (2) all medium frequency nodes denoted as M, (3) all low frequency nodes denoted as L, (4) high and medium frequency nodes denoted as HM, (5) high and low frequency nodes denoted as HL, (6) medium and low frequency nodes denoted as ML, and (7) high, medium and low frequency nodes denoted as HML. For each class, we construct 10 subtrees of different sizes, and whose labels all fall in the given set of frequency classes. The size of the queries are selected randomly to be a number between 1 and 10 nodes. We refer to these second category of queries as Frequency Based or *FB query-set*. The list of high, medium and low frequency labels used in generating queries in *FB query-set* appear in Table A.2 in Appendix A.

Our subtree index was implemented as a native disk-based B+Tree index over the flattened unique subtrees as index keys. We did not implement a caching system over the B+Tree and relied on the page buffering of the operating system for any savings in the number of disk page accesses. Each leaf of the B+Tree index pointed to a posting list which was sorted first based on *treeId* and then on *left* values. We also flattened and sequentially stored parse trees in a separate file, which we call the data file. The *treeId* values in the index provided the offset of each individual parse tree in the data file.

All our experiments were run on a 64-bit machine with 64 GB of physical memory and a 4x quad-core processor. The system page size was 4096 bytes. The reported index sizes will be different on 32-bit addressing systems or with different page sizes.

6.2.2 Index Construction

In this section we study the characteristics of the indexes built over syntactically annotated trees experimentally. We investigate how the size of the index is affected by the choice of the coding scheme and size of the input data. We also study the index construction time for different coding schemes and input sizes.

Index Size

Figure 6.6 shows the subtree index size for the three proposed coding schemes, varying input sizes, with the top left sub-figure displaying the index sizes, when input size is 100 sentences, and the top right, bottom left and bottom right sub-figures displaying the same results for input sizes of $1k$, $10k$ and $100k$ sentences, respectively. Furthermore, in each sub-figure, we vary the maximum subtree size, mss, from 1 to 5, as shown on the X axis.

As the figure shows, the size of the index is smallest for filter-based coding, and largest for the subtree interval coding in all cases. One interesting pattern in the results for sizes of the index is that as mss increases, the gap between the sizes of root-split and subtree interval codings grows. The reason is that for larger subtrees, subtree interval coding uses larger postings, because it has to store the structural information for individual nodes. However, the posting size in root-split coding has constant size, and the index size increases only due to more keys being indexed.

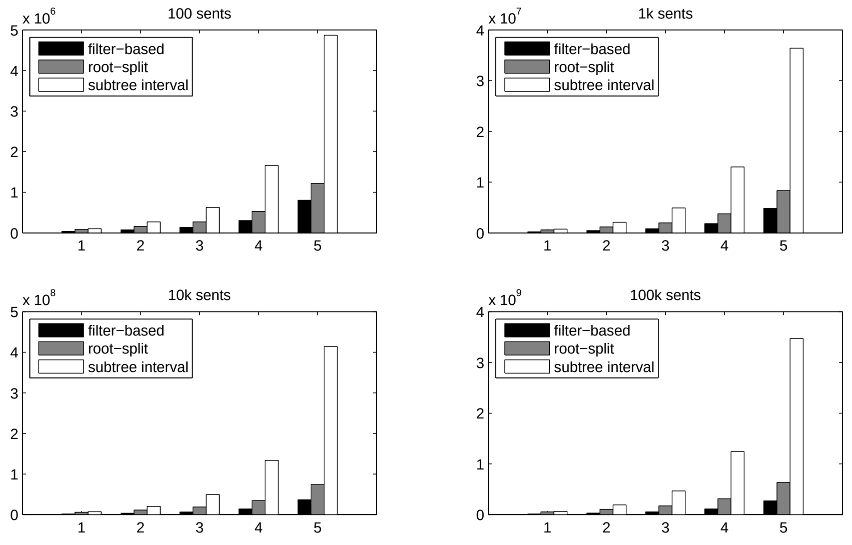


Figure 6.6: SI size for filter-based, root-split and subtree interval codings, with $mss = 1, \ldots, 5$. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences.

Table 6.3 shows the ratio of the index size when mss is 5 to the the index size when mss is 1, for all three coding schemes and four dataset sizes. As the table depicts, root-split coding shows the smallest increase in the size of the index among all coding schemes.

The size reduction for root-split coding is due to (1) reducing the size of each posting as only structural information of roots are stored, and (2) reducing the number of postings as

Table 6.3: Ratio of the subtree index size when mss is 5 to the index size when mss is 1

	Filter-based	**Root-split**	**Subtree Interval**
100	22	15	48
$1k$	24	14	50
$10k$	23	13	59
$100k$	21	12	54

multiple subtrees which have the same key and the same root structural information will be represented with only one posting in root-split coding, while every single subtree requires a distinct posting using the subtree interval coding. Figure 6.7 depicts the number of postings for our three coding schemes, varying the dataset size and mss. As this figure shows, for $mss = 1$ the number of postings of root-split and subtree interval codings are equal and as mss increases the gap between the number of postings for these coding schemes widens. Filter-based coding has the smallest number of postings as it only stores unique $treeId$s, and no structural information.

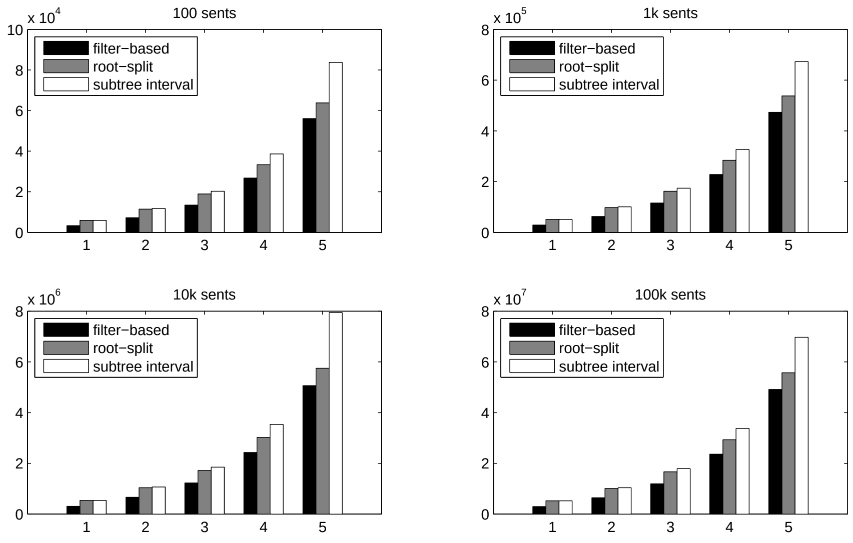


Figure 6.7: Total number of postings over all keys for filter-based, root-split and subtree interval codings, with varying input sizes and mss values. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences.

The number of keys to index varies with changes in the input dataset and the mss values. Figure 6.8 displays the number of keys (in log-scale) in terms of varying input sizes and mss values, with the left sub-figure showing the absolute number of keys and the right sub-figure showing the cumulative number of keys. The figure shows that the number of absolute keys grows dramatically for $mss = 4$ and $mss = 5$. A reason for this increase in the number of keys is that the total number of possible subtree structures of size m follows the m^{th} Catalan number and as m increases, this number can grow dramatically. The first five Catalan numbers are $1, 1, 2, 5$ and 14. The absolute number of keys grows almost proportionately to these values. The cumulative numbers are the number of keys stored in each index for different values of mss.

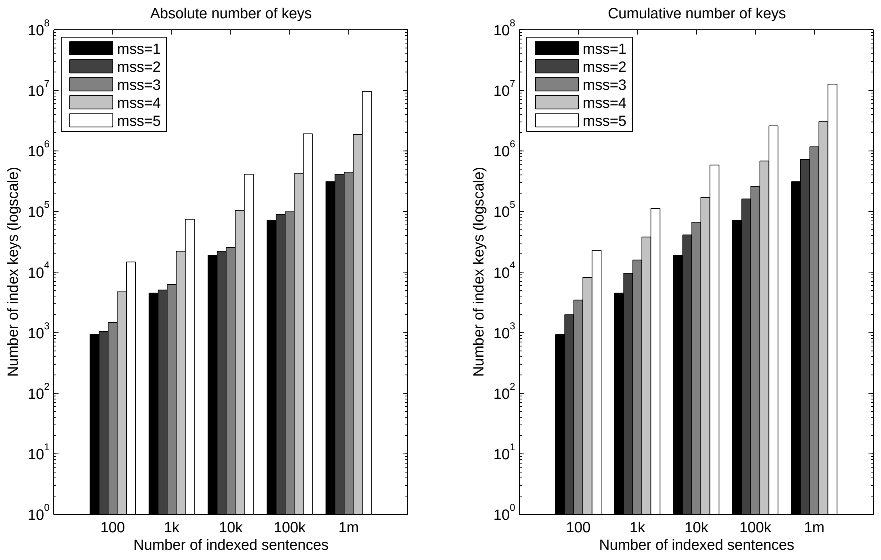


Figure 6.8: Total number of index keys for varying input sizes and mss values. (Left) absolute number of keys, (right) cumulative number of keys.

Finally, to have an idea of the space overhead of the index, the size of a B+tree constructed over subtree inverted lists is comparable to the size of the data file for $mss = 1$. For larger values of mss, the gap between the data file size and subtree index size grows. For $mss = 5$ and subtree interval coding, the size of data file is two orders of magnitude smaller than the subtree index size.

Index Construction Time

Figure 6.9 shows the construction time of the subtree index for different datasets, coding schemes and mss values. As shown, the construction time is smallest for filter-based coding and largest for subtree interval coding. Root-split has a construction time that is slightly larger than filter-based coding. As mss increases the difference in the construction time between subtree interval coding and the other two codings grows. This is mostly because the size of the index for subtree interval coding is larger and as a result more data has to be written on disk.

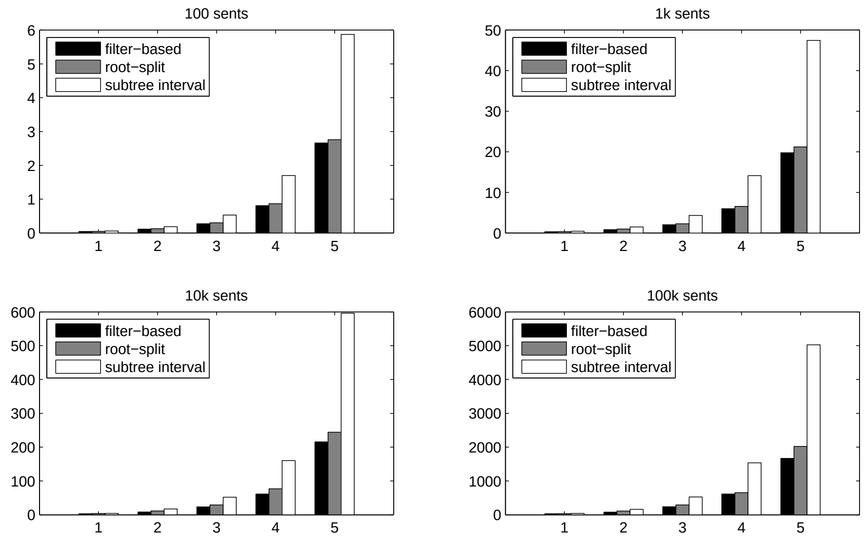


Figure 6.9: Index construction time for filter-based, root-split and subtree interval codings, with $mss = 1, \ldots, 5$. Input size is (top left) 100 sentences, (top right) 1000 sentences, (bottom left) $10,000$ sentences, (bottom right) $100,000$ sentences.

