TOWARDS EFFICIENT DATA ANALYSIS AND MANAGEMENT OF SEMI-STRUCTURED DATA

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

By
Shirish Tatikonda, B.E., M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Dissertation Committee:
Prof. Srinivasan Parthasarathy, Advisor
Prof. P. Sadayappan
Prof. Xiaodong Zhang
ABSTRACT

Over the last decade, there has been an enormous growth in both the amount and the complexity of online content that is collected and processed by humans and machines. Such a growth has spurred interest in flexible and fluid (semi-structured) data models that do not constrain the data to follow a fixed schema. Many applications ranging from bioinformatics to XML repositories, from software engineering to computational linguistics, are now generating and processing large amounts of semi-structured data. For these applications to reach their full potential, we need to build an effective set of tools to index, process, manage, and analyze such data. This dissertation focuses on a specific class of semi-structured data that is denoted using hierarchical tree objects. We specifically address the following questions pertaining to mining and managing tree-structured data: How can we provide quick access mechanisms to large semi-structured data stores? How can we discover hidden structural patterns from such data collections? How can we devise strategies to realize performance that is commensurate with modern computer architectures?

In the context of managing tree-structured data, first, we develop an indexing mechanism that extracts discriminant features from the database and indexes them using a simple tunable inverted structure. Such an index is complemented with an efficient holistic query processing technique that retrieves the matches by operating entirely on space-efficient sequential representation of trees. Second, we propose a framework that enables the development of application-specific hash functions that convert variable-sized graph and tree structured data into fixed-sized hash values. We demonstrate the usability of this framework by developing a hash-based distributed data placement service for semi-structured data. We argue that this service is capable of supporting large scale data management and data mining algorithms.

In the context of mining tree databases, first, we explore the role of succinct sequential data structures for efficiently discovering frequent tree patterns. Second, we propose a “memory-conscious design” wherein the algorithms trade memory for
redundant computations to improve the memory system performance. Third, we consider the case of deploying data mining workloads on modern multicore systems. Here, we demonstrate that the bandwidth to main memory becomes a precious shared commodity as one increases the number of cores present in the system. We present mechanisms to alleviate the bandwidth pressure and show their effectiveness. Fourth, we explore an adaptive task and data parallel algorithm design that facilitates effective parallelization in the presence of data and workload skew. This algorithm is integrated into a general purpose scheduling service that supports the development of adaptive and moldable algorithms for database and mining tasks. Finally, we develop a hash-based distributed data placement service that can support the development of large scale distributed data mining and data management applications.
Dedicated to my parents and my brother
ACKNOWLEDGMENTS

I doubt if I can properly express my gratitude to my adviser, Prof. Srinivasan Parthasarathy. I owe so much to him. He has been an incredible source of inspiration and encouragement. He allowed me to explore different research topics, helped me think outside the box, taught me effective technical communication skills, motivated me in difficult times, and always led me in successful and impactful research path. I have learned a lot under his tutelage.

I would like to thank Prof. P. Sadayappan and Prof. Xiaodong Zhang for serving on my dissertation committee, and Dr. Hakan Ferhatosmanoglu for being part of my candidacy examination. Interactions with Prof. Sadayappan have always been very insightful and thought-provoking. Every time I had a chance to discuss with him, I gained a great deal of knowledge. I thank Dr. Tahsin Kurc for his constructive suggestions and comments during the years of our collaboration. During my internship stint at Yahoo! Research, I had an opportunity to work with outstanding researchers, Dr. Flavio Junqueira, Dr. B. Barla Cambazoglu, and Dr. Vassilis Plachouras. They have made my stay at beautiful Barcelona incredibly memorable and highly productive.

I am thankful to the National Science Foundation for supporting my research at Ohio State through grants CAREER-IIS-0347662, NGS-CNS-0406386, CAREER-IIS-0347662, RI-CNS-0403342, CCF-0702587, and IIS-0917070. Any opinions, findings, and conclusions or recommendations expressed here are those of the author and, if applicable, his adviser and collaborators, and do not necessarily reflect the views of the National Science Foundation.

I am very fortunate to meet my dearest friends Muthu, Shantanu, Ramya, and Bea at OSU. Life in graduate school would have been impossible without them. They played a vital role in every success I had at OSU. I thank them for making my long and rocky road to PhD alot smoother. They were always with me in both good and bad times. Bea’s lovely pet, Yoghurt needs a special mention for helping me relax after having long days at work.
I would like to thank several present and past members of data mining research laboratory for making my stay in the department very pleasant. In alphabetical order, they are Amol, Chao, Duygu, Greg (who introduced me to a wonderful tool, *valgrind*), Huai, Hui, Keith, Mahashweta, Matt Goyder, Matt Otey, Sameep, Sitaram, Venu, Xintian, and Ye. Their comments and suggestions during group meetings and practice presentations were very helpful. I would also like to thank my other good friends whom I met here at OSU – Chaitanya, Bhaskar & Sridevi, Uday, Qian, Brian, Jim, Seetha, and Gayathri. A very special thanks to my old friend Jaggu who inspired me to pursue a Ph.D. degree, and to all my other friends who always kept me running towards the goal with the question, “are you done yet?”.

None of this work would have been ever possible without the continuous and unconditional love and support from my parents, Radha Kishan and Chandrakala. Thanks to my father who had always given education its due importance, and gave me a chance to attend best schools throughout my career. Without my mother’s invaluable support, I would have never achieved anything in my life. I wish to thank my brother Harish for his invaluable advice at every stage in my career. My parents and my brother have been three pillars of my life. I hope by dedicating this dissertation I sufficiently express my deepest gratitude to them. I am thankful to my other family members my sister Sarita, her husband Prasad, my sister-in-law Deepa and all my nieces for their support and encouragement over the years. Last, but not the least, I want to thank God Almighty for giving me the knowledge, strength, energy, and confidence to gain the achievements I have.
VITA

Aug 1997-June 2002 ............ B.E. Computer Science and M.S. Economics
   Birla Institute of Technology and Science (BITS),
   Pilani, India

Jul 2002-Jul 2004 ............... Applications Engineer
   Oracle Corporation, Hyderabad, India

Sep 2004-present ............... Graduate Research Associate
   Computer Science and Engineering
   The Ohio State University

Oct 2008-Dec 2008 ............... Research Intern
   Yahoo! Research, Barcelona, Spain

Jun 2008-Sep 2008 ............... Research Intern
   IBM T.J. Watson Research Center, New York, USA

Mar 2007-Jun 2007 ............... Research Intern
   IBM India Research Laboratory, New Delhi, India

RESEARCH PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science and Engineering
TABLE OF CONTENTS

Abstract ................................................................. ii
Dedication .................................................................. iii
Acknowledgments ....................................................... v
Vita ........................................................................... vii
List of Tables ................................................................ xii
List of Figures ................................................................ xiii

CHAPTER PAGE
1 Introduction .......................................................... 1
  1.1 Thesis Statement ............................................... 5
  1.2 Contributions ................................................... 5
    1.2.1 Managing Tree Databases ................................. 6
    1.2.2 Mining Tree Databases ..................................... 7
    1.2.3 Parallel Algorithms ......................................... 8
  1.3 Organization ...................................................... 10
2 Indexing Tree-structured Data ................................. 11
  2.1 Related Work ..................................................... 14
    2.1.1 Path-based Approaches ................................. 15
    2.1.2 TwigStack-style Approaches ......................... 17
    2.1.3 Sequence-based Approaches ......................... 17
  2.2 LCS-TRIM: LCS based TRee Indexing and Matching .... 20
    2.2.1 Data Transformation ..................................... 22
    2.2.2 Tree Matching Algorithm ............................... 26
    2.2.3 Indexing Method .......................................... 33
    2.2.4 Optimizations ............................................. 35
    2.2.5 Putting it all together .................................... 37
  2.3 Empirical Evaluation ............................................ 39
    2.3.1 Comparison with PRIX .................................. 42
    2.3.2 Comparison with TwigStack-style algorithms .... 47
2.3.3 Extensions .............................................. 49
2.4 Conclusions .............................................. 51

3 Hashing Tree-structured Data ................................ 52
  3.1 Sketching a Tree Structure ................................ 54
      3.1.1 Transformation Function (tf) ......................... 54
      3.1.2 Signature Function (sf) .......................... 58
      3.1.3 Different Transformation Functions ................. 59
  3.2 Theoretical Analysis ..................................... 61
      3.2.1 Bounds on Node Level Pivots ......................... 62
      3.2.2 Incremental Construction .......................... 62
      3.2.3 Relation to Perfect Hashing ....................... 63
      3.2.4 Relation to Tree Edit Distance .................... 69
  3.3 Empirical Evaluation .................................... 72
      3.3.1 Case Study I: De-duplication of XML Documents ........ 76
      3.3.2 Case Study II: Phylogenetic analysis ............... 79
  3.4 Related Work ........................................... 80
  3.5 Conclusions ............................................ 81

4 Mining Tree-structured Data ................................ 82
  4.1 Background and Problem Definition ....................... 84
  4.2 Related Work .......................................... 86
      4.2.1 Mining an Ordered Forest ......................... 87
      4.2.2 Mining an Unordered Forest ...................... 90
      4.2.3 Mining a Forest of Free Trees .................... 92
      4.2.4 Other Variants .................................. 93
  4.3 TRIPS and TIDES: New Algorithms for Tree Mining ........ 95
      4.3.1 Data Transformation ................................ 96
      4.3.2 Candidate Generation ................................ 97
      4.3.3 Support Counting .................................. 103
      4.3.4 Putting it All Together ............................ 104
      4.3.5 TIDES: Tree mining using DEpth first Sequences ...... 106
  4.4 Empirical Evaluation .................................... 106
      4.4.1 Evaluation on synthetic data sets .................... 107
      4.4.2 Evaluation on real-world data sets .................. 110
      4.4.3 Comparison of TRIPS and TIDES ..................... 113
      4.4.4 Performance Characterization ....................... 115
  4.5 Conclusions ............................................ 116