6.2.3 Querying Performance

In this section, we experimentally evaluate the performance of querying our subtree index under different settings. In particular, we investigate the runtime of queries in terms of their number of matches for the filter-based, root-split and subtree interval codings as mss values and query sizes vary. We also present some scalability results for the three coding schemes using data sizes of one thousand to one million sentences. Finally, we study the performance of our splitting algorithms by comparing the number of joins that are required under each split policy.

Response time of queries

To obtain the query response time over our subtree index, we used all the 48 WH queries and 70 FB queries, and tried each query 5 times and took the average running time per query. We grouped the queries according to their number of matches into the following bins: (1) less than 10, (2) between 10 and 100, (3) between 100 and $1k$, (4) between $1k$ and $10k$ and (5) larger than $10k$ matches. Figure 6.10 shows the average run-time of queries varying the number of matches, over $100k$ sentences. Tables A.3 to A.13 in Appendix A display the individual running times of the queries averaged over five runs.

As Figure 6.10 shows, the running time of the queries decreases for all coding schemes as mss grows. This reduction is smallest for queries with large number of matches using filter-based coding, as the time of the *filtering* phase becomes a dominating factor. As shown in the figure, Root-split coding performs better than subtree interval coding in all cases. Filter-based coding performs better than root-split coding for $mss = 1$ and less than 10 matches on average. However, for larger values of mss, which are mainly interesting for a subtree index, root-split coding performs better than the other two coding schemes.

Also, unlike the filter-based coding, both the root-split and subtree interval codings display a reduction in their average query response times for larger number of matches. This happens for the following two reasons: (1) The intermediate result size of a query with a small number of matches could be large and this would affect the runtime of queries under root-split and subtree interval codings, but not under filter-based coding. (2) As can be consulted with Tables A.3 to A.13 in Appendix A, our queries with larger number of terms have on average smaller number of matches; however, these queries require a larger number of joins and take longer for these two coding schemes. This pattern can also be seen in Figure 6.11 where the running time is depicted as the query size varies.

Figure 6.11 displays the runtime of queries in terms of the query size using the same settings as in Figure 6.10. In this figure, we only included queries which have 100 and more matches. As this figure shows, root-split and subtree interval codings show an increasing trend with respect to the size of queries. Filter-based coding displays a somewhat random behavior with respect to the query size as its performance is mostly determined by the number of matches and how well the splits can perform filtering. According to this figure, as mss increases, root-split and subtree interval codings perform better on larger queries as they require smaller number of joins to compute the result set of queries.

Comparison with Other systems

Table 6.4 displays the results of comparing our SI using root-split coding with ATree-Grep [104] and a frequency-based approach that is an adaptation of TreePi [123] for indexing parse trees. These results are over $100k$ sentences and SI uses $mss = 3$. Similar to TreePi, the frequency-based approach stores in the index all single nodes and a percentage of larger highest frequency subtrees. This percentage is denoted in brackets in the last three columns of Table 6.4.

The results in Table 6.4 are obtained over the queries in our *FB query-set* and are grouped by the frequency classes. Since ATreeGrep does not support all the queries, the results are averaged over as many queries as there were results for. As this table depicts, SI with root-split coding outperforms other index structures by at least one order of magnitude over all frequency classes.

Table 6.4: Average running time of queries in seconds for queries in FB query set classes using Subtree index with root-split coding ($mss = 3$), ATreeGrep and Frequency-based approaches with varying frequency cutoff thresholds.

	RS	**ATG**	**FB(0.1%)**	**FB(1%)**	**FB(10%)**
L	**0.09**	1.9	3.05	3.03	3.04
M	**0.01**	10.06	12.32	0.8	0.35
ML	**0.25**	2.13	10.3	9.62	9.25
H	**1.73**	22.4	39.21	34.51	34.53
HL	**1.57**	32.97	34.58	34.61	34.6
HM	**1.76**	37.08	35.54	31.40	31.57
HML	**1.76**	86.02	49.03	42.97	43.13

Scalability Results

Figure 6.12 presents the runtime of our queries over four subsets of our parsed collection: $1k$, $10k$, $100k$ and $1m$ sentences. We used $mss = 3$ for the results reported in this figure, but the result for other values of mss were similar. The reported runtimes are the average query response times for each group of *FB queries* and using our three coding schemes. The results in this figure show that all three coding schemes display a similar pattern as the dataset size increases, i.e. the running time grows approximately linearly with the number of sentences indexed.

Figure 6.12 also shows that the root-split coding scales up better with the dataset size. This is mostly evident on the figure between $1k$ and $10k$ as well as between $10k$ and $100k$ results, especially for query classes that contain *high* frequency labels; i.e. *H*, *HL*, *HM* and *HML*. Averaged over all 7 FB query-set categories, ranging from $1k$ to $1m$ sentences of our parsed collection, the query runtime increases for filter-based, subtree interval and root-split codings by a factor of 1025, 752 and 529, respectively.

Splitting Algorithms Results

Table 6.5 displays the number of joins required per group of 12 queries of *Who*, *Which*, *Where* and *What* queries for mss values of 2 to 5. The values reported for root-split coding are the total number of joins required for evaluating a cover generated using the $minRC$ in addition to the joins obtained from $addRCJoins$ algorithm. Similarly, joins reported for subtree interval codings are due to $optimalCover$ and $addRCJoins$ algorithms (See Sections 5.2.2 and 5.3.2).

As Table 6.5 shows, $optimalCover$ achieves a fewer number of joins for all groups of queries and mss values[2]. Despite a fewer number of joins obtained for filter-based and subtree interval codings, root-split still manages to have a smaller query response time, by minimizing the I/O cost and avoiding to perform filtering.

Table 6.5: Total number of joins required over queries in the WH query set. r=root-split, s=subtree interval.

Query-set	$mss = 2$		$mss = 3$		$mss = 4$		$mss = 5$	
	r	**s**	**r**	**s**	**r**	**s**	**r**	**s**
Who	71	65	57	40	36	26	29	20
Which	82	75	65	51	51	36	39	27
Where	59	57	53	40	32	25	27	19
What	67	64	55	40	35	27	27	19

[2]In the case where $mss = 1$, root-split and subtree interval will have equal number of joins, which is equal to $|Q| - 1$

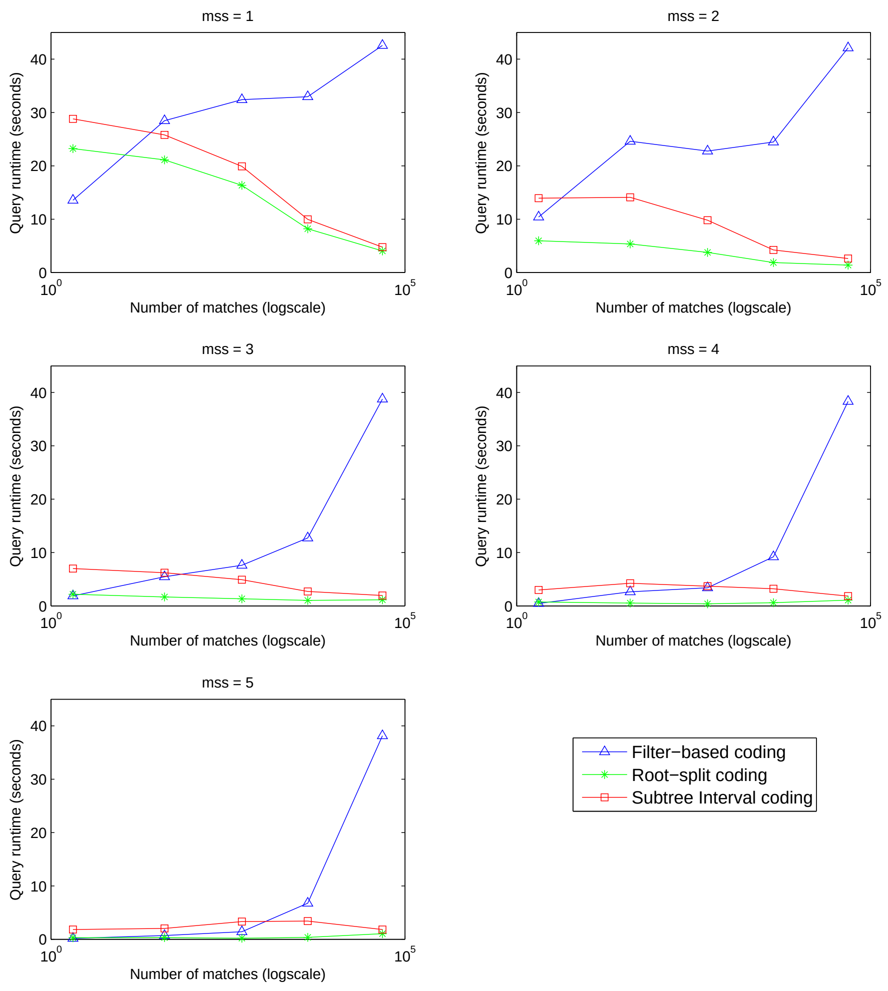


Figure 6.10: Average runtime of queries in terms of their number of matches for filter-based, root-split and subtree interval codings and mss values of 1 to 5

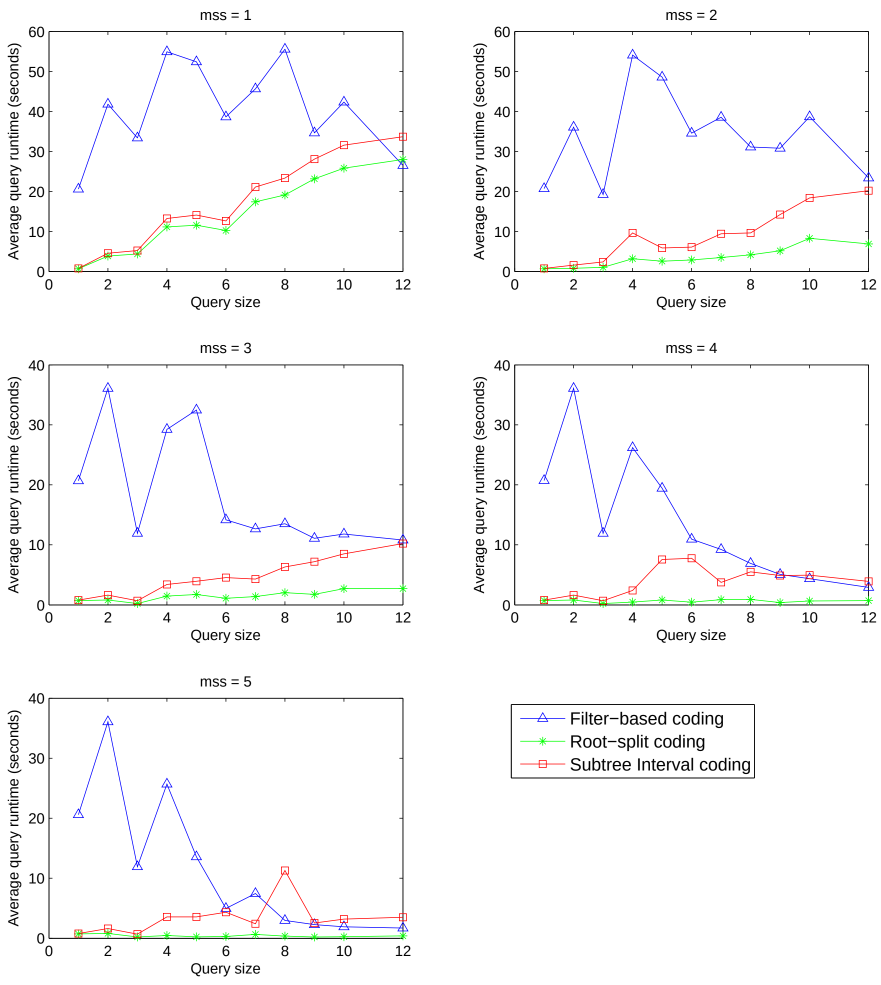


Figure 6.11: Average runtime of queries in terms of the size of queries for filter-based, root-split and subtree interval codings and mss values of 1 to 5

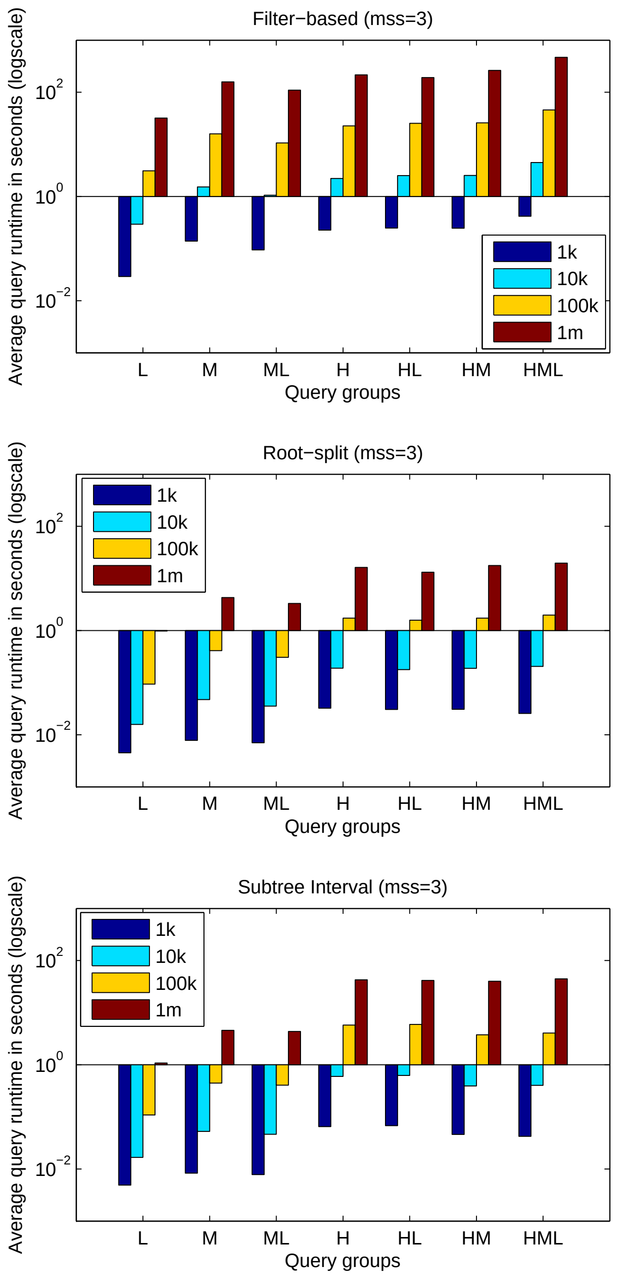


Figure 6.12: Average runtime of queries ($mss = 3$) over groups of FB queries over datasets of $1k$, $10k$, $100k$ and $1m$ sentences and using different coding schemes.

Chapter 7

Conclusions and Future Directions

This chapter concludes this thesis by providing a discussion of the benefits of the approaches presented, their limitations, and the avenues this thesis opens for future work and improvements.

7.1 Summary and Discussion

In this thesis we studied the problems of indexing and querying over natural language text under two scenarios. In the first scenario, we considered text as sequences of words and studied indexing techniques over word level wild card queries. In the second scenario, we considered text as a collection of syntactically annotated trees, and studied efficient methods for retrieving subtrees matching a query tree.

7.1.1 Word Permuterm Index

In the first scenario, we discussed the development of Word Permuterm Index (WPI) which supports single wild card natural language queries. WPI fills in the gap for a time-efficient index supporting a wide range of wild card queries over natural language text. Our asymptotic analysis of the complexity bounds of querying over different indexes shows a better time complexity for WPI over other approaches. Our wide range of experiments over different combinations of data and query sets and the number of bindings show a large gap in terms of the performance between WPI and the neighbor and the inverted indexes. Our results also show that WPI performs better than the neighbor index even in the lack of sufficient physical memory, resulting in paging memory pages in and out of the disk which greatly reduces its performance. Word permuterm index is limited to wild card queries containing only a single wild card. More general wild card queries have not been the focus of word permuterm index and further optimization might be necessary to support queries with more than one wild card.

7.1.2 Subtree Index and Root-split coding

In the second scenario, we proposed subtree index (SI) as a novel indexing strategy over syntactically annotated trees. We studied the architecture of SI and provided algorithms for building it. We further investigated different coding schemes for encoding subtree information into inverted lists of SI and proposed two baseline coding schemes, filter-based and subtree interval codings, and a novel root-split coding scheme. Later, we discussed query matching over SI which includes two phases, a split phase and a join phase. In the split phase, we studied algorithms that compute join optimal covers over subtree interval and filter-based codings. We also proposed the $minRC$ algorithm that computes the best possible root-split cover, which might not be join optimal. In the join phase, we studied the join approaches over SI and discussed how query matching is affected when we have injective matching assumption. Our experimental results shows that root-split coding performs better than subtree-interval coding in all cases and filter-based coding might outperform root-split coding in a rare case where SI is constructed only over the nodes and the number of matches of the query is very small. We also experimentally showed that the index size and index construction time of root-split coding are better than those of subtree interval coding and slightly larger than those of filter-based coding. Finally, we showed that over our set of queries, $minRC$ algorithm generates covers which are smaller than the theoretical worst case discussed, in terms of the number of their subtrees.

One limitation of the current implementation of SI is that it does not support all axes as efficiently as the parent-child axis. For instance, a query containing all ancestor-descendant axes has to be split into its nodes. Thus, storing larger subtrees in the index will not be beneficial for such queries.

7.1.3 Discussion

Word permuterm index and subtree index were introduced in this thesis to address two scenarios that natural language text is often represented with. These scenarios consider text as sequences of words or as collections of parse trees. However, natural language text can be represented in numerous other formats. For instance, one could consider flat structures that are annotated with part of speech tags. As another example, annotations exist over text that can be modelled using a directed acyclic graph or a general graph. It is important to note that this thesis only studies two of these scenarios and leaves the rest to the future work.

Subtree index (SI) provides the functionality to search for node labels and relationships between them expressed using navigational axes. Therefore, it can provide more informed forms of searches compared to WPI. WPI provides very fast in-memory searches over wild card queries, while SI does not directly support wild cards in its queries and requires extra effort to answer such queries. Direct support of wild card queries over SI could be an interesting future work. While the two approaches address different querying needs and query formats, they might both be useful in retrieving more meaningful answers to natural language questions.

7.2 Future Directions

Our solutions and algorithms can be extended in a few interesting directions.

(i) Allowing the operating system to swap memory pages in and out of the disk is a naive approach for solving the high memory consumption of WPI. One future extension would be to benefit from the localities available in natural language text to store WPI structures over disk in such a way that the number of disk block accesses is optimized, hence, increasing the efficiency.

(ii) High space consumption is currently one of the main drawbacks of WPI. An area for improvement is finding compression techniques that can further reduce the size of WPI. Another area is to make use of parallelism and distribute WPI over multiple machines.

(iii) SI improves the query response time at the expense of more space consumption and a longer index construction time. One possible improvement over SI would be to materialize certain subtrees, rather than all subtrees. Which subtrees to materialize could be learned from the distribution of the subtrees in a query log over syntactically annotated trees, if one is available. Otherwise, an objective cost function can be used to estimate the cost of joins to build each subtree and materialize subtrees whose costs exceeds a certain threshold, hence limiting the worst case query runtime performance.

(iv) Our query splitting algorithms for different coding schemes over SI only take into account the size of a cover, in terms of the number of subtrees included, as a measure for improving the query execution performance. It would be interesting to study how much the performance can be improved by using more levels of optimizations such as building histograms and taking selectivities of subtrees into account.

(v) Currently the subtree interval and the root-split codings use the multi-predicate merge join approach which performs extra comparisons for parent-child axes. As a result, querying performance over these two coding schemes is degraded compared to the filter-based approach, which uses a straightforward sorted list intersection approach. It would thus be interesting to adopt more recent structural join approaches such as StackTree [1] and TwigStack [11] for our coding schemes, and study the performance of SI under such algorithms.

Bibliography

[1] S. Al-Khalifa, HV Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2002.

[2] Altavista. `http://www.altavista.com`.

[3] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for XML. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2005.

[4] A. Andersson, N.J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[5] The aquaint corpus of english news text, 2002. `http://www.ldc.upenn.edu/Catalog/docs/LDC2002T31/`.

[6] N. Augsten, D. Barbosa, M. Bohlen, and T. Palpanas. Tasm: Top-k approximate subtree matching. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2010.

[7] D. Bahle, H.E. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. In *Proceedings of the International Conference on Information Retrieval (SIGIR)*, 2002.

[8] M. Banko, M.J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[9] S. Bird, Y. Chen, S.B. Davidson, H. Lee, and Y. Zheng. Designing and evaluating an xpath dialect for linguistic queries. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2006.

[10] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2006.

[11] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2002.

[12] M. Burrows and D.J. Wheeler. *A block-sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, 1994.

[13] M.J. Cafarella and O. Etzioni. A search engine for natural language applications. In *Proceedings of International Conference on World Wide Web (WWW)*, 2005.

[14] M.J. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured querying of web text data: A technical challenge. In *Proceedings of the Biannual Conference on Innovative Data Systems Research (CIDR)*, 2007.

[15] S. Cassidy and J. Harrington. Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1-2):61–77, 2001.

[16] Online Computer Library Center. Dewey decimal classification, 2006. `http://www.oclc.org/dewey`.

[17] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *Proceedings of International Conference on World Wide Web (WWW)*, 2006.

[18] S. Chaudhuri, K. Church, A.C. Konig, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *Proceedings of the International Conference on Information Retrieval (SIGIR)*, 2007.

[19] S. Chen, H.G. Li, J. Tatemura, W.P. Hsiung, D. Agrawal, and K.S. Candan. Twig 2 Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2006.

[20] T. Chen, J. Lu, and T.W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2005.