5 Parallel Algorithms ........................................ 118
  5.1 Algorithms for Multicore Systems ......................... 121
      5.1.1 The Multicore Challenge .......................... 121
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Tree Indexing – Notation</td>
</tr>
<tr>
<td>2.2</td>
<td>Data sets for evaluating <em>LCS-TRIM</em></td>
</tr>
<tr>
<td>2.3</td>
<td>Swissprot workload</td>
</tr>
<tr>
<td>2.4</td>
<td>Dblp workload</td>
</tr>
<tr>
<td>2.5</td>
<td>Treebank workload</td>
</tr>
<tr>
<td>2.6</td>
<td>Cslogs workload</td>
</tr>
<tr>
<td>2.7</td>
<td>NLM workload</td>
</tr>
<tr>
<td>3.1</td>
<td>Notation</td>
</tr>
<tr>
<td>3.2</td>
<td>Datasets and their characteristics</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of frequent subtree mining algorithms</td>
</tr>
<tr>
<td>4.2</td>
<td>Hash Table for Candidate Subtrees</td>
</tr>
<tr>
<td>4.3</td>
<td>Data sets for evaluating TRIPS</td>
</tr>
<tr>
<td>5.1</td>
<td>Characterization of Tree Mining Algorithms</td>
</tr>
<tr>
<td>5.2</td>
<td>Cost Analysis of Adaptive Parallelization (Treebank)</td>
</tr>
<tr>
<td>5.3</td>
<td>Cost Analysis of Adaptive Parallelization (Cslogs)</td>
</tr>
<tr>
<td>5.4</td>
<td>Parallel TreeMiner – Need for Memory Optimizations</td>
</tr>
<tr>
<td>5.5</td>
<td>Variance in Performance From Different Partitions</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of Distributed Indexing Methods</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Examples of semi-structured data</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Dissertation Overview</td>
<td>6</td>
</tr>
<tr>
<td>2.1 XML documents and XPath query</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Framework of LCS-TRIM</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Comparison of different sequencing methods</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Tree Matching: Dynamic programming matrix and derived matches</td>
<td>33</td>
</tr>
<tr>
<td>2.5 Effect of Tree Matching Optimizations</td>
<td>42</td>
</tr>
<tr>
<td>2.6 Performance Comparison: LCS-TRIM vs. PRIX</td>
<td>44</td>
</tr>
<tr>
<td>2.7 Index Size Comparison</td>
<td>48</td>
</tr>
<tr>
<td>2.8 Performance Comparison: LCS-TRIM vs. TwigStack-style Methods</td>
<td>48</td>
</tr>
<tr>
<td>2.9 Heuristics for Unordered Subtree Matching</td>
<td>50</td>
</tr>
<tr>
<td>3.1 Bounds on Number of Pivots</td>
<td>57</td>
</tr>
<tr>
<td>3.2 Incremental Construction of Pivot Set</td>
<td>62</td>
</tr>
<tr>
<td>3.3 Counter Examples for Perfect Hashing</td>
<td>64</td>
</tr>
<tr>
<td>3.4 Proof of Perfect Hashing</td>
<td>65</td>
</tr>
<tr>
<td>3.5 Structural Equivalence</td>
<td>66</td>
</tr>
<tr>
<td>3.6 Tree Edit Distance – Operations</td>
<td>69</td>
</tr>
<tr>
<td>3.7 Construction cost &amp; Information content in Pivot Multisets</td>
<td>73</td>
</tr>
<tr>
<td>3.8 Approximating Edit Distance</td>
<td>74</td>
</tr>
<tr>
<td>3.9 Case Study on XML Deduplication</td>
<td>77</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>5.16</td>
<td>Performance of data partitioning methods</td>
</tr>
<tr>
<td>5.17</td>
<td>Variance of cost metric across different partitions ( \text{( minsup = 58% )} )</td>
</tr>
<tr>
<td>5.18</td>
<td>Evaluation of Hash-based Sampling Framework</td>
</tr>
<tr>
<td>5.19</td>
<td>Framework for Distributed Data Placement</td>
</tr>
</tbody>
</table>
### LIST OF ALGORITHMS

1. Transformation of Trees into Sequences ........................................ 22
2. Enumerating the set of all subsequences ........................................ 29
3. Algorithm for matching the structure of two sequences .................. 31
4. \textit{LCS-TRIM}: Algorithm for subtree matching ............................. 38
5. Transformation function that constructs embedded pivots .............. 56
6. Similarity between two trees $T_1$ and $T_2$ .................................. 59
7. Tree Sketching (\textit{sketch}) ...................................................... 60
8. Pattern mining with embedding lists ............................................. 103
9. Trips and Tides: Algorithms for frequent subtree mining ................ 105
10. Tree pruning and recoding algorithm .......................................... 127
11. On-the-fly embedding list construction ...................................... 129
12. Memory Conscious Trips (MCT) ............................................... 132
13. Parallel Tree Mining ............................................................. 137
14. Parallel Index Construction ....................................................... 154
15. Prototype interface for scheduling service .................................... 160
16. Stratified partitioning ............................................................. 165
17. Stratification via LSH (\textit{stratify}) .............................................. 166
Advances in technology have enabled us to collect large amounts of data in all walks of human endeavor. Instruments like Large Hadron Collider are capable of producing torrents of experimental data within a short span of time. Many organizations like Walmart have setup data warehouses which host tera-byte and peta-byte scale data sets. The challenge of managing and analyzing these large data collections is an immense task. The field of Knowledge Discovery in Databases (KDD), popularly known as Data Mining, is concerned with automatically extracting interesting and actionable knowledge from these large data sources in as efficient a manner as possible. It embraces ideas from databases, statistics, machine learning, and high-performance computing. Over the last two decades, the field has gone through enormous transformations and has been shown to be useful in many diverse applications. Examples include identifying fraudulent and money laundering activities; analyzing astronomical data to automatically search for new galaxies and stars; developing systems to discover genetic basis for diseases like schizophrenia; detecting intrusions and malicious activities by examining network packet flows; characterizing the origin and evolution of defects in materials; and predicting the speed and direction of disastrous hurricanes.

Efficiency is critical to the knowledge discovery process as it is iterative (repetitive, exploratory) in nature and involves a human-in-the-loop (interactive). Interactivity is often the key to facilitate effective data understanding and knowledge discovery since lengthy time delays between responses of two consecutive user requests can disturb the flow of human perception and formation of insight. Furthermore, efficient and scalable methods are essential to process large data sets. Strategies that improve the efficiency of algorithms can be classified into three main categories: (i) data-driven approaches, which include classical methods like sampling, dimensionality reduction, compression, etc.; (ii) computation driven approaches, which include techniques that intelligently prune the search space, algorithms that compute approximate solutions, and other
algorithmic optimizations; and (iii) architecture-conscious approaches, which follow an orthogonal direction to other existing methods. The goal here is to characterize and understand the features as well as limitations of the underlying computer architecture, and subsequently to re-architect the data mining algorithms to better utilize the system resources.

Since the inception of World Wide Web, there has been an unprecedented growth in both the amount and the complexity of online content that is collected and processed by humans and machines. The extreme simplicity of HTML and HTTP has made it easy to author and exchange information over the web. Such a growth has spurred interest in flexible and fluid (semi-structured) data models that do not constrain the data to follow a rigid or regular schema. The information that is normally associated with a schema is implicitly contained within the data itself. These self-describing models can thus be treated as a middle path between the two extremes – unstructured models with no schema (e.g., text) and structured models with a fixed schema (e.g., RDBMS). The structure that is present in such data is irregular, implicit, partial, and often evolving. It is thus represented using elastic data structures such as graphs and trees.

Semi-structured data models allow diverse entities to communicate and exchange information in a seamless and schema-independent manner. It may often be desirable to represent structured data using semi-structured models for the purpose of browsing [43]. Applications ranging from network analysis to transportation systems, from bioinformatics to XML repositories, from software engineering to computational linguistics, are now generating and processing a lot of semi-structured data (see Figure 1.1). For these applications to reach their full potential, there is a clear need to build an effective set of tools to index, process, manage, and analyze such data. There are two fundamental questions of interest:

1. **Data Management**: How can we efficiently manage and provide quick access mechanisms to these semi-structured data stores?

2. **Data Mining**: How can we discover hidden structural patterns from such data collections?

This dissertation focuses on data management and data mining issues in the context of a specific type of semi-structured data where the data is modeled using hierarchical tree-structured objects.
Figure 1.1: Examples of semi-structured data

(a) Internet topology (lumeta.com)
(b) Friendship network [133]
(c) Web site [139]
(d) Glycan structure [100]
(e) XML tree from DBLP
(f) Treebank [171]
There are a number of challenges in effectively dealing with semi-structured data. First and foremost is the algorithmic complexity involved in handling such data. The operations related to graph and tree structures can easily lead us to the space of hard problems. The key problem of containment or isomorphism is NP-hard for graphs. While it is tractable for trees, incorporating application-driven constraints like ancestor-descendant relationships is hard. Such containment checks are inherent in several data management and data mining tasks, including indexing, matching, and mining. Traditional strategies from the realm of relational databases are not readily applicable since the distinction between schema and data is fuzzy. Unlike in RDBMS, queries and updates to semi-structured data may refer to both the data and the schema, simultaneously. It is thus essential to rethink the design of data models, query languages, query processing techniques, and related optimizations.

The search space involved in processing semi-structured data sets when compared to transactional and sequential counterparts is typically very large. A rule of thumb is that the search space increases tremendously as the data becomes more complex. As an illustration, compare the number of distinct transactions, sequences, and rooted trees that can be constructed using a given set of labels/nodes. Appropriate strategies must be devised to efficiently traverse and effectively prune different portions of the search space. A query processor or a query optimizer should generate and evaluate a large number plausible query plans to select the best available plan for a given query. Similarly, a data mining engine must be able to sift through exponential number of subgraphs to discover interesting patterns in the data. The need for efficiency is thus immense while handling semi-structured data sets.

Traditional data-driven approaches towards efficient algorithms are however hard to employ in the context of semi-structured data. These approaches operate on selected portions of the database rather than processing all database records. Such selection is non-trivial in the context of data that is represented as trees and graphs. A sampling mechanism, for example, must not only sample from nodes or vertices in the graph but also from the linked structure associated with the selected nodes. Similarly, a compression scheme should be capable of summarizing the information related to nodes and their connections. The primary challenge is to design methods that respect both the content as well as the structure that is present in the data. Furthermore, incorporating domain-specific and user-defined constraints into the process is also a matter of concern while developing efficient techniques for semi-structured data.

Tapping the potential of modern computer systems for efficient algorithms is
challenging. Recent studies have shown that most data mining and data management algorithms grossly under-utilize the capabilities of modern computer systems \[84, 113, 128\]. In recent years, technology scaling has pushed the semi-conductor designs to reach new frontiers. Processor frequencies have reached their physical limits and led to the invention of multicore systems, in which multiple processing elements operating at low frequencies are packaged in a single dye to deliver better performance at smaller power envelopes. Multicore chips have altered the course of computing as we know it by bringing parallel processing paradigms to commodity desktop and high-performance server markets. Coping up with these technological advancements in the context of data analytic applications is a grand challenge. While multicore systems introduce new set of challenges, complex data types further add to the difficulty. For example, the inherent dependency structure present within the data coupled with parameter-driven workload skew make it hard to design good load balancing strategies for tree- and graph-structured data.

1.1 Thesis Statement

The complexity of data produced by applications is rapidly growing. Applications that produce and leverage semi-structured data are becoming ubiquitous. Simultaneously, modern computer architectures have ushered in new computing paradigms. Traditional algorithmic and data-driven optimizations, as well as architecture-conscious solutions are essential for realizing efficient algorithms for managing and analyzing semi-structured (complex) data sets.

1.2 Contributions

The contributions of this dissertation are highlighted in Figure 1.2. We investigate issues related to data analysis and data management of tree-structured data. Such data is commonly found in applications such as software engineering, bioinformatics, linguistics, etc. Towards efficient realizations, we follow a two-pronged approach that consists of two high-level strategies. The first strategy is to develop new algorithms and heuristics to deal with the data complexity, and the second strategy is to design architecture-conscious techniques that leverage the underlying computer systems. More specifically, we address the following problems: indexing and querying
tree databases; hashing tree-structured data; and mining frequent patterns from a collection of trees. These contributions are detailed below:

![Dissertation Overview](image)

**Figure 1.2: Dissertation Overview**

### 1.2.1 Managing Tree Databases

**Indexing:** Indexing is a fundamental technique for precise information organization and retrieval. It helps in quickly sifting through large data repositories to find the requested information (i.e., a query). In the context of semi-structured data sets, queries are also *structured* in nature and they are represented as small trees or graphs. Index structures thus have to deal with isomorphism and containment checks, which are known to be computationally expensive. Indexes must also be concise and should enable effective pruning of search space for efficient retrieval.

We propose a mechanism that extracts discriminant features from a given database and indexes them using a simple tunable inverted structure [177]. The resulting index is used to prune the search space while answering given structural queries. It is complemented with an efficient holistic query processing technique that retrieves matches by operating entirely on “succinct” sequence encodings of tree-structured data. Sequence-based representations as opposed to pointer-based data structures
enable the development of memory hierarchy conscious solutions. The proposed method retrieves the query matches orders of magnitude faster than state-of-the-art approaches. It also supports the retrieval of approximate matches and allows the user to specify constraints on the retrieved output.

**Hashing and Placement:** A hash table is a popular data structure that maps a large, possibly variable sized datum into a small fixed sized value via hash functions. Hash tables are ubiquitous and they are primarily used to improve the efficiency of search, for example in quickly finding items in a database. While there exists a number of methods for hashing numerical data, designing such functions for semi-structured data is an open problem. The inherent structure present in graphs and trees makes it difficult to pack them into simple hash values.

We develop a general framework that supports the development of hash functions for graph and tree structured data [176]. This framework consists of two operators. First, a transformation operator that converts a semi-structured data point into a multi-set of pivotal elements, where each pivot captures a finite set of structural relations present in the original data. The resulting multiset is fed into a sketching operator that maps it to a fixed size signature-sketch of the data point. Transformation functions are defined in such a way that they capture both structure as well as content that is present in the data. The key highlight here is that the framework allows the users to tune these two operators as per the demands of underlying application.

### 1.2.2 Mining Tree Databases

**Algorithms for Pattern Mining:** Frequent pattern mining is a fundamental task in the knowledge discovery process. It deals with mining useful and common patterns from massive data sets. In the context of semi-structured data sets, one is faced with two important challenges: first, the search space is exponentially large; second, it requires repeated executions of expensive isomorphism checks.

We propose new algorithms that address these challenges. First, a injective mapping between trees and concise sequences is established so that the search space is traversed via simple and fast operations on sequences [178]. Second, the subtree matching problem is recasted into a much simpler problem of subsequence matching, wherein appropriate structural constraints are embedded deep into the matching
process. Since the computation is performed on array based data structures, our algorithms exhibit very good data locality and provide high instruction-level parallelism.

1.2.3 Parallel Algorithms

Algorithms for Multicores: Multicore systems, primarily motivated by power and energy considerations, are becoming increasingly common-place. For these architectures, one must not only have to develop mechanisms to hide memory latency but also consider methods to reduce the memory bandwidth i.e., off-chip traffic. Equally important is the design of scalable parallel algorithms to deliver efficient performance on multicore chips. As most data mining workloads possess highly irregular and complex access patterns with many control and data dependencies, it is difficult to achieve good load balance among different processing elements.

We show that existing tree mining algorithms fail to deliver good performance on modern computer systems. They are memory intensive and exhibit poor data locality. We put forth a new paradigm of memory-conscious design in which algorithms trade memory for redundant computations. We demonstrate that such a design significantly reduces the memory footprint, which may often translate into improved execution times [174, 175]. It greatly reduces the off-chip traffic, alleviating the pressure on memory bandwidth. These memory optimizations are complemented with a multi-level adaptive parallelization framework that automatically and adaptively modulates the type and granularity of the work that is being shared among cores. Such a dynamic strategy is able to provide good parallel efficiency that is commensurate with number of cores on a chip.

Algorithms for Distributed Clusters: Mining truly large data sets is extremely challenging. The performance of most data mining algorithms does not scale linearly with the data set size. A cost-effective and viable solution to this problem is to use a tightly interconnected distributed cluster of processors.

We show that an effective data placement is very important while designing algorithms for distributed systems. We develop a distributed data placement framework that relies on our hashing algorithms to partition a given data set across multiple compute nodes in a cluster [41]. Such a service can be used to enhance the performance of data-intensive applications via localizing the computations, minimizing the communication costs, improving the reliability, and ensuring a balanced workload among cluster nodes. We demonstrate the usability of our framework by devising
distributed algorithms for pattern mining as well as database indexing. We also show some preliminary empirical results in this regard.

To reiterate, this dissertation presents new algorithms as well as system-specific optimizations for efficient semi-structured data mining and management. In terms of novel algorithms, we make the following contributions:

- We propose an injective sequential representation for trees that captures the complete information via two non-redundant and complementing sequences. Such a representation enables the design of cache-conscious algorithms.

- We present two algorithms that leverage our injective sequence mappings in realizing a pattern-growth mechanism for enumerating the set of all frequent subtrees from a forest rooted ordered trees. The candidate patterns are generated by traversing the search space via simple and fast operations on sequences.

- We design a sequence-based algorithm for the fundamental problem of embedded subtree isomorphism. The subtree matching problem is recasted into a much simpler problem of subsequence matching, wherein appropriate structural constraints are embedded deep into the matching process. This algorithm is complemented with a small and tunable inverted index to realize an efficient indexing mechanism to query XML databases.

- We present a generic framework for developing hash functions that map variable-sized semi-structured data objects into fixed size hash values. The hash functions respect both the content as well as the structure present in such data.

With respect to architecture-conscious techniques, we make the following contributions:

- We design several memory-conscious optimizations for realizing efficient semi-structured data mining algorithms on modern multicore systems. They are aimed at addressing the classical issue of latency to main memory and the problem of limited memory bandwidth available to all cores in the system.

- We develop scalable strategies for parallelizing pattern mining workloads on multicore systems. These strategies automatically detect the load imbalance in the system and adaptively partitions the work to guarantee good parallel performance. These strategies are embedded into a general-purpose task scheduling
service that supports the development of adaptive and moldable techniques for parallel database and mining tasks.

- We design a hash-based distributed framework for effective data placement on commodity cluster systems to enhance the performance of distributed data mining and management techniques.

1.3 Organization

In Chapter 2, we present our sequence-based approach for indexing tree databases. The hashing mechanism that converts a variable sized tree structure into a fixed-size signature is presented in Chapter 3. Our algorithms for frequent subtree mining are discussed in Chapter 4. We present several parallel algorithms for tree-structured data mining and data management that are targeted at shared-memory multicore and shared-nothing cluster systems in Chapter 5. Finally, Chapter 6 concludes the contributions of this dissertation and point out relevant future directions.
CHAPTER 2
INDEXING TREE-STRUCTURED DATA

The use of semi-structured data has been growing at a tremendous pace and spanning a wide range of application domains. Over the last few years, eXtensible Markup Language (XML) [34] has emerged as a de-facto standard for information exchange in many commercial and scientific applications. XML provides a flexible and open approach to model and share information among diverse organizations. The U.S. Federal Deposit Insurance Corporation (FDIC) has launched a very large and successful XBRL \(^1\) project in October 2005 that involves the collection of quarterly financial statements (call reports) from over 8300 banks. The system developed by them, called Central Data Repository (CDR), has revolutionized the way call reports are collected, validated, managed, and distributed [76]. Similarly, multimedia content description standards like MPEG-7 are now modeling a variety of multimedia features in a tree-structured form, allowing a uniform representation and content based image retrieval. Along with the use of semi-structured data, the need for high-performance query processing techniques to handle large scale data repositories is also growing.

Data in such XML documents is self-describing. The data is represented in the form of nested structure, similar to HTML documents. Users are free to create, use, and extend their own tags. While the tags in HTML documents typically stand for presentation-style, the tags in XML documents can be associated with semantic meanings. Furthermore, XML offers flexibility in organizing information. Unlike in flat relational database models, XML documents allow objects of the same type to have variable number and type of subobjects [89].

The basic data model of XML data is a rooted ordered labeled tree (see Figure 2.1). The nodes in the tree can be classified into three types – element nodes (e.g., ingredientlist); attribute nodes (e.g., type); and value nodes (e.g., sugar). The

---

\(^1\)XBRL (eXtensible Business Reporting Language) is a XML-based standard to define and exchange business and financial performance information.
edges denote the structural relationships among these nodes. There exists an extended data model that aims to reduce the redundancy via ID/IDREF features. An ID attribute uniquely identifies an element, and IDREF attributes refer to other elements that are explicitly identified by their ID attributes. For example, consider a university database with many teachers and students – all teaching and taking a number of different courses. One may choose to store all the information associated with teachers in a “teachers” element, and have all instances of teachers point to the main “teachers” element. Such data models are denoted using directed acyclic graphs. In this chapter, we primarily focus on a tree representation, as shown in Figure 2.1.

Since XML documents encapsulate both content and structure, the queries (also known as twigs) typically specify a subset of nodes as well as the structural relations present among them. XML queries can be categorized into two classes [130] – database style queries and information retrieval style queries. The latter type is mainly used
for databases which are text dense. Such queries return “fuzzy” results that are often ranked by their relevance (similar to Web search queries). In this chapter, we mainly consider database style queries that return all results that exactly match the queries (similar to SQL queries).

There exist several languages to specify database style queries. Examples include Lorel [5], XML-QL [69], XML-GL [78], XPath [57], XSLT [58], XQuery [44], and Quilt [45]. They have been developed with a view to specify complex structured queries on tree-structured data. The common feature of these languages is that the use of regular path expressions for querying [78]. For example, XPath uses path notation for navigating through hierarchical structure in the document. It is a syntax for defining different parts of an XML document. It is W3C recommended standard that is used in a number of other more expressive languages such as XQuery and XSLT.

XPath queries are represented as a sequence of alternating axes and tags. There are two important axes – parent-child axis that is denoted using a slash symbol (/); and ancestor-descendant axis that is represented as using double-slash (//). An XPath query “/recipe//@ingredient” returns all ingredient elements under all top-level recipe elements. XPath supports a number of other axes, such as parent, ancestor, attribute, following-sibling, etc. We mainly focus on the two axes / and // that are more general and well-known. Furthermore, a star symbol (*) used to match any node, just like in traditional regular expressions. XPath queries can also specify more complex structure by making use of predicates. For example, in a query “/recipe[type = cake]//@ingredient”, the text within square brackets specify the predicate (see Figure 2.1). It returns all ingredient elements from all cake recipes.

Given a database of XML documents, we focus on efficiently answering XPath twig queries with parent-child and ancestor-descendant axes. Since both documents and twigs are denoted as rooted ordered labeled trees, query processing involves performing embedded subtree isomorphism or minor containment checks. There are several efforts in leveraging well-established techniques from relational database systems (RDBMS) for querying semi-structured data repositories [78, 163, 204, 212]. However, they are inefficient and fail to deliver all the required functionalities [123, 163].

Much of recent research has focused on developing native query evaluation methods. Initial methods are directed towards path-based approaches, which break the query into simple paths on which the database is indexed. They consider a single path as the basic unit for querying. The result sets from each path expression are
then combined via an expensive join operation to obtain final set of database trees that match the given twig query. Researchers have proposed holistic query processing techniques to address the limitations of path-based approaches. Holistic methods do not break the query, and moreover, the matching is performed by considering the whole twig query as a single unit.