[21] Y. Chen and D. Cooke. Unordered tree matching and strict unordered tree matching: The evaluation of tree pattern queries. In *Proceedings of Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2010.

[22] Z. Chen, HV Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2001.

[23] Z. Chen, HV Jagadish, L.V.S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2003.

[24] Y.S. Choi. Tree pattern expression for extracting information from syntactically parsed text corpora. *Data Mining and Knowledge Discovery*, 1(2):211–231, 2011.

[25] J. Christensen, Mausam, S. Soderland, and E. Etzioni. Semantic role labeling for open information extraction. In *Proceedings of the North American Chapter of The Association for Computational Linguistics (NAACL)*, 2010.

[26] P. Chubak and D. Rafiei. Index Structures for Efficiently Searching Natural Language Text. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, 2010.

[27] J. Clark and S. DeRose. Xml path language (xpath), 1999. `http://www.w3.org/TR/xpath`.

[28] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic O (n log 3 n)-time. In *Proceedings of the Symposium on Discrete algorithms (SIAM)*, 1999.

[29] W3C Consortium. Xquery 1.0: An xml query language, 2007. `http://www.w3.org/TR/xquery/`.

[30] H.T. Dang, D. Kelly, and J. Lin. Overview of the TREC 2007 question answering track. In *Proceedings of Text REtrieval Conference (TREC)*, 2007.

[31] D. DeHaan, D. Toman, M.P. Consens, and M.T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2003.

[32] P.F. Dietz. Maintaining order in a linked list. In *Proceedings of the Symposium on Theory of Computing (STOC)*, 1982.

[33] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *Journal of the ACM*, 41(2):205–213, 1994.

[34] O. Etzioni, M.J. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D.S. Weld, and A. Yates. Web-scale information extraction in knowitall: (preliminary results). In *Proceedings of International Conference on World Wide Web (WWW)*, 2004.

[35] S. Evert and H. Voormann. The nite query language, 2002. `http://www.ltg.ed.ac.uk/NITE/documents/NiteQL.v2.1.pdf`.

[36] L. Faulstich, U. Leser, and T. Vitt. Implementing a Linguistic Query Language for Historic Texts. *Proceedings of the Conference on Current Trends in Database Technology (EDBT)*, 2006.

[37] P. Ferragina and J. Fischer. Suffix arrays on words. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, 2007.

[38] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):581, 2005.

[39] P. Ferragina and R. Venturini. Compressed permuterm index. In *Proceedings of International Conference on Information Retrieval (SIGIR)*, 2007.

[40] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *Data Engineering Bulletin*, 22(3):27–34, 1999.

[41] E. Garfield. The permuterm subject index: An autobiographical review. *American Society for Information Science*, 27(5):288–291, 1976.

[42] S. Ghodke and S. Bird. Fast Query for Large Treebanks. *Proceedings of the North American Chapter of The Association for Computational Linguistics (NAACL)*, 2010.

[43] R. Goldman and M. Jarke. Dataguides: Enabling query formulation and optimization in semistructured. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 1997.

[44] Google. `http://www.google.com`.

[45] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *Transactions on Database Systems*, 30(2):491, 2005.

[46] M. Götz, C. Koch, and W. Martens. Efficient algorithms for the tree homeomorphism problem. In *Proceedings of the International Conference on Database Programming Languages (DBPL)*, 2007.

[47] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[48] N. Grimsmo, T.A. Bjorklund, and M.L. Hetland. Fast Optimal Twig Joins. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2010.

[49] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Symposium on Discrete Algorithms (SIAM)*, 2003.

[50] T. Grust. Accelerating XPath location steps. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2002.

[51] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2003.

[52] S. Guha, HV Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *Proceedings of International Conference on Management of Data (SIGMOD)*. ACM, 2002.

[53] H. He and A.K. Singh. Closure-tree: An index structure for graph queries. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2006.

[54] U. Heid, H. Voormann, J.T. Milde, U. Gut, K. Erk, and S. Pado. Querying both time-aligned and hierarchical corpora with NXT Search. In *Proceedings of Conference on Language Resources and Evaluation (LREC)*, 2004.

[55] E. Hinrichs, S. Kübler, K. Naumann, H. Telljohann, and J. Trushkina. Recent developments in linguistic annotations of the tüba-d/z treebank. In *Proceedings of the Third Workshop on Treebanks and Linguistic Theories (TLT)*, 2004.

[56] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[57] Indri - language modeling meets inference networks. `http://www.lemurproject.org/indri/`.

[58] Oracle text, an oracle technical white paper, 2005. `http://www.oracle.com/technology/products/text/pdf/10gR2text_twp_f.pdf`.

[59] A. Ittycheriah, M. Franz, and S. Roukos. IBM's statistical question answering system. *Proceedings of Text REtrieval Conference (TREC)*, 2002.

[60] S. Kepser. Finite Structure Query: A tool for querying syntactically annotated corpora. In *Proceedings of the Conference on European Chapter of the Association for Computational Linguistics (EACL)*, 2003.

[61] S. Kepser. Querying linguistic treebanks with monadic second-order logic in linear time. *Journal of Logic, Language and Information*, 13(4):457–470, 2004.

[62] D. Klein and C.D. Manning. Accurate unlexicalized parsing. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2003.

[63] E. Konig and W. Lezius. The TIGER language: a Description Language for Syntax Graphs. *Technical Report, University of Stuttgart*, 2001.

[64] E. Konig, W. Lezius, and H. Voormann. TIGERSearch user's manual. *University of Stuttgart*, 2003.

[65] T. Krause, J. Richling, V. Rosenfeld, A. Zeldes, F. Zipser, C. Chiarcos, and J. Ritz. Annis2, search and visualization in multilevel linguistic corpora, 2009. `http://www.sfb632.uni-potsdam.de/d1/annis/`.

[66] C. Kwok, O. Etzioni, and D.S. Weld. Scaling question answering to the Web. *Transactions on Information Systems*, 19(3):242–262, 2001.

[67] C. Lai and S. Bird. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australian Language Technology Workshop*, 2004.

[68] C. Lai and S. Bird. LPath+: A first-order complete language for linguistic tree query. In *Proceedings of the Pacific Asia Conference on Language, Information and Computation (PACLIC)*, 2005.

[69] C. Lai and S. Bird. Querying linguistic trees. *Journal of Logic, Language and Information*, 19(1):53–73, 2010.

[70] R. Levy and G. Andrew. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the Conference on Language Resources and Evaluation (LREC)*, 2006.

[71] D. Lin, K. Church, H. Ji, S. Sekine, D. Yarowsky, S. Bergsma, K. Patil, E. Pitler, R. Lathbury, V. Rao, et al. New tools for web-scale n-grams. In *Proceedings of Conference on Language Resouces and Evaluation (LREC)*, 2010.

[72] J. Lin and B. Katz. Question answering techniques for the World Wide Web. *Proceedings on the Conference on European Chapter of the Association for Computational Linguistics (EACL) - Tutorial*, 2003.

[73] J. Lu, T. Chen, and T.W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, 2004.

[74] Apache lucene. `http://lucene.apache.org/java/2_3_2/queryparsersyntax.html`.

[75] A. Maier and H.J. Novak. Db2's full-text search products - white paper, 2006.

[76] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[77] C.D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, 2000.

[78] M.P. Marcus, M.A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):330, 1993.

[79] H. Maryns and S. Kepser. Monasearch: Querying linguistic treebanks with monadic second-order logic. In *The International Workshop on Treebanks and Linguistic Theories*, 2009.

[80] N. Mayo, J. Kilgour, and J. Carletta. Towards an alternative implementation of NXT's query language via XQuery. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2006.

[81] T. Milo and D. Suciu. Index structures for path expressions. *Proceedings of the International Conference on Database Theory (ICDT)*, 1999.

[82] Minipar home page. `http://webdocs.cs.ualberta.ca/~lindek/minipar.htm`.

[83] J. Mirovsky. Netgraph: A tool for searching in prague dependency treebank 2.0. *Proceedings of the Workshop on Treebanks and Linguistic Theories (TLT)*, 2006.

[84] J. Mirovsky. Netgraph - Making Searching in Treebanks Easy. In *Proceedings of the International Joint Conference on Natural Language Processing (IJCNLP)*, 2008.

[85] J. Mirovsky. PDT 2.0 requirements on a query language. *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2008.

[86] J. Mirovsky. Towards a Simple and Full-Featured Treebank Query Language. In *Proceedings of the International Conference on Global Interoperability for Language Resources (ICGL)*, 2008.

[87] D. Moldovan, S. Harabagiu, R. Girju, P. Morarescu, F. Lacatusu, A. Novischi, A. Badulescu, and O. Bolohan. Lcc tools for question answering. In *Proceedings of Text REtrieval Conference (TREC)*, 2002.

[88] D. Moldovan, M. Pacsca, S. Harabagiu, and M. Surdeanu. Performance issues and error analysis in an open-domain question answering system. *Transactions on Information Systems*, 21(2):133–154, 2003.

[89] P. Nakov, A. Schwartz, B. Wolf, and M. Hearst. Supporting annotation layers for natural language processing. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2005.

[90] G. Navarro and V. Makinen. Compressed full-text indexes. *Computing Surveys*, 39(1):2, 2007.

[91] Openephyra - ephyra question answering system. `http://mu.lti.cs.cmu.edu/trac/Ephyra/wiki/OpenEphyra`.

[92] P. Pajas and J. Štěpánek. System for querying syntactically annotated corpora. In *Proceedings of the International Joint Conference on Natural Language Processing (IJCNLP)*, 2009.

[93] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2004.

[94] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proceedings of the International Conference on Scalable Information Systems*, 2006.

[95] U. Petersen. Emdros: A text database engine for analyzed or annotated text. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2004.

[96] U. Petersen. Querying both parallel and treebank corpora: Evaluation of a corpus query system. In *Proceedings of Conference on Language Resouces and Evaluation (LREC)*, 2006.

[97] R. Pito. Tgrep Manual Page. *Linguistic Data Consortium, University of Pennsylania*, 1994.

[98] The penn treebank project, 1999. `http://www.csi.upenn.edu/~treebank/`.

[99] D. Rafiei and H. Li. Data extraction from the web using wild card queries. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, 2009.

[100] B. Randall. CorpusSearch users manual. *University of Pennsylvania*, 2000.

[101] D.L.T. Rohde. TGrep2 User Manual version 1.15. *Massachusetts Institute of Technology.* *`http://tedlab.mit.edu/dr/Tgrep2`*, 2005.

[102] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 1999.

[103] D. Shasha, J.T.L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2002.

[104] D. Shasha, J.T.L. Wang, H. Shan, and K. Zhang. Atreegrep: Approximate searching in unordered trees. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2002.

[105] I. Steiner and L. Kallmeyer. VIQTORYA–a visual query tool for syntactically annotated corpora. In *Proceedings of Conference on Language Resources and Evaluation (LREC)*, 2002.

[106] I. Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2002.

[107] Y. Tian, R.C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232, 2007.

[108] Y. Tian and J.M. Patel. Tale: A tool for approximate large graph matching. *Proceedings of International Conference on Data Engineering (ICDE)*, 2008.

[109] M. Volk, J. Lundborg, and M. Mettler. A search tool for parallel treebanks. In *Proceedings of the Linguistic Annotation Workshop*, 2007.