In this chapter, we explore the use of sequence-based holistic methods for the purpose tree indexing and matching. Such methods transform trees into sequences, and subsequently, operate only on sequences. Current techniques in this category \cite{187, 150} deliver poor performance due to space-inefficient sequence representations and expensive algorithms. We introduce a new Longest Common Subsequence based method for Tree Indexing and Matching (LCS-TRIM). We modify the classic dynamic programming based LCS method to find the location of all subsequence matches, and use it for our purpose. We design several optimizations which push the structural constraints deep in to the matching process. LCS-TRIM relies on a simple inverted index structures that can effectively prune the search space. Additionally, we present mechanisms that enable the user to specify constraints on the retrieved output and show how they can be pushed deep into the retrieval process. Our method can also support the retrieval of approximate matches. Through a detailed evaluation on real-world data sets, we demonstrate that LCS-TRIM exhibits up to three orders of magnitude speedup in query retrieval times over state-of-the-art XML indexing techniques.

**Chapter Organization:** We discuss the existing literature on the subject in Section 2.1. Our algorithms and optimizations for efficient embedded subtree matching are presented in Section 2.2. Section 2.3 evaluates the performance of proposed techniques. Finally, we conclude this chapter in Section 2.4.

### 2.1 Related Work

Most of the till date research on tree indexing can be broadly divided into two classes – path-based approaches and holistic approaches. Holistic approaches can further be divided in to two classes: twigstack-style and sequence-based approaches. We briefly review the literature in each of these categories.
2.1.1 Path-based Approaches

In these approaches, the given database is processed to build an index or summary for all individual paths. A given twig query is then broken down into paths and the index is probed for each resulting path to obtain a partial set of results. These partial answers are then joined together to form the answer for the given query. However, paths obtained from the query do not completely capture the structural information, and hence their pruning power is less when compared to that of the whole twig. Therefore, the paths retrieve a lot of false positive matches, which make the joining step very expensive, especially where the queries have more involved structure.

There are two important sub-classes here. The first type creates a structural summary of the given database, in the form of a labeled graph. The main goal, while constructing the summaries, is to preserve all the paths, and at the same time, to minimize the size of the resulting summary structure. These summaries are not static and are updated as and when the database is updated. Since they do not contain all data nodes, many paths need to still be examined.

Goldman et al proposed DataGuides which provide concise and accurate structural summaries of path structures for a semi-structured database. Milo et al pointed that DataGuides are restricted to a single regular expression, and are not useful for more complex queries with several regular expressions. To address this issue, they proposed template index that trades space for generality. The index is created using template paths, which contain one or more placeholders. Still, this index is not general enough to handle complex queries. Copper et al proposed IndexFabric that uses prefix-encoding where the paths are encoded as strings and are inserted to Patricia tries. This index (or structural summary) is specifically designed for path queries originating from the root node. Chung et al proposed an adaptive index called APEX that leverages data mining algorithms to summarize the frequent paths in the query workload. A drawback of this approach is that instead of keeping all root-to-leaf paths, it keeps all the paths of length two. Therefore, the results for queries with more than two nodes are obtained by performing joins. Since the features (paths of length two) are too small to capture the structural relationships, the performance can get significantly worse with the queries with complex branching structures.

The key limitations of methods relying on structural summaries are the following.
In practice, they can grow arbitrarily complex making it expensive to construct and also to access the summaries. Furthermore, when the given twig query is not in the indexed paths, then we need to rely again on expensive join operations.

The other class of path-based mechanisms are the ones which rely on node numbering schemes. In these algorithms, all the nodes are assigned a unique interval such that the evaluation of parent-child and ancestor-descendant relationships can be done in constant time by evaluating interval containment checks. Such checks are targeted to improve the performance of joining step. Li and Moon proposed an indexing system XISS [123] that numbers the tree nodes using a durable numbering system. In order to accommodate document updates, they have extended the widely used numbering scheme that depends on pre- and post-order traversals [13, 71]. Each node in their method is tagged with an interval \((\text{start}, \text{end})\), \(\text{end} > \text{start}\), such that: (i) the interval of any given node is contained in the interval of its parent node. (ii) for any two sibling nodes \(u\) and \(v\), if \(u\) appears before \(v\) in the pre-order traversal then \(\text{end}(u) < \text{start}(u)\). \(\text{start}\) numbers follow the pre-order traversal, but the ranges of unused numbers are left between subsequent nodes to make room for future insertions. We refer to this numbering scheme as range encoding, throughout this chapter.

Khalifa et al proposed a stack-tree-desc algorithm for structural joins, which are used to find all pairs of elements satisfying primitive structural relationships like parent-child and ancestor-descendant relationships [13]. Each element list in this approach is stored ordered on \(\text{start}\) and a stack is used to maintain elements that will be used later in the join. Such an approach makes a single sequential scan on each list while joining the intermediate results. Chien et al found that the performance can further be improved by by indexing the ordered element lists using traditional indices like \(B^+\) trees and \(R\)-trees [54]. Indices are leveraged to avoid the evaluation of elements which do not participate in joins. They concluded that the use of \(B^+\) trees is robust when compared to using \(R\)-trees to index the element lists.

In a completely orthogonal direction, some research has been done to optimize the query processing by eliminating redundant steps in XPath query processing [15, 77, 88, 155]. These approaches rely on the observation that the query evaluation time, in worst case, is exponential in the size of queries [88]. Strategies in this direction include changing the queries by identifying rewriting rules [27], eliminating reverse axes in queries to facilitate their evaluation on streaming data [144], minimizing the wildcard steps [46], and transforming the given queries in to algebraic form for efficient evaluation [95].
2.1.2 TwigStack-style Approaches

Bruno et al addressed the issue of expensive joining steps in path-based approaches by designing a holistic twig join algorithm called TwigStack [37]. In terms of ideas, this work is an extension of Khalifa et al work in [13]. This algorithm do not explicitly decomposes the query in to paths. It employs a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths. Although TwigStack helps to improve I/O and CPU performance for queries with only ancestor-descendant edges, it still can not control the size of intermediate results for queries with parent-child edges [129]. To address this limitation, Lu et al developed a look-ahead technique called TwigStackList. It associates each node in the twig query with two data structures, stack and list. The stack is used in a similar manner to TwigStack for compact representation of paths. List is used to cache a limited number of elements in the input data streams by looking-ahead. These cached elements are used to determine whether or not an element possibly contributes to final answer.

However, all these schemes deliver poor update performance. Chen et al proposed a dynamic labeling structure L-Tree that provides $O(\log n)$ amortized update cost [50]. Wu et al proposed various techniques to select the order in which structural joins should be performed to improve the query performance [194], similar to join order selection in relational database systems. Yu et al developed an algorithm TwigStackList$\rightarrow$ to incorporate NOT queries in to holistic twig joins [205].

By extending the idea of using indices for improving structural joins [54], researchers have improved the performance of holistic twig joins also by leveraging index structures [37, 103, 122]. A variety of indexes such as B$^+$Tree [54], XB-Tree [37], XR-Tree [103] are used to speedup the query processing. Typical probes to the index are in the form of findAncestors and findDescendants, which are answered efficiently by skipping some of the unnecessary data. A comparative study on the effectiveness of these indexes is done by Li et al [122]. They found that XB-TwigStack performs better than XR-TwigStack when skipping of ancestors is necessary. B$^+$-trees, on the other hand, performs as well as XB-TwigStack while skipping the descendants but are not efficient for skipping ancestor nodes.

2.1.3 Sequence-based Approaches

Another class of algorithms recast the tree matching problem in to subsequence matching problem by encoding the trees as sequences, and by operating on sequences
thereafter. The query processing starts by finding the subsequence matches for the twig’s sequence. A key challenge here is to discover the subsequence matches efficiently. As Moro et al pointed, this is an important step in this class of algorithms \[134\] One or more post-processing steps are then employed to filter the subsequence matches which do not correspond to actual twig matches. The challenge here is to detect and prune the false positives in as efficient a manner as possible.

ViST proposed by Wang et al transforms the trees into sequences using pre-order traversals \[187\]. PRIX \[150\] proposed by Rao et al uses Prüfer sequences which are constructed from post-order traversals. Prasad et al later modified the sequencing approach of PRIX for improved query performance \[98\]. Zezula et al used both pre- and post-order traversals simultaneously to represent the trees. All these sequencing methods typically result in long sequences which are space-inefficient (see Section 2.2.1). Wang and Meng proposed various performance-oriented principles to guide the tree sequencing process \[186\]. A limitation here is that for dynamic datasets the sequencing process cannot be determined apriori and for large datasets computing the optimal (constraint) sequencing may be expensive.

Most of these algorithms rely on index-based mechanisms to find the set of all subsequence matches. The tree nodes are indexed using positional representation such as range encoding \[150, 187\]. A series of range queries are then issued to the index to find the matches where each range query obtains the set of all descendants of a given node. For example, ViST stores the sequences in a disk-based virtual trie built using $B^+$-trees. More specifically, it uses two indices DocId $B^+$-tree and the combined $D$-Ancestor and $S$-Ancestor $B^+$-tree. The worst case storage requirement of one of the index ancestor index is linear in the total number of elements in the XML document. Indexing large sequence may result in underflows and it requires too many disk I/O’s to find matches due to top down (i.e., pre-order) transformation of trees. Furthermore, in cases where the database trees have identical sibling nodes, ViST produce a lot of false positive subsequences \[98\]. PRIX also finds the subsequence matches in a similar manner to ViST except that the Prüfer sequences, instead of pre-order sequences, are indexed in to disk-based indices by following range encoding method. Note that, each matching step (for a child query node) in these algorithms involves an index probe and a range search. Moro et al showed that the way subsequence matches are found in ViST and PRIX correspond to static plans because the sequence of the joins is statically defined by the query sequence \[134\]. They proposed indexing methods called Indexed Nested Loop Join (INLJ) for lightweight XML processors.
For example, they presented efficient algorithms for finding ancestor nodes using $B^+$-trees, which are lightweight when compared to other robust indices, and backward range searches. They compared all plans of INLJ with TwigStack and showed that TwigStack algorithms are robust and deliver better performance.

Not all the subsequence matches found correspond to structure matches. As pointed out in [89], the number false positive subsequence matches, in a worst case, exceed the size of intermediate results produced in TwigStack-style algorithms. Therefore, efficient pruning of false positives is very important in achieving good performance. Current sequence-based approaches filter out the false positive subsequences by employing expensive join operations [187] or by employing post-processing steps [150]. For example, PRIX employs up to four phases of refinements where each phase makes a complete pass over the given twig query. There are two main disadvantages of such approaches: first, the filtering process starts only after generating all the subsequence matches. It would be ideal to have an approach that integrates the filtering process and the subsequence matching process so that the false positives are avoided in early stages. Second, the amount of computation performed while filtering is very high (e.g., four different stages in PRIX). Therefore, the number of false positive matches significantly affects the overall performance of the algorithm. The techniques proposed by Wang and Meng [186] address the first problem but it again relies on large indices to enumerate the subsequence matches.

Our algorithm LCS-TRIM falls under the category of sequence-based algorithm since we also determine the tree matching by finding the subsequence matches. Our approach significantly differs from state-of-the-art approaches in all phases of matching i.e., in tree encoding, in subsequence matching, and in filtering. First, our tree encoding methods result in very efficient and concise representations, which help not only in reducing the memory requirements but also in facilitating the matching process. Unlike current algorithms, we leverage a simple dynamic programming based approach for enumerating the subsequence matches. This is significant because it completely avoids the probes in to large disk-based indices in every step of the matching process. Furthermore, our filtering mechanisms efficiently address the limitations of current approaches. First, they are embedded in to the subsequence matching process so that the false positives are detected and avoided in early stages – the number of false positives is no longer an issue. Second, our algorithm makes a single pass over the twig in determining whether or not a subsequence is false positive.
the filtering phase is no longer expensive. To further improve the performance, we leverage a simple inverted index structure to prune the search space efficiently.

Zezula et al proposed methods which are based on tree signatures constructed from both pre- and post-order traversals [211]. While they also determine the twig matches by employing a dynamic programming based approach, our algorithms differ from their methods in many different ways. First, our sequences are much more compact than their extended signatures because of firstFollowing and firstAncestor nodes. Second, the way LCS-TRIM identifies the false positive subsequences is a major difference from their method. Third, our simple inverted index complements our tree matching algorithm in early-pruning of the search space. Fourth, our novel optimizations in LCS-TRIM greatly reduce the computation overhead. Finally, embedding of pruning steps into the matching process is a key innovation we have that is not in the methods proposed by Zezula et al. These differences directly translate into tremendous performance improvement in query processing (see Section 2.3).

2.2 LCS-TRIM: LCS based TRee Indexing and Matching

In this section, we describe our approach LCS-TRIM for indexing and querying tree databases. The framework is shown in Figure 2.2. There are three important steps: data representation; tree matching; and tree indexing. We are given a database $D$ of trees (e.g., XML documents). All trees are first transformed into sequences in the data transformation phase. We propose two different methods for sequencing trees. Along with the database, twigs from the query workload $Q$ are also mapped to respective sequences. The transformed database trees are then used to construct the index structure. Our approach relies on a simple inverted index structure to prune the search space. Both the transformed database $T(D)$ and the transformed workload $T(Q)$ are then fed to the filtering phase. This phase probes our index structure to select a subset of trees $T'(D) \subseteq T(D)$. For each tree in the filtered database, our tree matching algorithm is run to obtain the final answer that consists of all matching documents. The tree matching algorithm is divided into three main parts: subsequence checking, subsequence matching, and structure matching.

We first describe our data transformation methods and compare them with other competing methods in Section 2.2.1. We then present our tree matching algorithm in Section 2.2.2 and our indexing method in Section 2.2.3. We finally show several optimizations that improve the performance of tree matching algorithm in Section 2.2.4.
Figure 2.2: Framework of *LCS-TRIM*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>A database tree with $n$ nodes</td>
</tr>
<tr>
<td>$Q$</td>
<td>A twig query with $m$ nodes</td>
</tr>
<tr>
<td>$PON$</td>
<td>Post-Order traversal Number of a tree node</td>
</tr>
<tr>
<td>$NPS_T$</td>
<td>Numbered Prüfer sequence of $T$</td>
</tr>
<tr>
<td>$LS_T$</td>
<td>Label sequence of $T$</td>
</tr>
<tr>
<td>$NPS_T[k]$</td>
<td>$k^{th}$ entry in $NPS_T$ (similarly $LS_T[k]$)</td>
</tr>
<tr>
<td>$CPS(T)$</td>
<td>$(NPS_T, LS_T)$ – (consolidated) Prüfer sequence of $T$</td>
</tr>
<tr>
<td>$DFS(T)$</td>
<td>$(Parent_T, LS_T)$ – depth first sequence of $T$</td>
</tr>
<tr>
<td>$R[i,j]$</td>
<td>An entry in the LCS matrix $R$</td>
</tr>
<tr>
<td>$SM$</td>
<td>A subsequence match $(i_1,...,i_m)$</td>
</tr>
</tbody>
</table>

Table 2.1: Notation
Algorithm 1 Data transformation – Prüfer sequences

Input: Tree $T$ of size $n$
Output: Sequence representation of $T$ – $CPS(T) = (LS_T, NPS_T)$

1: $LS_T \leftarrow \phi$
2: $NPS_T \leftarrow \phi$
3: Perform post-order traversal on $T$ {each node is tagged with its Post-Order traversal Number (PON)}
4: for $i = 1$ to $n$ do
5: $v \leftarrow$ tree node whose PON = $i$
6: $LS_T[i] = v.label$
7: if $i! = n$ then
8: $NPS_T[i] = v.parent.nps$
9: else
10: $NPS_T[i] = 0$
11: end if
12: end for

2.2.1 Data Transformation

In this phase, all database trees are transformed into sequences using one of our two tree sequencing methods. The first method is based on post-order traversal, whereas the second one relies on a pre-order traversal of tree nodes. Sequences constructed from post-order traversal are referred to Prüfer sequences and the ones built from pre-order traversal are called as Depth First Sequences. The method in which these two types of sequences are constructed is very similar. In the following discussion, we describe the construction of Prüfer sequences.

Prüfer sequences were first used by Heinz Prüfer in the context of proving Cayley’s formula in 1918 [152]. They provide a bijection between the set of trees with distinctly labeled nodes (n of them) and the set of sequences of length $n-2$ over the same label space. Trees in most real-world data sets rarely contain uniquely labeled nodes. Therefore, we attach more information to classical Prüfer sequences so that nodes with same label can be represented uniquely. In our representation, each tree is associated with two sequences, viz. Numbered Prüfer Sequence $NPS_T$ and Label Sequence $LS_T$ – see Table 2.1.

We first perform a post-order traversal on a given tree $T$, and annotate each node with its post-order traversal number (PON). Starting with two empty sequences, we iteratively construct NPS and LS for $T$. In each iteration, we remove the node with
smallest PON, add its label to LS and its parent’s post-order number to NPS. We continue this process until all the nodes are removed and the sequences are appended with corresponding labels and PONs. Since the root node node does not have any parent, when it is removed, its NPS entry is set to 0. The entire procedure is summarized as Algorithm 1.

The NPS resulting from this algorithm would be as follows: \((p_1, p_2, ..., p_{n-1}, 0)\), where \(p_i\) is the parent of the node whose PON = \(i\) i.e., the one that is deleted in \(i^{th}\) iteration. The label sequence \(LST\) is the sequence of node labels that are deleted in each step. We refer to the pair \((NPS_T, LST)\) as the Consolidated Prüfer Sequence (CPS) of \(T\). In summary, the CPS of a tree \(T\) with \(n\) nodes is an ordered sequence of pairs \(<(p_i, l_i)>\), where \(l_i\) is the label of a node whose PON is equal to \(i\), and \(p_i\) is the PON of its parent \((p_n=0)\). Note that each entry \((p_i, l_i)\) in CPS corresponds to an unique edge in the tree. Hereafter, we refer to a CPS as just a Prüfer sequence, for simplicity. Example trees \((T_1 \text{ and } T_2)\) and corresponding Prüfer sequences are shown in Figure 2.3.

\[\text{Figure 2.3: Comparison of different sequencing methods}\]

**Lemma 2.2.1.** Let \(\mathbb{T}\) be a set of all rooted ordered labeled trees, and \(\mathbb{S}\) be the set of all valid Prüfer sequences constructed from Algorithm 1. The mapping \(f : \mathbb{T} \rightarrow \mathbb{S}\) is an injective function.

Lemma 2.2.1 states that each tree \(T \in \mathbb{T}\) is uniquely represented by its corresponding \(CPS(T) \in \mathbb{S}\). The numbered sequence \(NPS\) is constructed from post-order
traversal numbers, and it uniquely captures the entire structure of the tree. Tree with 
the same size but different structure will map to different numbered sequences. Given 
a NPS, one can easily construct the topology of the corresponding tree by following 
Algorithm 1 in reverse order. The label sequence LS, on the other hand, provides 
ode labels for the topology given by NPS. In other words, NPS and LS capture 
mutually exclusive but complementing information. Therefore, NPS and LS uniquely 
represents a rooted labeled ordered tree, in which multiple nodes can share the same 
label.

The depth first sequences (DFS) are also constructed in a similar manner except 
that they are based on pre-order or depth-first order traversal instead of post-
order traversal. In the case of DF sequences, we refer to the NPS as the parent se-
dequence Parent – see Table 2.3. Algorithm 1 can easily be modified 
to construct depth first sequences. All properties that are applicable to prüfer se-
quences are also applicable to depth first sequences. For example, depth first sequence 
DFS(T) = (Parent, LS) denotes a unique rooted labeled ordered tree T (similar to 
Lemma 2.2.1). The algorithms developed for prüfer sequences are also applicable 
to DFS. For the sake of explanation, we present all our algorithms in the context of 
Prüfer sequences. Similar techniques are also applicable for trees that are represented 
using depth first sequences.

**Comparison with Other Tree Sequencing Methods**

There are several advantages of our sequencing methods when compared to the meth-
ods used by other existing methods. Both NPS\(T\) and LS\(T\) store absolutely no redundant 
information, and moreover, they complement each other. While NPS\(T\) provides 
the structure of the tree, LS\(T\) gives the labels for each node in the tree. Since 
they uniquely represent the given tree, they provide an injective transformation (see 
Lemma 2.2.1). Furthermore, since the sequences do not store any redundant infor-
mation, the transformation is concise. We will now compare our sequencing method 
with existing techniques.

Prüfer sequences are also used in Prix algorithm [150] – they are Regular Prüfer 
Sequence (RPS) and Extended Prüfer Sequence (EPS) [150] (see Figure 2.3). RPS(T) 
contains two sequences NPS and LPS. Our method to construct NPS from Algo-

rithm 1 is same as the one in which NPS sequences in Prix are constructed. Unlike 
our label sequences, RPS stores a Labeled Prüfer Sequence (LPS) that is constructed 
by taking the parent node labels of the deleted nodes. Note that both sequences in
RPS (i.e., NPS and LPS) encapsulate information about only the parent nodes in the tree. Therefore, RPS can not represent the leaf nodes. Also, note that the label of each tree node is repeated in LPS as many times as the number of its children. If a node has \( k \) number of children then its label is repeated \( k \) times in LPS. RPS thus not only stores incomplete information but also host a lot of redundant information.

To address the limitations of RPS, Prix extends each leaf with a dummy node and then constructs NPS and LPS on top of the extended tree. The resulting sequence ignores all dummy nodes and captures both internal and leaf nodes. This representation is called as an Extended Prüfer Sequence (EPS). Even though EPS contain the leaf information, they can potentially be longer in size – consider the example tree \( T_2 \) in Figure 2.3. The space complexity of RPS is same as our CPS but it is incomplete. On the other hand, EPS is complete but its space complexity is high – in the worst case, it is twice that of CPS. Therefore, our CPS representation is space efficient when compared to sequences used in Prix.