[110] E. M. Voorhees and D. M. Tice. Building a question answering test collection. In *Proceedings of the International Conference on Information Retrieval (SIGIR)*, 2000.

[111] S. Wallis and G. Nelson. Exploiting fuzzy tree fragment queries in the investigation of parsed corpora. *Literary and Linguistic Computing*, 15(3):339–361, 2000.

[112] S. Wallis and G. Nelson. Knowledge discovery in grammatically analysed corpora. *Data Mining and Knowledge Discovery*, 5(4):305–335, 2001.

[113] D.W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2007.

[114] H.E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *Transactions on Information Systems*, 22(4):573–594, 2004.

[115] F. Wu and D.S. Weld. Open information extraction using wikipedia. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, 2010.

[116] Yahoo! search - web search. `http://search.yahoo.com`.

[117] X. Yan, P.S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2004.

[118] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *Transactions on Internet Technology*, 1(1):110–141, 2001.

[119] M. Yue. A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1, \forall l$ for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica (English Series)*, 7(4):321–331, 1991.

[120] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2001.

[121] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

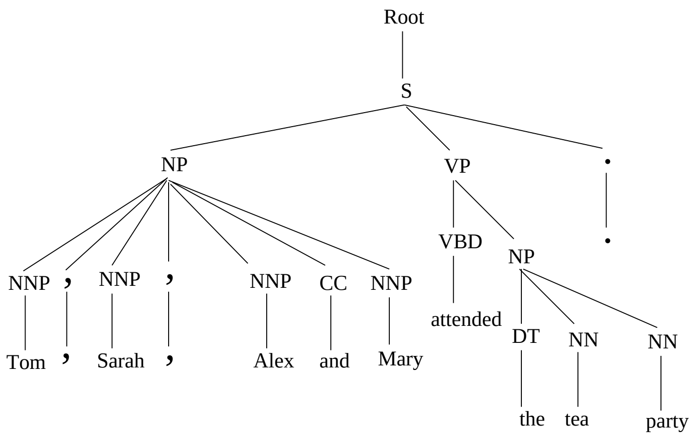
[122] N. Zhang, V. Kacholia, and M.T. Ozsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2004.

[123] S. Zhang, M. Hu, and J. Yang. TreePi: A novel graph indexing method. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2007.

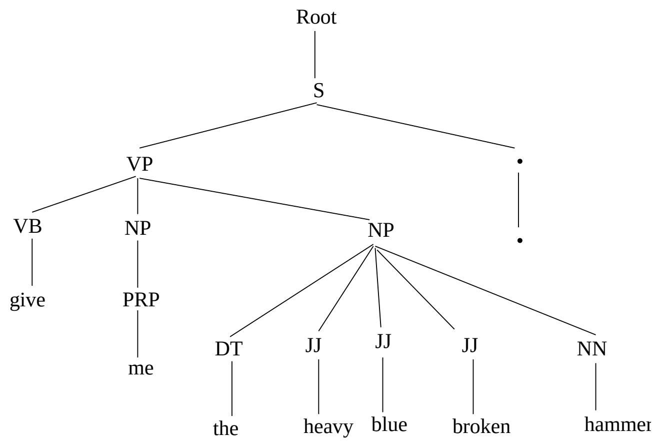
Appendix A

Supplementary Tables and Figures

In this chapter we present a few supplementary tables and figures. Figure A.1 displays two sample parse trees with high branching nodes described in Example 5.1.1.



(a)



(b)

Figure A.1: Sample natural language text sentences that contain nodes with high branching factors

Figure A.2 displays the number of extracted subtrees in terms of the branching factor of the corresponding node over which the extraction happens (See Section 5.1).

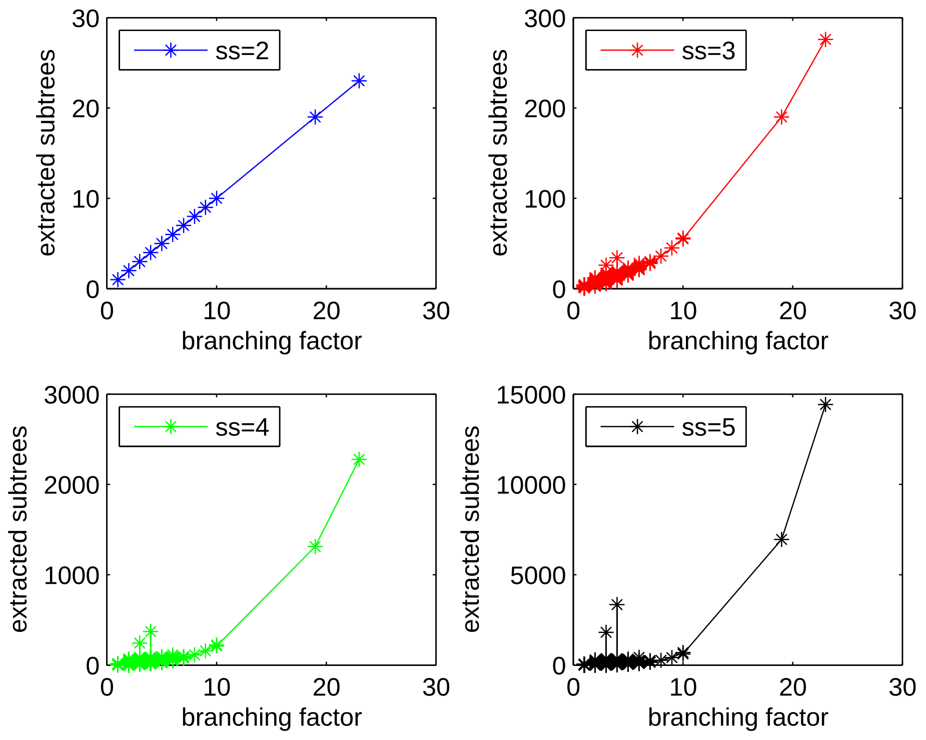


Figure A.2: Number of subtrees with sizes varying between 2 to 5 in terms of the branching factor of the nodes over which the subtrees are constructed

Table A.1 depicts the individual set of questions used in generating the queries in *WH query set* and their corresponding queries used in our experiments (See Section 6.2.1. Note that the questions listed might have misspellings and errors which originate from the queries in the query-log.

Table A.2 depicts the set of low, medium and high frequency labels we use for generating our set of *FB query-set* in Section 6.2.1.

Tables A.3 to A.13 display display the individual running times of the queries averaged over five runs for our queries in *WH query-set* and *FB query-set*.

Table A.1: The list of WH queries and their corresponding query structures

who got fired on tonight's episode of the apprentice?	S(NP(NN)VP(VBD VP(VBN PP(IN NP(NP(NP(NN POS))NN)PP(IN NP(DT NN))))))
who is patch adams?	S(NP(NN NNS)VP(VBZ NP(NN)))
who was cinderella?	S(NP(NN)VP(VBD NP(NN)))
who wants to be a millionaire?	S(NP(NN)VP(VBZ S(VP(TO VP(VB NP(DT NN))))))
who won the boxing fight last night on pay per view?	S(NP(NP(JJ NN)PP(IN NP(NP(NN)PP(IN NP (NN NN)))))VP(VBD NP(DT NN NN)))
who designed the statue of liberty?	S(NP(NN)VP(VBD NP(NP(DT NN)PP(IN NP(NN)))))
who is de vince?	S(NP(FW FW)VP(VBZ NP(NN)))
who invented the tv?	S(NP(DT NN)VP(VBD VP(VBN PP(IN NP(NN)))))
who killed jesus?	S(NP(NN)VP(VBD NP(NNS)))
who is adolf hitler?	S(NP(NN NN)VP(VBZ NP(NN)))
who created chocolate?	S(NP(NN)VP(VBD NP(NN)))
who sings bottom line?	S(NP(NN)VP(VBZ NP(JJ NN)))
which woman had made the most movies?	S(NP(NN)VP(VBD VP(VBN NP(DT JJS NNS))))
which golf ball is the best?	S(NP(NN NN NN)VP(VBZ NP(DT JJS)))
which greek god loved apollo?	S(NP(NP(NN NN)NP(NN))VP(VBD NP(NN)))
which pda is the best to buy?	S(NP(NN)VP(VBZ NP(NP(DT JJS NN)SBAR(S(VP(TO VP(VB)))))))
which planet is called the red planet?	S(NP(NN)VP(VBZ VP(VBN NP(DT JJ NN))))
which presdient started social security?	S(NP(NN NN)VP(VBD NP(JJ NN)))
which cell phone works best for rural coverage?	S(NP(NN NN NN)VP(VBZ NP(NP(JJS)PP(IN NP(JJ NN)))))
which color does hispanic women like on their toes?	S(NP(JJ NNS)VP(VBP NP(NP(NN NN)PP(IN NP(PRP$ NNS)))))
which is the best wave grease to use?	S(NP(NN)VP(VBZ(NP(NP(DT JJS NN NN)SBAR(S(VP(TO VP(VB))))))))
which paper towel is the most absorbant?	S(NP(NN NN NN)VP(VBZ NP(DT JJS NN)))
which cell phone plan is best in miami?	S(NP(NN NN NN NN)VP(VBZ NP NP(JJS)PP(IN NP(NN))))
which state has the most black people?	S(NP(NN NN)VP(VBZ NP(DT ADJP(RBS JJ)NNS)))
where is hell?	S(NP(NNP)VP(VBZ PP(IN NP(NN))))
where bloating occurs?	S(NP(NN)VP(VBZ PP(IN NP(NN))))
where is costa rica?	S(NP(FW FW)VP(VBZ PP(IN NP(NN))))
where the sidewalk ends?	S(NP(DT NN)VP(VBZ PP(IN NP(NN))))
where did the mound builders live?	S(NP(DT NN NS)VP(VBD PP(IN NP(NN))))
where have all the flowers gone?	S(NP(PDT DT NNS)VP(VBP VP(VBN S(VP(TO VP(VB))))))
where does lightning begin?	S(NP(NN)VP(VBZ PP(IN NP(NN))))
where to get steroids?	S(VP(VB NP(NNS)PP(IN NP(NN))))
where can i buy lg washer?	S(VP(VB NP(NN NN)PP(IN NP(NN))))
where are they now?	S(NP(PRP)VP(VBP PP(IN NP(NN))ADVP(RB)))
where to meet cowboys?	S(VP(VB NP(NNS)PP(IN NP(NN))))
where does zac efron live?	S(NP(NN NN)VP(VBZ PP(IN NP(NN))))
what is a trojan?	S(NP(DT NN)VP(VBZ NP(DT NN)))
what year did the battle of trenton take place?	S(NP(NP(DT NN)PP(IN NP(NN))) VP(VBD NP(NP(NN)PP(IN NP(NN NN)))))
what's hot?	S(NP(NN)VP(VBZ ADJP(JJ)))
what does invoke mean?	S(NP(JJ NNS)VP(VBP))
what to bring to your road test?	S(VP(VB NP(NN)PP(TO NP(PRP$ NN NN))))
what is a state of emergency?	S(NP(NN)VP(VBZ NP(NP(DT NN)PP(IN NP(NN)))))
what to put on a wedding registry?	S(NP(NN)VP(VBP PP(IN NP(DT NN NN))))
what can ferret eat?	S(NP(NNP)VP(MD VP(VB NP(NN))))
what does an effective school psychologist do?	S(NP(DT JJ NN NN)VP(VBZ NP(NN)))
what is good for stress?	S(NP(NN)VP(VBZ ADJP(JJ PP(IN NP(NN)))))
what the bible says about gossip?	S(NP(DT NN)VP(VBZ NP(NP(NN)PP(IN NP(NN)))))
what does the giant squid eat?	S(NP(DT JJ NN)VP(VBP NP(NN)))

Table A.2: List of *high*, *medium* and *low* frequency labels used in building FB queries.

high	`DT, IN, JJ, NN, NNP, NNS, NP, PP, VP, S`
medium	`ADJP, ADVP, CC, CD, MD, NP-TMP, POS PRP, PRP$` `QP, RB, SBAR, TO, VB, VBD, VBG, VBN, VBP, VBZ`
low	`CONJP, EX, FRAG, FW, JJR, JJS, NNPS, NX, PDT, PRN, PRT` `RBR, RBS, RP, SINV, UCP, WHADVP, WHNP, WHPP, WP, WRB`

Table A.3: Summary of the query runtimes for *who* parsed queries

			1			**2**			**3**			**4**			**5**		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q1	0	20	**10.6**	45.6	55.7	**7.9**	12.7	33.6	**1.7**	5.9	16.2	**0.5**	1.3	5.6	**0.4**	0.7	3.7
q2	8	8	**17.4**	17.6	21.3	16.3	**3.8**	12.22	2.3	**0.5**	2.4	0.12	**0.04**	0.3	0.15	**0.06**	0.3
q3	1819	7	53.2	**20.1**	23.8	51.8	**3.7**	11.9	10.2	**0.7**	3.9	4.7	**0.2**	1.7	3.6	**0.16**	1.5
q4	28	13	**13.3**	23.9	28.9	9.3	**6.4**	16.5	5.4	**1.2**	6.4	**0.5**	0.6	2.9	**0.25**	**0.25**	0.7
q5	0	21	**42.9**	50.8	62.4	37.5	**18.2**	33.8	**3.4**	5.5	15.9	**0.5**	1.5	7.2	**0.34**	0.92	5.5
q6	94	13	51.5	**36.5**	44.4	39.8	**9.4**	25.6	12.9	**3.4**	12.1	8.5	**0.8**	9	1.8	**0.5**	4.9
q7	0	8	**1.6**	13	16.4	**0.7**	1.7	5.7	**0.2**	**0.2**	1.7	**0.03**	**0.03**	0.08	**0.04**	**0.04**	0.1
q8	210	12	**26.5**	28	33.7	23.4	**6.9**	20.2	10.8	**2.7**	10.2	2.9	**0.7**	3.9	1.7	**0.4**	3.5
q9	859	7	42.6	**16.8**	20.5	40.9	**3.1**	9.2	6.1	**0.5**	3.2	2.3	**0.2**	1.4	1.8	**0.1**	1.2
q10	52	8	22.1	**17.8**	21.9	20.8	**3.5**	8	4	**0.6**	3	0.2	**0.05**	0.4	0.2	**0.06**	0.5
q11	1819	7	52.9	**19.8**	23.9	51.2	**3.7**	11.7	10.1	**0.7**	3.8	4.6	**0.2**	1.7	3.5	**0.2**	1.5
q12	107	8	22.1	**19.3**	24.1	19.8	**4.4**	14	8.6	**0.6**	3.7	1.9	**0.2**	2.6	0.3	**0.06**	0.5

Table A.4: Summary of the query runtimes for *which* parsed queries

			1			**2**			**3**			**4**			**5**		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q13	0	13	**2.9**	20.3	24.8	**1.3**	4	11.6	**0.5**	0.8	5	**0.2**	0.3	1.5	**0.1**	**0.1**	1
q14	0	11	**3.5**	15.7	19.9	**1.7**	2.9	5.5	**1.1**	3	8.5	**0.2**	0.6	1.3	0.07	**0.06**	0.6
q15	8	11	49.8	**29.3**	36.3	47.5	**9.4**	18.2	9.5	**2.5**	11.2	1.9	**1.3**	9.4	**0.1**	**0.1**	1
q16	0	16	**3.9**	24.3	30.7	**2**	6.6	15.7	**0.6**	2.1	8	**0.3**	1.5	3.1	**0.2**	0.3	2.2
q17	22	11	**15.2**	21.5	26.5	13	**5.8**	17.6	1.8	**1.2**	4.5	2.7	**0.3**	2.7	0.3	**0.1**	1.2
q18	103	9	43.1	**19.2**	23.7	38.9	**5.1**	12.3	15.2	**0.9**	5.1	1	**0.2**	2.7	0.4	**0.1**	0.5
q19	0	15	**4.7**	28.6	35.9	**2.9**	7.5	15.9	**1.5**	3.5	10.5	**0.4**	1	2.8	**0.1**	0.2	1.3
q20	0	15	**5.5**	26.7	33.6	**3.6**	5.5	15.7	**0.7**	1.8	6.1	**0.3**	**0.3**	3.2	**0.1**	**0.1**	0.9
q21	0	17	**4.2**	25.5	32.2	**2**	6.9	13.3	**0.5**	1.9	5.6	**0.2**	1.5	2.5	**0.2**	1.1	2.1
q22	0	9	**3.7**	15.5	19.9	**2.1**	4.1	9.8	**1**	2.8	6.3	**0.2**	0.6	1.3	**0.06**	0.1	0.6
q23	0	15	**4.9**	30.2	37.5	**2.2**	6.7	13.7	**1.5**	3.5	11.6	**0.8**	1.9	5.3	**0.4**	0.7	4.7
q24	0	8	**2.8**	14.8	19	**1.4**	3.6	7.7	**0.4**	0.6	2.6	**0.1**	**0.1**	1.2	**0.1**	**0.1**	0.4

Table A.5: Summary of the query runtimes for *where* parsed queries

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q25	127	9	**20.9**	23.5	28.6	19.5	**4.5**	15.8	5.7	**2**	6.7	6.1	**0.4**	5.8	2	**0.2**	2.3
q26	180	9	24.4	**23.3**	28.4	23	**4.8**	15.1	6.4	**2**	5.9	6.9	**0.4**	4.6	2.4	**0.2**	2.1
q27	0	10	**2.1**	17.6	22.0	**1**	3.2	10	**0.5**	1.7	5.8	**0.3**	**0.3**	3.4	**0.1**	**0.1**	1.7
q28	111	10	**22.6**	24.8	30.2	21	**6.5**	18.9	9.8	**2.5**	8.5	4.9	**0.5**	4.2	1.8	**0.2**	2.9
q29	29	11	36.2	**23.2**	28.8	34	**6.4**	19.4	5.5	**2.7**	9.9	2.5	**1**	4.5	0.4	**0.2**	2.3
q30	3	14	**2.8**	15.2	19.4	**1.2**	4.2	8.6	**0.4**	1	4.7	**0.2**	1.1	1.8	**0.2**	0.3	2.1
q31	180	9	24.4	**23.4**	28.3	22.9	**4.8**	15.1	6.4	**2**	5.8	6.9	**0.4**	4.6	2.3	**0.2**	2.1
q32	23	9	33.4	**20.3**	24.9	31.8	**4.2**	13	**0.8**	1.3	4.9	0.9	**0.3**	3.3	**0.2**	**0.2**	1.7
q33	7	10	40.8	**22.7**	27.7	34	**5.1**	14.2	**0.9**	2.1	5.5	2.4	**0.5**	3.9	**0.2**	**0.2**	2.1
q34	2	11	**4.1**	18	22.9	**2.8**	3.6	10.6	**1.2**	1.9	5.9	0.5	**0.3**	3.1	0.2	**0.1**	1.9
q35	23	9	33.6	**20.4**	24.9	31.9	4.2	12.8	**0.8**	1.3	4.9	0.9	**0.3**	3.3	**0.2**	**0.2**	1.7
q36	35	10	**22.3**	22.6	27.6	20.5	**5.1**	12.4	5.5	**2.1**	7.1	1.6	**0.3**	3.7	0.6	**0.2**	2.4

Table A.6: Summary of the query runtimes for *what* parsed queries

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q37	65	9	22.7	**19.5**	23.6	21.4	**6.6**	18.7	17.8	**1.3**	5.7	1.2	**0.3**	2.2	0.3	**0.1**	2.3
q38	1	19	**48.6**	49.5	59.9	36.4	**14.3**	28.9	12	**5.8**	18.4	**0.5**	1.4	7.9	**0.5**	0.9	2.1
q39	450	7	**7.4**	11.7	14.1	5.9	**2.3**	6.9	1.9	**0.3**	1.9	1.4	**0.2**	1.3	0.9	**0.1**	1.7
q40	1557	6	12.9	**6.8**	8.7	11.6	**2.6**	6.8	7.9	**0.3**	1.6	2.4	**0.1**	0.8	2.1	**0.1**	2.9
q41	0	11	**10.5**	17.7	21.8	4	**3.7**	8.9	**0.3**	0.6	2.7	**0.1**	0.2	0.8	**0.1**	0.2	2.3
q42	56	13	**24.7**	35.5	42.7	19.4	**8.8**	24.6	6.7	**3.3**	10.6	4.2	**0.8**	7.6	1.1	**0.5**	2.1
q43	4	9	**15.1**	21.5	26	13.5	**6.2**	14.7	**1.5**	2	5.7	**0.3**	0.4	1.9	**0.1**	**0.1**	2.1
q44	558	9	16.8	**20.7**	24.9	15.4	**3.1**	12.2	4.5	**0.8**	4.7	3.1	**0.3**	3.3	1.9	**0.2**	1.7
q45	7	10	**18.7**	19.2	23.5	16.6	**4**	10.7	2.2	**0.8**	4.1	**0.2**	0.3	1.1	0.2	**0.1**	2.1
q46	18	11	**8.4**	23.8	28.7	6.4	**4.9**	15.1	**1.2**	1.9	3.9	1.1	**0.4**	3.8	**0.1**	**0.1**	1.9
q47	76	13	**23.9**	32.6	39.9	18.9	**8.6**	23.9	10.6	**3**	13.8	2.5	**1**	6.2	0.9	**0.8**	1.7
q48	3	9	**12.5**	17.9	22	11	**5.1**	15.7	**1.2**	1.3	3.3	**0.6**	0.8	2.8	**0.1**	**0.1**	2.8

Table A.7: Summary of the query runtimes for parsed queries over *low* frequency labels