ViST represents trees using structure encoded sequences (SES) \([187]\). SES of a tree is derived from its pre-order traversal, and is a sequence of \((\text{symbol}, \text{prefix})\) pairs, where \text{symbol} represents a node in the given tree and \text{path} denotes the path from root node to the current node. For example, SES of \( T_1 \) in our example database in Figure 2.3 would be the following:

\[
\begin{align*}
\end{align*}
\]

Prasad et al modified the Prüfer based encoding of Prix and proposed new sequencing method, which we refer to as MPS \([98]\). Similar to EPS of Prix, MPS also relies on extended trees where each leaf node is extended with a dummy node. Let \( u \) be a node in the tree whose label is \( l \) and parent is \( v \). Each node \( u \) in the tree is represented with the tuple \((\text{label}, \text{elementNum}, \text{level}, \text{count})\), where \text{label} is the label of the parent node \( v \), elementNum is \( 1 + 1 + \text{(number of nodes with same label as } l\text{ that appear before this node in the depth first traversal of the tree)} \), level is the depth of \( v \), and count is the number of nodes (including dummy nodes) in the subtree rooted at the node \( u \) in the original tree. For example, modified Prüfer sequence for \( T_1 \) from Figure 2.3 is the following:

\[
\begin{align*}
\text{MPS}(T_1) &= (F, 1, 3, 1)(B, 1, 2, 2)(A, 1, 1, 3)(D, 1, 5, 1)(B, 2, 4, 2) \\
&\quad (D, 2, 5, 1)(C, 1, 4, 2)(A, 2, 3, 6)(E, 1, 2, 7)(A, 1, 1, 8) \\
\end{align*}
\]
Similarly, in the approach proposed by Zezula et al, each tree is represented by its signature (ZS). The tree signature of a tree makes use of both pre- and post-order traversals of the tree. Let \( u \) be a node in the given tree with label \( l \) and its parent is \( v \). The signature is a sequence of tuples of the form \((lab, post, ff, fa)\) ordered by their pre-order number, where \( lab \) is the label of \( u \), \( post \) is the post order number of \( u \), \( ff \) and \( fa \) are the pre-order numbers of first following and first ancestor node of \( u \). For example, the signature of \( T_1 \) in our example database is the following:

\[
ZS(T_1) = (A, 9, 10, 0)(B, 2, 4, 1)(F, 1, 4, 2)(E, 8, 10, 1) \\
(A, 7, 10, 4)(B, 4, 8, 5)(D, 3, 8, 6)(C, 6, 10, 5)(D, 5, 10, 8)
\]

To summarize, the sequences produced by our transformation method shown in Algorithm 1 are much smaller than those generated by existing methods. Our approach represents the trees in a concise and space-efficient manner, when compared to competing methods. This is a very important property because longer sequences tend to result in poor cache performance as they may not fit in few cache lines. Once the database trees and queries in the workload are transformed into sequences, our tree matching algorithm and subsequent optimizations operate entirely on sequences.

### 2.2.2 Tree Matching Algorithm

Our tree matching algorithm operates on two trees, \( T \) and \( Q \), where \( T \) is a database tree and \( Q \) is a given twig query. Without loss of generality, let the number of nodes in \( Q \) and \( T \) is \( m \) and \( n \), respectively. The algorithm enumerates all the embeddings of \( Q \) in \( T \). Note that the embeddings are embedded subtree isomorphisms.

Since the trees in both the database and workload are converted into sequences, we recast the tree matching process into the process of subsequence matching. A subsequence of some sequence is a new sequence that is formed from the original sequence by deleting some of the elements without disturbing the relative positions of the remaining elements.

**Definition: 2.2.1.** A sequence \( X = \{x_1, \ldots, x_m\} \) is said to be a subsequence of another sequence \( Y = \{y_1, \ldots, y_n\} \) if \( x_j = y_{i_j}, 1 \leq j \leq m, 1 \leq i_j \leq n, \) and \( i_1 < i_2 < \ldots < i_m \).

**Theorem 2.2.1.** Consider a tree \( T \) and a twig query \( Q \) with their label sequences \( LS_T \) and \( LS_Q \), respectively. If \( Q \) is a subtree of \( T \) then \( LS_Q \) is a subsequence of \( LS_T \).
Proof. If $Q$ is a subtree of $T$ then for each node $v_i \in Q$ there exists a node $u_i \in T$ such that each edge $(v_i, v_j)$ in $Q$ corresponds to an ancestor-descendant or a parent-child relationship between $u_i$ and $u_j$ in $T$. Moreover, the order among the child nodes of $v_i$ in $Q$ is preserved from $T$. Therefore, the order in which $v_i$’s are deleted while constructing $LS_Q$ is same as the order in which $u_i$’s are deleted when constructing $LS_T$. The label sequence of twig query $Q$ is thus a subsequence of the label sequence of tree $T$. □

Theorem 2.2.1 provides a necessary but not sufficient condition for twig matching. Since it is only a necessary condition, there can exist some subsequences which do not correspond to subtree matches. Therefore, our tree matching algorithm contains three steps – checking for subsequence, subsequence matching, and structure matching. In the first step, we check if the label sequence of the query tree $LS_Q$ is a subsequence of $LS_T$. If yes, we move on to the next step that enumerates the set of all subsequence matches of $LS_Q$ in $LS_T$. Since Theorem 2.2.1 is only a necessary condition, we need to have a post-processing step that filters the false positive subsequences. This step is called as structure matching. We now describe each step, in detail.

Step 1: Checking for Subsequence
In this step, we evaluate the necessary condition given by Theorem 2.2.1. Here, we make the following simple observation that links the subtrees to longest common subsequences.

**Property 2.2.1.** If a label sequence $LS_Q$ is a subsequence of another label sequence $LS_T$ then $LS_Q$ is the longest common subsequence of $LS_Q$ and $LS_T$.

From the above property, in order to check if $LS_Q$ is a subsequence of $LS_T$, it is sufficient to check their longest common subsequence (LCS). Finding a longest common subsequence is a well-addressed problem in the literature. A large number of algorithms have been proposed for this purpose [28, 99, 184]. We employ a traditional dynamic programming based approach where the LCS length between two input strings $LS_Q[1..m]$ and $LS_T[1..n]$ is computed by finding the LCS lengths for all possible prefix combinations of $LS_Q$ and $LS_T$. Computed LCS lengths are stored in a matrix and are used later in finding the LCS length for longer prefixes – dynamic

27
programming. Equation 2.2.1 gives the recurrence relation for extending the LCS length for each prefix pair \((LS_Q[1..i], LS_T[1..j])\) \[^{184}\].

\[
R[i, j] = \begin{cases} 
0, & \text{if } i = 0, j = 0 \\
R[i - 1, j - 1] + 1, & \text{if } LS_Q[i] = LS_T[j] \\
\max(R[i - 1, j], R[i, j - 1]), & \text{if } LS_Q[i] \neq LS_T[j] 
\end{cases} \tag{2.2.1}
\]

An entry \(R[i, j]\) gives the LCS length for a prefix pair \((LS_Q[1..i], LS_T[1..j])\). The bottom-right corner entry \(R[m, n]\) gives the overall LCS length. If that is different from \(m\) then we can derive, from Property 2.2.1 and Theorem 2.2.1, that the twig \(Q\) is not a subtree of \(T\). Note that in case of DFS, either the backtracking has to start from the top left corner of \(R\) or the matrix has to be constructed on the reversed label sequences. **Complexity Analysis:** The recurrence in Equation 2.2.1 scans every entry in \(R\) exactly once. Therefore the run time complexity of this algorithm is \(\Theta(mn)\).

**Step 2: Subsequence Matching**

Given that \(LS_Q\) is a subsequence of \(LS_T\), we now enumerate all the subsequence matches of \(LS_Q\) (see Algorithm 2). In order to do that, we employ a backtracking approach that recurses on the \(R\)-matrix that is constructed in the previous step. The backtracking starts from the bottom-right corner \(R[m, n]\) and proceeds to the top-left corner \(R[1, 1]\) (lines 5 – 6, 9, 11). At every step in backtracking, we evaluate whether both label sequences agree on labels. At any step, if both the label sequences match at positions given by \(Q_{ind}\) and \(T_{ind}\), we record the matching location \(T_{ind}\) (line 1 – 2). If the labels do not match, then the backtracking is continued in the direction that has maximum LCS length (line 8). Whenever the match for the node at position \(Q_{ind}\) is recorded, we have the complete match and it is reported (line 5). Since \(R\) is processed from the bottom-right corner to the top-left corner, the subsequence matches are established from right to left. Note that, Algorithm 2 generates all subsequence matches and each match is generated exactly once.

Each resulting subsequence match \((SM)\) is denoted by the list of positions at which the match is found – \((i_1, ..., i_m)\) i.e., \(LS_Q[k] = LS_T[i_k], 1 \leq k \leq m\). Alternatively, \(SM\) is represented as a sequence that is formed by the entries of CPS(\(T\)) taken from the matching positions. More formally,

\[
SM = ((NPS_T[i_1], LS_T[i_1], i_1) \ldots (NPS_T[i_m], LS_T[i_m], i_m)) \tag{2.2.2}
\]
Algorithm 2 Enumerating the set of all subsequences

naiveBacktrack \((Q_{ind}, T_{ind})\)

**Input:** \(L_{ST}, L_{SQ}, R\)-matrix

1. if \(L_{SQ}[Q_{ind}] = L_{ST}[T_{ind}]\) then
2. record the match for \(L_{SQ}[Q_{ind}]\)
3. if \(Q_{ind} == 1\) then
4. output the match
5. return
6. naiveBacktrack \((Q_{ind}-1, T_{ind}-1)\)
7. naiveBacktrack \((Q_{ind}, T_{ind}-1)\)
8. else
9. if \(R[Q_{ind}-1, T_{ind}] > R[Q_{ind}, T_{ind}-1]\) then
10. naiveBacktrack \((Q_{ind}-1, T_{ind})\)
11. else
12. naiveBacktrack \((Q_{ind}, T_{ind}-1)\)

Note that, the sequence representation in Equation 2.2.2 is an alternative representation to the sequence of matching positions \((i_1, \ldots, i_m)\). All the generated subsequence matches are fed into the structure matching step that filters out the false positive matches.

**Complexity Analysis:** Algorithm 2 is clearly exponential because the potential number of subsequence matches enumerated by it is exponential. We therefore analyze its worst case complexity in terms of the total number of recursive calls made. Consider the worst case scenario where each node in the tree and the twig has the same label. In this case, only the recursions in lines 3 and 4 are executed because the labels of both sequences match at every position. Say that \(a_{m,n}\) is the number of recursions made by Algorithm 2 in the worst case and \(b_{m,n}\) is the number of matches in the worst case, where \(m\) and \(n\) are the input parameters \(Q_{ind}\) and \(T_{ind}\), respectively.

The recurrences to derive the values of \(a_{m,n}\) and \(b_{m,n}\) are shown in Equation 2.2.3 and Equation 2.2.4. The exponential nature of \(a_{m,n}\) is evident from the second condition in Equation 2.2.3. The recurrence of \(b_{m,n}\) has a nice closed form \(\binom{n}{n-m}\). As an example, for a twig of size 5 and a tree of size 10, the algorithm makes \(a_{5,10}=1,275\) number of recursions to find \(b_{5,10}=252\) subsequence matches.

\[
a_{m,n} = \begin{cases} 
1 + a_{m-1,n-1} + a_{m,n-1}, & \text{if } n > m \\
1 + 2 \times a_{n-1,n-1}, & \text{if } n = m \\
1 + 2 \times n, & \text{if } m = 1
\end{cases} \tag{2.2.3}
\]
Step 3: Structure Matching

The false positives are generated in the previous step because the subsequence matching algorithm considers only the label information and completely ignores the structural information. There can exist multiple twig patterns which differ in structure but with the same label sequence. Our structure matching algorithm prunes these false positive subsequences by considering the structure given by $NPS$.