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q1	9136	1	7.6	**0.2**	**0.2**	7.7	**0.2**	**0.2**	7.6	**0.2**	**0.2**	7.7	**0.2**	**0.2**	7.6	**0.2**	**0.2**
q2	5652	2	4.3	**0.2**	**0.2**	4.3	**0.1**	0.2	4.3	**0.1**	0.2	4.3	**0.1**	0.2	4.3	**0.1**	0.2
q3	9136	1	7.6	**0.2**	**0.2**	7.7	**0.2**	**0.2**	7.6	**0.2**	**0.2**	7.7	**0.2**	**0.2**	7.6	**0.2**	**0.2**
q4	4330	1	3.4	**0.1**	**0.1**	3.4	**0.1**	**0.1**	3.4	**0.1**	**0.1**	3.4	**0.1**	**0.1**	3.4	**0.1**	**0.1**
q5	6745	1	4.4	**0.1**	**0.1**	4.4	**0.1**	0.2	4.4	**0.1**	0.2	4.4	**0.1**	0.2	4.4	**0.2**	**0.2**
q6	981	1	0.2	**0.02**	**0.02**	0.2	**0.03**	**0.03**	0.2	**0.03**	**0.03**	0.2	0.04	**0.03**	0.2	0.04	**0.03**
q7	679	1	0.5	**0.02**	**0.02**	0.5	**0.02**	**0.02**	0.5	**0.02**	**0.02**	0.5	**0.03**	**0.03**	0.5	**0.03**	**0.03**
q8	981	1	0.2	**0.02**	**0.02**	0.2	**0.02**	**0.02**	0.2	**0.02**	**0.02**	0.2	**0.02**	**0.02**	0.2	**0.02**	0.03
q9	2356	2	2	**0.1**	**0.1**	2	**0.06**	0.09	1.9	**0.06**	0.09	2	**0.08**	0.09	1.9	**0.06**	0.09
q10	1209	1	0.9	**0.03**	0.04	0.9	**0.03**	**0.03**	0.9	**0.03**	0.04	0.9	0.04	**0.03**	0.9	**0.04**	**0.04**

Table A.8: Summary of the query runtimes for parsed queries over *medium* frequency labels

			1			**2**			**3**			**4**			**5**		
	M	S	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q11	229	3	4.3	**0.4**	0.5	0.2	**0.03**	0.07	0.2	**0.02**	**0.02**	0.2	**0.02**	0.03	0.2	**0.02**	0.03
q12	2840	2	4.9	**0.4**	0.4	2.1	**0.06**	0.1	2.1	**0.07**	0.1	2.1	**0.08**	0.1	2.1	**0.08**	0.1
q13	35012	1	26.6	**0.7**	**0.7**	27	**0.7**	**0.7**	26.7	**0.7**	**0.7**	26.8	**0.7**	**0.7**	26.5	**0.7**	**0.7**
q14	34357	1	26	**0.6**	0.7	26.3	**0.6**	0.7	26.2	**0.7**	**0.7**	26.2	**0.7**	**0.7**	25.7	**0.7**	**0.7**
q15	33828	1	25.7	**0.6**	0.7	25.9	**0.6**	0.7	25.9	**0.6**	0.7	25.9	**0.6**	0.7	25.4	**0.7**	**0.7**
q16	22526	1	15.1	**0.4**	**0.4**	15.3	**0.4**	**0.4**	15.2	**0.4**	0.5	15.3	**0.4**	**0.4**	15.1	**0.4**	**0.4**
q17	346	2	22.31	**0.6**	0.7	0.5	**0.01**	0.02	0.5	**0.01**	0.03	0.5	**0.02**	0.03	0.5	**0.02**	0.04
q18	49512	1	33.9	**0.9**	1	34.3	**0.9**	1	34.0	**0.9**	1	34.3	**0.9**	1	33.8	**0.9**	1
q19	6	3	11.4	**0.5**	0.6	0.3	**0.02**	0.03	0.01	**0.01**	**0.01**	0.02	**0.01**	**0.01**	0.02	0.02	**0.01**
q20	35973	1	27.8	**0.7**	**0.7**	28.2	**0.7**	**0.7**	28.0	**0.7**	**0.7**	28.1	**0.7**	**0.7**	27.6	**0.7**	**0.7**

Table A.9: Summary of the query runtimes for parsed queries over *medium* and *low* frequency labels

			1			**2**			**3**			**4**			**5**		
	M	S	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q21	23571	1	16.4	**0.4**	**0.4**	16.6	**0.4**	0.5	16.6	**0.4**	0.5	16.6	**0.4**	0.5	16.6	**0.4**	0.5
q22	981	1	0.2	**0.02**	**0.02**	0.2	**0.02**	**0.02**	0.2	**0.02**	**0.02**	0.2	**0.02**	0.03	0.2	**0.02**	0.03
q23	15178	1	10.5	**0.3**	**0.3**	10.6	**0.3**	**0.3**	10.6	**0.3**	**0.3**	10.6	**0.3**	**0.3**	10.6	**0.3**	**0.3**
q24	22305	2	16.1	**0.8**	0.9	16.0	**0.4**	0.7	16.0	**0.4**	0.7	16.0	**0.4**	0.7	16.0	**0.4**	0.7
q25	22305	2	16.1	**0.8**	0.9	16.0	**0.4**	0.7	16.0	**0.4**	0.7	16.0	**0.4**	0.7	16.0	**0.4**	0.7
q26	42535	1	26.1	**0.9**	1.0	26.5	**0.9**	1.0	26.5	**0.9**	1.0	26.5	**0.9**	1.0	26.5	**0.9**	1.0
q27	34	3	7.0	**0.8**	1.0	0.3	**0.1**	0.3	0.03	**0.01**	**0.01**	0.03	**0.01**	**0.01**	0.04	**0.01**	**0.01**
q28	22526	1	15.1	**0.4**	**0.4**	15.3	**0.4**	**0.4**	15.3	**0.4**	**0.4**	15.3	**0.4**	**0.4**	15.3	**0.4**	**0.4**
q29	9250	3	8.0	**1.2**	1.4	7.9	**0.8**	1.0	5.9	**0.2**	0.5	5.9	**0.2**	0.5	5.9	**0.2**	0.5
q30	75	3	4.4	**0.5**	0.6	4.4	**0.3**	0.4	0.1	**0.01**	**0.01**	0.1	**0.01**	**0.01**	0.1	**0.01**	**0.01**

Table A.10: Summary of the query runtimes for parsed queries over *hight* frequency labels

			1			**2**			**3**			**4**			**5**		
	M	S	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q31	72830	2	68.3	**6.3**	7.4	68.3	**1.4**	2.5	68.3	**1.4**	2.5	68.3	**1.4**	2.5	68.3	**1.4**	2.5
q32	952	8	55.8	**20.8**	25.4	46.5	**4.1**	9.9	7.7	**1.1**	10.7	4.9	**0.2**	5.3	2.5	**0.2**	5
q33	18	10	45.0	**23.2**	28.7	35.6	**6.7**	15.9	8.5	**1.4**	14.4	0.5	**0.4**	4.3	0.06	**0.2**	0.3
q34	6334	3	55.7	**5.1**	6.1	8.3	**1.1**	2.8	6.7	**0.1**	0.3	6.7	**0.1**	0.3	6.7	**0.1**	0.3
q35	899	7	50.8	**26.5**	32	48.8	**5.1**	16.4	9.6	**2.4**	7.0	8.7	**0.6**	5.1	1.8	**0.1**	0.8
q36	72830	2	68.5	**6.1**	7.3	68.3	**1.4**	2.5	68.3	**1.4**	2.5	68.3	**1.4**	2.5	68.3	**1.4**	2.5
q37	1738	8	56.6	**17.1**	21.1	44.6	**2.7**	4.9	21.8	**2.5**	6.0	21.6	**1.7**	3.1	3.5	**0.2**	32.6
q38	14	9	53.5	**22.7**	27.8	52.5	**7.2**	14	1.4	**2.6**	4.3	14.5	**1.1**	18.9	0.9	**0.1**	9.5
q39	183	7	49.3	**14.8**	18.3	8.3	**3.3**	5.3	3.8	**1.9**	3.6	1.2	**1.4**	11.2	0.5	**0.4**	6.1
q40	929	8	57.5	**21.9**	26.6	52.2	**4.7**	9.5	27.7	**2.5**	6.1	7.6	**0.3**	12.8	2.3	**0.2**	8.9

Table A.11: Summary of the query runtimes for parsed queries over *high* an *low* frequency labels

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q41	3555	6	55.9	**13.2**	16.1	46.3	**2.6**	5	17.2	**1.4**	9.3	7.0	**0.3**	6.0	5.4	**0.2**	7.1
q42	651	9	58.6	**25.6**	31.2	46.2	**7.1**	11.9	24.1	**1.8**	13.7	6.6	**0.5**	7.6	3.7	**0.2**	5.6
q43	292	10	62.1	**26.9**	33	56.4	**10.1**	17.9	13.8	**2.9**	8.5	3.8	**0.8**	5.7	2	**0.3**	3.5
q44	4830	5	51.8	**11.1**	13.7	43.1	**2.1**	5.3	12.5	**1.2**	2.2	7.1	**0.3**	1.9	6.1	**0.1**	0.7
q45	74296	1	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8
q46	32240	3	65.6	**10.9**	13	60.6	**2.3**	5.7	34.9	**0.5**	1.9	34.9	**0.5**	1.9	34.9	**0.5**	1.9
q47	1753	7	60.5	**15.6**	19.1	56.1	**4.2**	9.1	14.3	**1.0**	5.7	5.2	**0.7**	2.4	2.6	**0.5**	1.4
q48	28	10	50.3	**24.4**	29.8	47.4	**6.2**	10.3	14.2	**1.8**	4.5	5.2	**1.5**	2.7	4.8	**1.5**	2.2
q49	1116	8	57.1	**22.0**	26.3	8.9	**6.0**	13	5.5	**2.8**	9.4	3.2	**2.2**	2.7	2.2	**0.8**	3.7
q50	72830	2	68.5	**6.1**	7.2	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5

Table A.12: Summary of the query runtimes for parsed queries over *high* and *medium* frequency labels

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q51	1906	8	52.4	**17.8**	21.7	48.4	**4.1**	9	25.7	**3.6**	6.0	8.1	**0.9**	13.1	11.2	**0.5**	9.0
q52	121	8	93.5	**20.8**	25.4	19.5	**4.2**	10.8	2.2	**0.8**	3.4	0.6	**0.4**	2.1	0.1	**0.1**	0.1
q53	74296	1	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8
q54	34	7	44.7	**10.9**	13.8	35.8	**2.2**	5.2	0.9	**0.8**	1.8	0.2	**0.2**	0.5	0.3	**0.6**	1.0
q55	1233	6	47.2	**10.8**	13.2	45.9	**3.5**	6.5	17.4	**1.6**	2.7	23.4	**0.9**	16.5	7.4	**0.6**	3.0
q56	74296	1	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8
q57	357	8	49.7	**13.3**	16.2	9.2	**3.1**	6.1	8.9	**2.3**	5.2	7.4	**1.4**	2.3	1.6	**0.8**	30.6
q58	72830	2	68.5	**6.1**	7.3	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5
q59	16257	4	57.7	**9.6**	11.5	56.5	**3.9**	10.5	22.3	**1.0**	3.9	17.3	**0.3**	1.4	17.3	**0.3**	3.7
q60	966	9	54.4	**26.5**	31.7	49.8	**6.8**	17.4	15.3	**2.7**	8.5	4.9	**0.5**	5.6	3.2	**0.4**	3.5

Table A.13: Summary of the query runtimes for parsed queries over *high*, *medium* and *low* frequency labels

			1			2			3			4			5		
	M	**S**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**	**f**	**r**	**s**
q61	8266	5	53.9	**10.4**	12.6	52.9	**2.3**	3.8	52.3	**2.8**	3.9	25.5	**1.0**	16.5	10.3	**0.2**	7.9
q62	7650	2	56.5	**2.3**	2.9	7.7	**0.1**	0.2	7.7	**0.1**	0.2	7.7	**0.1**	0.2	7.7	**0.1**	0.2
q63	68541	2	59.5	**8.4**	9.7	58.9	**1.5**	4.6	58.9	**1.5**	4.6	58.9	**1.5**	4.6	58.9	**1.5**	4.6
q64	70973	2	61.7	**9.9**	11.8	57.6	**1.6**	3.5	57.6	**1.6**	3.5	57.6	**1.6**	3.5	57.6	**1.6**	3.5
q65	74296	1	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8	49.1	**1.6**	1.8
q66	20539	5	51.6	**13.2**	16.1	49.8	**3.4**	8.6	32.6	**1.2**	5.7	25.7	**1.1**	4.3	24.3	**0.4**	2.1
q67	89451	1	49.8	**3.8**	4.3	49.8	**3.8**	4.3	49.8	**3.8**	4.3	49.8	**3.8**	4.3	49.8	**3.8**	4.3
q68	3478	7	48.7	**14.2**	17.4	45.5	**2.7**	5.1	45.3	**3.5**	5.2	45.7	**3.5**	5.1	44.9	**3.5**	5.1
q69	72830	2	68.5	**6.1**	7.3	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5	67.8	**1.4**	2.5
q70	31564	4	52.2	**12.7**	15.0	51.7	**2.5**	8.8	36.2	**1.9**	2.9	35.1	**0.6**	3.4	34.1	**0.6**	3.4

Appendix B

Proof of Lemmata and Theorems

In this chapter, we provide the proof of a few of the lemmata and theorems presented in Chapter 5.