**Definition: 2.2.2. Structure Agreement:** Consider two sequences, derived from two trees $T_1$ and $T_2$, $S_1 = ((A_1, B_1) \ldots (A_m, B_m))$ and $S_2 = ((C_1, D_1) \ldots (C_m, D_m))$, where $A_i$'s and $C_i$'s define the structure; $B_i$'s and $D_i$'s provide the labels. Both $S_1$ and $S_2$ are said to agree on structure at position $i$ if and only if the following three conditions hold:

i) $1 \leq i \leq m$, ii) $B_i$ is equal to $D_i$, iii) If $A_i$ is the parent of $B_i$ in $T_1$ then $C_i$ is the parent of $D_i$ or the nearest ancestor of $C_i$ that is in $S_2$ must agree on structure with $S_1$ at position $A_i$.

**Theorem 2.2.2.** A twig query $Q$ is a subtree of $T$ if and only if its Prüfer sequence $CPS(Q)$ and its subsequence match $SM$ at locations $(i_1, \ldots, i_m)$ in $T$ agree on the structure at all positions $k, \forall k, 1 \leq k \leq m$.

**Proof.** **IF:** From the definition of a subsequence, the first two conditions of Definition 2.2.2 trivially hold true for $CPS(Q)$ and $SM$. Consider an entry in the twig's prüfer sequence $(NPS_Q[k], LS_Q[k])$ and its corresponding entry $(NPS_T[i_k], LS_T[i_k])$ in its $SM$. Recall that each entry in a Prüfer sequence is an edge. $NPS_Q[k]$ is thus the parent of the query node at position $k$. Since $Q$ is an embedded subtree of $T$, this edge corresponds to a parent-child relation in $SM$ (i.e., $NPS_T[i_k] = NPS_Q[k]$ – first part of condition iii) or an ancestor-descendant relation $SM$ (second part of condition iii). **ONLY IF:** An isomorphism between $Q$ and its subsequence match is a bijective label-preserving function $f$ that maps every vertex $v_k$ in $Q$ to a vertex

\[
 b_{m,n} = \begin{cases} 
 b_{n-m+1} \sum_{i=1}^{n-m+1} b_{n-1,n-i}, & \text{if } n > m \\
 1, & \text{if } n = m \\
 n, & \text{if } m = 1 
\end{cases}
\]
\( f(v_k) \) in \( T \) in such a way that the adjacencies are preserved. In other words, for every node \( v_j \) that is adjacent to \( v_k \) there is a node \( f(v_j) \) that is adjacent to \( f(v_k) \).

Now consider a function \( g : Q \rightarrow SM \) such that \( g(v_k) = u_{i_k} \), \( v_k \)'s and \( u_k \)'s are vertices in \( Q \) and \( T \), respectively. Furthermore, assume that \( g \) satisfies all the conditions in Definition 2.2.2 at each \( k \). First two conditions infer that \( g \) is a label-preserving function. The third condition infers that \( g \) maps every edge in \( Q \) to a parent-child or ancestor-descendant relation in \( T \). Since the node relationships are determined by edges, we can deduce that \( g \) preserves the adjacencies among vertices in \( Q \). Therefore, \( g \) defines an isomorphism between \( Q \) and the subtree formed by its subsequence match in \( T \).

From the above theorem, it is sufficient to check for the structure agreement between \( Q \) and its subsequence match \( SM \) at all positions in order to find whether \( Q \) is a subtree of \( T \) or not. The structure matching algorithm based on this theorem is shown in Algorithm 3.

Algorithm 3 Algorithm for matching the structure of two sequences

| Input: CPS(Q), CPS(T), SM=(i_1, ..., i_m) |
| Output: mapping: positions at which Q matches to a subtree in T |
1: mapping[m] ← i_m
2: for \( k = m - 1 \) to 1 do
3: \( p_q \leftarrow NPS_Q[k] \)
4: \( p_t \leftarrow NPS_T[i_k] \)
5: if mapping[\( p_q \)] is equal to \( p_t \) or is an ancestor of \( p_t \) in \( T \) then
6: mapping[\( k \)] ← \( i_k \)
7: else
8: Report that \( Q \) is not an embedded subtree of \( T \)
9: Report that \( Q \) is an embedded subtree of \( T \)

Algorithm 3 iteratively processes the nodes in \( Q \) and stores the structure match found so far in \( mapping \) array. Let \( u \) be a node in \( Q \) and \( v \) be its corresponding node in a particular subsequence match. When processing the node \( u \), we evaluate if \( u \)'s parent in \( Q \) is mapped to an ancestor of the corresponding node \( v \) in \( T \) or not. Such a structure agreement check basically transforms the parent-child relation between \( u \) and its parent in \( Q \) to an ancestor-descendant relation between \( v \) and \( mapping[v] \).
The structure agreement check at a node is performed *only after* processing all of its ancestors because the check involves evaluating the mapping of parent nodes. The nodes in the twig are thus processed from right-to-left i.e., from root to leaves (line 2) (similar to subsequence matching). Since the root node does not have any ancestors, it is mapped without performing any checks (line 1). As soon as the check at position $k$ is successful, the matching node in $T$ at position $i_k$ is recorded in mapping array (line 6). To perform such a check, the algorithm considers the parent node numbers from $k^{th}$ entry in both the prufer sequence of $Q$ and its subsequence match $SM$ (lines 3-4). The algorithm checks if the node mapped to $p_q$ is same as $p_t$ or is an ancestor of $p_t$ (line 5). By doing so, the algorithm makes sure that a parent-child relation in $Q$ is translated into an ancestor-descendant relation in $T$. If the check fails at any position, it can be concluded that $Q$ is not a subtree of $T$. Since $SM$ is a label match to $Q$, Algorithm 3 does not check for labels. Note that, mapping[$p_q$] in line 5 will always have a value as the structure match is established from right to left.

**Complexity Analysis:** Algorithm 3 accesses each node in the twig exactly once (line 2). This scan takes $\Theta(m)$ time. The structure agreement check at each position traverses the path from $i_k$ to mapping[$p_q$] (line 5). In the best case where $p_t = mapping[p_q]$, the traversal takes $\Theta(1)$ time. In the worst case, $p_q$ is mapped to the root of $T$ in which case the traversal time is in the order of the tree’s depth. The complexity of line 5 in such cases is $O(n)$ because the maximum depth of a tree is equal to the number of nodes. Therefore, Algorithm 3 has a *best case run time complexity* of $\Omega(m)$ (for the case of induced subtree matching where there are no wildcards) and a *worst case complexity* of $O(mn)$. The constant factor involved is usually small because in the worst case scenario (a single long path), if the check at a node $A$ traverses till the root then the checks for all $A$’s descendants will at most traverse up to $A$ and never hence reaches the root node.

**Example:** Figure 2.4 shows the subsequence and structure matches of the twig query $Q_1$ in the example database trees $T_1$ and $T_2$. Each subsequence match can also be represented in the sequence format. For example, the sequence for $M_1$ is $((B, 9, 2) (D, 4, 3) (A, 8, 7))$ and for $M_3$ it is $((B, 7, 4) (D, 5, 6) (A, 8, 7))$. Algorithm 3 detects both $M_1$ and $M_2$ as false positives because they fail the structure agreement check at position 1. For example, for $M_1$, $(i_1,i_2,i_3)=(2,3,7)$. The structure matching algorithm maps the root node $A$ (node# 3) in $Q$ to node 7 in $T_1$ and maps $D$ (node# 2) in $Q_1$ to node 3 in $T_1$. Now consider the check at position 1. The parent of node# 1 $(B)$ in $Q_1$ is 3 i.e., $p_q=3$ and $p_t=NPST_{T_1}[i_1]=9$. The mapped node for $p_q$, mapping[$p_q$]=7,
is not same as $p_t$ and it is not an ancestor of $p_t$. Therefore, the check at position 1 fails for $M_1$ and hence it is not a structure match. Similarly, $M_2$ does not agree on structure at position 1. In $T_2$, Both the subsequence matches, $M_7$ and $M_8$, can be shown to be actual twig matches for $Q_1$.

As noted earlier, the ancestor check in line 5 of Algorithm 3 can traverse till the root node, in worst case. Approaches such as scoping [208] and range encoding [134, 187] are known for making fast ancestor checks. They take $O(1)$ time as we just need to check if $p_t \in \text{Scope}(\text{mapping}[p_q])$ or not. However, this seemingly accurate approach cannot find all the embeddings of $Q$, and it can potentially generate false positive structure matches. For example, consider the twig query that is matched with the nodes 2, 6, 8, and 9 in $T_1$. The scope-based approach incorrectly flags the subsequence match $(4, 6, 8, 9)$ as a valid structure match. Therefore the traversal in line 5 is mandatory for the correctness of Algorithm 3.

### 2.2.3 Indexing Method

Our tree matching algorithm presented in previous sections can find all the matches for the twig in a given database tree. A naive method to determine all the matches in the database is to run the tree matching algorithm on every tree in the database. This is definitely an inefficient approach, especially when the database contains a
large number of trees. Instead, we leverage our index structure to select a subset of trees from the database on which we run the tree matching algorithm.

Our index is a simple inverted index with as many entries as the number of distinct labels in the database. Each entry in the index corresponds to a distinct label \( l \) in the database, and is associated with a list \( S_l \), where \( S_l \) contains the set of identifiers of trees in which label \( l \) occurs. For a given twig with labels \( L_i, 1 \leq i \leq m \), we probe the index to obtain corresponding lists \( S_i \)’s. The intersection of these sets (i.e., \( \bigcap S_i \)) then contains the set of trees which contain all the twig’s labels. We run our tree matching algorithm on each of the resulting set. If the obtained lists from the index are large enough to make the intersection operation very expensive, we can resort to an heuristic. According to which, one can simply choose a list with the smallest size. A key highlight of our index structure is that it is independent of the tree matching algorithm because the index is probed before the algorithm is applied. Index is primarily employed to reduce the search space. In contrast, most of the state-of-the-art methods leverage the index structure in the subsequence matching phase.

**Index Maintenance:** In any case of insert, delete, or update the changes to be made to the index are minimal as only the lists of modified nodes are affected. However, in the case where a tree is updated by adding or deleting the nodes, few NPS entries might have to be changed to reflect the new post-order traversal numbers. In the worst case, when the change is at the left-most-leaf, the entire NPS needs to be updated. To reduce the impact of such cases, a batched lazy-update approach can be employed where a set of updates are grouped and applied on the database as a batch.

**Index size:** The size of our index structure is completely dependent on the distribution of labels in the database. If the database trees are highly associative then the size of each list is approximately equal to the database size. However, we observed that most of the real datasets exhibit a highly skewed distribution of labels i.e., very large number of labels occur in small number of trees, and few labels occur in a large number of database trees, thus have large lists. In such cases, we can reduce the space occupied by our index by stripping the index from least selective information. If the selectivity of a label is very low i.e., if the label occurs in a significant portion of the database, then the prunability offered by its corresponding list is very low. Therefore, we can remove the list from the index without incurring much penalty. We refer to such an index as an \( \alpha \)-infrequent index.
Definition: 2.2.3. An index is called $\alpha$-infrequent if it stores the tree identifier lists for only those labels which appear in less than a fraction of $\alpha$ trees in the database.