Lemma (Body of Lemma 5.2.1)**.** *For any two index keys s_1 and s_2 over a given SI, where $s_1 \precsim s_2$, we have*

(i) The posting list of s_2 is always a subset of the posting list of s_1 for filter-based coding.

(ii) The posting list of s_2 is a subset of the posting list of s_1 for root-split coding if and only if s_1 and s_2 share the same root.

(iii) The posting list of s_2 is not guaranteed to be a subset of the posting list of s_1 for subtree interval coding.

Proof of Lemma 5.2.1. The proof is based on the structure of the three proposed codings.

(i) In filter-based coding only the tid values are stored. If there exists a tree t_k in the posting list of s_2 (i.e. $s_2 \precsim t_k$), since $s_1 \precsim s_2$, then we have $s_1 \precsim t_k$. The subtree relationship, $\precsim$, similar to subset relationship is transitive.

(ii) For root-split coding, when $s_1 \precsim s_2$ and s_1 and s_2 share the same root, if there exists a tuple $< t_k, l_k, r_k, v_k >$ in the posting list of s_2, since $s_1 \precsim s_2$, it will be encoded using the same interval codings for its root and $< t_k, l_k, r_k, v_k >$ exists in its posting list. Therefore, s_2's posting list is a subset of s_1's.

(iii) For subtree interval coding, we prove using a counter example. Assume we are given a SI with $mss = 2$ which has only the following tree indexed `NP(NN)(NN)(NN)`. Apparently, `NP` $\precsim$ `NP(NN)`, however, there are three entries in the posting list size of `NP(NN)`, while there is only one entry in the posting list size of `NP`, which proves that the posting list of subtrees is not guaranteed to be supersets.

□

Theorem (Body of Theorem 5.2.3)**.** *Given a query Q and a subtree index with root-split coding and maximum subtree size mss, an optimal query plan for Q cannot have a subtree of size less than mss.*

Proof of Theorem 5.2.3. Assume by contradiction that there exists an optimal query plan some of whose leaves have a smaller size than mss, we denote this plan by P^*_{opt}. We build a plan P_{opt} by growing the subtrees s_i at leaves of P^*_{opt} whose sizes are less than mss as follows. We randomly add nodes from Q to s_i in such a way that s_i holds its property of being a subtree of Q and stop when $|s_i| = mss$. Thus, each subtree at a leaf of P^*_{opt} is a subtree of a subtree at a leaf of P_{opt}. By Lemma 5.2.1 and Lemma 5.2.2 we have that the selectivities of leaves of P^*_{opt} are at most as high as those of P_{opt} and therefore, P^*_{opt} cannot be an optimal plan. □

Theorem (Body of Theorem 5.2.4)**.** *For every query Q and size mss such that $|Q| \geq mss$, there exists a root-split max-cover C; i.e. for every subtree $c \in C$ we have $|c| = mss$.*

Proof of Theorem 5.2.4. We prove by induction that $minRC$ algorithm in Figure 5.9 computes a max-cover.
Base. At the base of this algorithm, lines $7-9$, $minRC$ assigns subtrees of size mss by calling $assign$ on Q. The $assign$ algorithm always returns a subtree of size equal to mss, and therefore, base of the induction is proved.
Induction. Assume that $minRC$ generates root-split max-covers for all children of Q with size larger than mss. All children with size equal to mss will also be immediately added to the cover by lines 3 and 4 of the algorithm. Thus, if we prove that Q is also covered using a subtree of size mss, the induction is proved. Once all Q's children get covered, Q will be covered by a call to $assign$ which is guaranteed to return a subtree of size mss. Thus, our theorem is proved. □

Theorem (Body of Theorem 5.2.11)**.** *$optimalCover$ returns a join optimal cover if (1) $mss \leq 6$ and (2) injective matching is not assumed.*

Proof of Theorem 5.2.11. We assume that $|Q| \geq mss$, otherwise, Q can be covered using a single subtree, which is obviously join optimal. $optimalCover$ starts from the root of Q. For each child c of Q, we have one of the following three cases, (1) $|c| < mss$, (2) $|c| = mss$, and (3) $|c| > mss$. Case (1) is handled by $assign$ algorithm which we showed join optimality in Lemma 5.2.10. Case (2) is directly assigned into an individual subtree partition at line 3 of the $optimalCover$. Finally, case (3) is handled by recursive calls of $optimalCover$ until either of cases (1) or (2) occur. Over internal nodes with condition of case (1), as soon as enough of their children are assign and their remaining size reduces to less than mss, $optimalCover$ returns and leaves their handling to the ancestor which satisfies case (1). As a result $optimalCover$ achieves a globally join optimal cover over Q. □

Theorem (Body of Theorem 5.2.13)**.** *$minRC$ returns the smallest root-split cover possible if (1) $mss \leq 6$ and (2) injective matching is not assumed.*

Proof of Theorem 5.2.13. $minRC$ handles internal nodes that fall into case (1) of the proof in Theorem 5.2.11 different from $optimalCover$. To avoid deep branching anomaly, it requires that each internal node is assigned to a subtree, before any of its ancestors are assigned. As a result, there are cases where $minRC$ does not achieve optimality. However, since all root-split covers have to handle deep branching anomaly, $minRC$ achieves the smallest cover possible among them, by repeatedly calling $assign$ on non-assigned subtrees, which was shown to be optimal. □

Theorem (Body of Theorem 5.3.3)**.** *Given a query Q, if there exist subtrees s_B and s_P of Q such that s_P hides s_B, then there exist covers over Q which do not guarantee a correct set of results if an injective matching is required.*

Proof of Theorem 5.3.3. We prove by showing how to build a cover which violates the injective matching assumption. Since s_P hides s_B, any tree that matches s_B, matches s_P as well. Therefore, if a cover requires s_B and s_P to participate in the same join (e.g. a parent-child join with their shared parent), any minimal tree that matches the subtree containing s_P and its parent, and for which there exist only a single node matching at least one node of s_B, will be a wrong match for Q. The reason is that Q requires distinct pair of nodes matching nodes in s_B and their corresponding nodes in s_P to guarantee injective matching. Therefore, such a cover will violate the injective matching property. □

Appendix C

Supplementary Algorithms

A few supplementary algorithms are covered in this chapter.

subtrees algorithm This algorithm extracts all unique subtrees of size n from an input tree t. The base of this algorithm, lines 1 to 3, solves for the case where $n \leq 1$ and therefore, only one subtree is possible. For larger n, line 4 computes a vector of vectors, which correspond to the different combinations $n-1$ nodes can be selected from children of t, to make subtrees of size n over t. The rest of the algorithm computes all combinations of subtrees whose sizes have been computed in cs over children of t and adds them as children of t. For example, if $n = 4$ and t has three children with sizes 4, 2 and 1, Then cs will look like $\{[3, 0, 0], [2, 1, 0], [2, 0, 1], [1, 2, 0], [1, 1, 1], [0, 2, 1]\}$. For each element of cs which is a vector of numbers, $mult$ stores the possible number of combinations possible. For instance for $[3, 0, 0]$, there are at most 3 possible combinations, as all the 3 nodes will be selected from child of t with size 4. In the worst case, this child has a root with 3 children which leads to 3 combinations of different possible subtrees of size 4 over t, using $[3, 0, 0]$. This process will be repeated over all elements of cs until all subtrees have been extracted.

combinations algorithm Given a vector of trees vt and a numbering value n, this algorithm computes the different combinations of sizes of trees in vt that add up to n. It works by assigning $vt[c]$ from 0 to $\min(n, |vt[c]|)$ and for each assignment computing the different combinations over $vt[1], \ldots, vt[c-1]$, recursively.

```
$\mathbf{subtrees(t, n)}$
**if** $n = 0$ **return** $\emptyset$
$rTree = t.root$, $resVec \leftarrow rTree$
**if** $n = 1$ **return** $resVec$
$cs = \mathbf{combinations(t.children, |t.children| - 1, n - 1)}$
**for** $i \in 1, \ldots, |cs|$
    $TC \leftarrow \emptyset$, $mult = 1$
    **for** $j \in 1, \ldots, |cs[i]|$
        $T \leftarrow \mathbf{subtrees(t.children[j], cs[i][j])}$
        $TC \leftarrow TC \cup T$
        **if** $|T| > 0$
            $mult = mult * |T|$
    **for** $j \in 1, \ldots, mult$
        $val = j$
        **for** $k \in 1, \ldots, |TC|$
            $m = val\%|TC[k]|$
            $rTree.children \leftarrow rTree.children \cup TC[k][m]$
            $val = val/|TC[k]|$
        $resVec \leftarrow resVec \cup rTree$
**return** $resVec$
```

Figure C.1: Algorithm for extracting subtrees of size n rooted at t.

```
$\mathbf{combinations(vt, c, n)}$
$sum = 0$
**for** $i \in 1, \ldots, |vt|$
    $sum = sum + |vt[i]|$
$temp \leftarrow \emptyset$, $res \leftarrow \emptyset$
**if** $c = 0$
    **if** $sum < n$ **return** $res$
    **else** $temp \leftarrow n$, $res \leftarrow temp$, **return** $res$
**if** $sum < n$ **return** $res$
$TR \leftarrow \emptyset$
**for** $j \in 0, \ldots, \min(n, |vt[c]|)$
    $TR \leftarrow \mathbf{combinations(vt, c - 1, n - j)}$
    **for** $k \in 1, \ldots, |TR|$
        $TR[k] \leftarrow TR[k] \cup j$
        $res \leftarrow res \cup TR[k]$
**return** $res$
```

Figure C.2: Algorithm that computes all the combinations of sizes of children that lead to a given subtree size