For example, a 0.8-infrequent index does not store lists for labels which appear in more than 80% of trees. A 1-infrequent index maintains lists for every label in the database and a 0-infrequent index is an empty index with no lists. If the list for a particular label of interest is not available then we consider its list as the set of all database trees. Therefore, in a worst case, when all the nodes in twig have very low selectivity and all the corresponding lists are not present in the index, then we apply our tree matching algorithm on every database tree. In other words, we resort to the naive approach in a worst case – a trade-off between space and time. Note that, if needed, our $\alpha$-infrequent index can easily be made out-of-core by storing few large lists on the disk. Therefore, the database administrator can choose either to completely delete the list from index or to push the list on to disk.

2.2.4 Optimizations

The main drawback of our tree matching algorithm is that it suffers from a lot of computational overhead. There are two sources of overhead, one due to excessive number of recursions while backtracking, and the other is due to the number of false positives that are fed into the structure matching phase. We now present three different optimizations which reduce the overhead as much as possible. The first two try to alleviate the recursion overhead and the third one attempts to avoid the false positives.

Label Filtering (LF)

This optimization relies on the distribution of labels in a database tree ($T$) over the nodes in the given twig query $Q$. It is based on the observation that the number of distinct labels in $T$ is usually a lot higher than the number in $Q$. Consider a partitioning of labels in $T$ into two mutually exclusive subsets, $V_1$ and $V_2$, such that $V_1 = \{l | l \in T, l \in Q\}$ and $V_2 = \{l | l \in T\} - V_1$. The values in $R$-matrix columns corresponding to the labels in $V_2$ simply carry the values of LCS values from one column to another and thus do not contribute in building the subsequence match and hence they can safely be excluded. In other words, $LS_T$ can be pruned from the labels in $V_2$ before constructing the $R$-matrix. For example in Figure 2.4, the columns corresponding to labels $C$, $E$, and $F$ are removed. This pruning process
is called as Label Filtering. The advantages of label filtering are two-fold. First, it reduces the size of $R$-matrices thereby making them fit in few cache lines. Second, it reduces the recursion overhead in Algorithm 2 as the irrelevant entries in $R$-matrix are now discarded.

### Dominant Match Processing (DM)

In cases where the set of all distinct labels in both the query tree and the database tree are almost same, the benefits from label filtering would be marginal. This optimization is designed to further reduce the number of recursions by limiting the backtracking to a very few entries, called as dominant matches, in the $R$-matrix.

Based on the values, the entries in $R$ can be partitioned into classes: $C_k = \{(i, j)| \ R[i, j] = k\}$, $1 \leq k \leq m$. For the example matrix in Figure 2.4, $C_3$ is $\{(3, 7), (3, 8), (3, 9)\}$. Algorithm 2 accesses the matrix elements in the decreasing order of their class because the match is constructed from right to left. Backtracking from $R[i, j]$ proceeds into two regions of the matrix – $R_1$: from $R[1, 1]$ to $R[i-1, j-1]$ and $R_2$: from $R[1, 1]$ to $R[i, j-1]$. Therefore, while extending the match from $R[i, j] \in C_k$ only those elements from these two regions which belong to $C_{k-1}$ need to be considered. For example in Figure 2.4, recursive calls from $R[3, 6]$ need not be made for elements with value same or more than $R[3, 6]$ (e.g, $R[2, 4]$).

Furthermore, only few elements of $R$, for which the second condition in Equation 2.2.1 holds true, contribute towards the subsequence match. These entries are identified at line 2 in Algorithm 2. Other elements, which satisfy the third condition in Equation 2.2.1, simply carry forward the LCS length through recursive calls. Say $R[i, j]$ and $R[k, l]$ ($i < k; j < l$) are two entries at which the LCS length is increased. Furthermore, assume that $\exists x$ such that $j < x < l$, and $LS_Q[k] = LS_T[x]$. In Algorithm 2, backtracking from $R[k, l]$ can directly jump to $R[i, j]$ as the intermediate cells simple carry the LCS value from $R[i, j]$ to $R[k, l]$. We refer to the cells $R[i, j]$ and $R[k, l]$ as dominant matches. In Figure 2.4, dominant matches for $Q_1$ in $T_1$ are encircled. Entries $R[2, 6], R[3, 5]$, and $R[3, 6]$ can be ignored from processing as they just carry the LCS value from $R[3, 7]$ to $R[2, 5]$.

Considering only the dominant matches imply that elements need to be considered in the decreasing order of their row & column indices (i.e., region $R_1$). The decreasing order is because the match is constructed from right to left. Therefore, backtracking from $R[i, j] \in C_k$ can now be limited to just dominant matches in $R_1$ which belong to $C_{k-1}$. Such a stringent condition drastically reduces the number of recursions while
matching the subsequence (see Section 2.2.5 for analysis). Furthermore, no special data structure is needed to implement this optimization – dominant matches can just be stored as negative numbers in $R$.

In Figure 2.4, backtracking from $R[2, 5] \in C_2$ needs to consider both $R[1, 4]$ and $R[1, 2]$ from $C_1$ in that order whereas processing from $R[2, 3] \in C_2$ needs to consider a single element $R[1, 2]$ from $C_1$.

**Simultaneous Subsequence and Structure Matching**

Both the optimizations $LF$ and $DM$ are targeted at the subsequence matching phase. They do not try to reduce the number of false positives fed into the structure matching phase. For $Q_1$ and $T_1$ in Figure 2.4, only 4 of the 6 subsequence matches are actual subtree matches. These false positives are identified and filtered only in the structure matching phase. They, when in large number, add a significant filtering overhead and may hence hinder the performance. This overhead is alleviated by detecting the false positives as early as possible.

Observe that both the subsequence and structure matching algorithms process the twig query from right-to-left. Therefore, instead of completely generating all the subsequences and then performing the structure matching, we can perform them simultaneously. As soon as label match is established at position $k$ ($1 \leq k \leq m$ (i.e., for $LS_{Q}[k]$)), the structure agreement check at that position is performed. In other words, the structure matching is integrated into the process of subsequence matching. Such unification is feasible because a structure agreement check for a node at position $k$ needs only its ancestors, for which the structure match is already established. By pushing the structure constraints deep into the process of subsequence matching the unified approach yields a better performance [148]. This is because we detect and prune them as early as possible, and thus we never generate the false positives completely.

2.2.5 Putting it all together

Our complete unified tree matching algorithm to find the set of all twig matches for $Q$ in $T$ is shown in Algorithm 4. $T$’s label sequence is first filtered from labels which are not in $Q$, using $LF$. Equation 2.2.1 is then used to construct the $R$-matrix. While constructing the matrix, the dominant matches are marked whenever both the
Algorithm 4 *LCS-TRIM*: Algorithm for subtree matching

**Input:** A database tree $T$ and a twig query $Q$

$\text{labelFilter}(T, Q) \{T \text{ contains the filtered sequence}\}$

$R \leftarrow \text{computeLcsMatrix}(T, Q)$

if $R[m, n] \neq m$ then
  Report that $Q$ is not a subtree of $T$

$SM \leftarrow \text{null}$

$\text{processLCS}(m, n, m)$

**Function:**

$\text{processLCS}(Q\text{ind}, T\text{ind}, \text{matchLen})$

1: if $\text{matchLen} = 0$ then
2: Report $SM$ as the twig match
3: for $i = T\text{ind}$ to 1 do
4: if $R[Q\text{ind}][i]$ is dominant & $R[Q\text{ind}][T\text{ind}] = \text{matchLen}$ then
5: if $\text{isInAgreement}(\text{CPS}(Q), SM, Q\text{ind})$ then
6: $SM[Q\text{ind}] \leftarrow \text{CPS}_T[T\text{ind}]$
7: $\text{processLCS}(Q\text{ind}-1, T\text{ind}-1, \text{matchLen}-1)$

label sequences match as per the second condition in Equation 2.2.1. The function $\text{processLCS}$ is invoked to enumerate the set of all twig matches.

For a given position $Q\text{ind}$ in the twig, $\text{processLCS}$ tries to find a label match in $T$ (line 3). As soon as a match is found, Algorithm 3 is executed at the matched location by the function $\text{isInAgreement}()$ (line 5). The resulting structure match is recorded in $SM$ at line 6. If the user is aware of the exact document structure then the user can provide the exact level difference between a node and its parent node. Such *level-wise constraints*, which can easily be incorporated into $\text{isInAgreement}()$, extend the capabilities of standard wildcards (‘/’ and ‘*’) and improve the matching process significantly. The twig match is recursively extended by backtracking in line 7 and is reported in line 2.

**Complexity Analysis:** Again, since the number of enumerated twig matches themselves are exponential in number, the over time complexity of the tree matching algorithm is exponential. We thus resort to analyze the number of recursions made by our unified algorithm in a worst case where all the tree nodes have the same label. Similar to $a_{m,n}$, let $c_{m,n}$ be the number of recursions made by Algorithm 4 in the worst case. Since only the dominant matches from region $R_1$ are processed, Equation 2.2.5 involves just one term $c_{m-1,n-1}$. $c_{m,n}$ has a closed form of $\binom{n+1}{n-m+1}$. Clearly,
Table 2.2: Data sets for evaluating *LCS-TRIM*

<table>
<thead>
<tr>
<th>Data set</th>
<th>Domain</th>
<th># of Trees</th>
<th>Max Depth</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swissprot</td>
<td>Bioinformatics</td>
<td>50,000</td>
<td>5</td>
<td>[4]</td>
</tr>
<tr>
<td>Treebank</td>
<td>Linguistics</td>
<td>52,851</td>
<td>36</td>
<td>[4]</td>
</tr>
<tr>
<td>DBLP</td>
<td>WWW</td>
<td>328,858</td>
<td>6</td>
<td>[4]</td>
</tr>
<tr>
<td>Cslogs</td>
<td>Web Mining</td>
<td>59,691</td>
<td>85</td>
<td>[207]</td>
</tr>
<tr>
<td>NLM</td>
<td>Medicine</td>
<td>450K-1M</td>
<td>8</td>
<td>[142]</td>
</tr>
</tbody>
</table>

label filtering and dominant match processing significantly reduces the number of recursions. For example, while finding matches for a twig of size 5 in a tree of size 10, the unified algorithm makes 462 recursive calls whereas the naive algorithm makes 1,275 number of recursions.

\[
c_{m,n} = \begin{cases} 
1 + \sum_{i=1}^{n-m+1} (1 + c_{m-1,n-1}), & \text{if } n > m \\
0, & \text{if } n = m \lor m = 1 
\end{cases} \quad (2.2.5)
\]

To summarize, we leverage a *dynamic programming* based approach instead of a traditional index-based approach for finding the set of all subsequence matches. Our *unified* matching algorithm and employed *optimizations* are expected to significantly reduce the query processing time. The nature of our approach and its reliance on sequential encoding and matrix or array based data structures reflects a cache conscious design with *small memory footprints*. Furthermore, the avoidance of pointer-based data-structures ensures that the instruction level parallelism of the approach will not be affected. Finally, our algorithms can easily be deployed on parallel systems, including shared-memory multicore systems and distributed cluster systems (see Section 5.1.3 and Section 5.2.3). For example, in a distributed setting, the database is partitioned across different processors. Each processor computes the matches for a given query from its local partition, and the results are combined to form the final answer (see Section 5.2.3).

### 2.3 Empirical Evaluation

We now evaluate the performance of our algorithm LCS-TRIM against PRIX and three different *TwigStack* algorithms: TwigStack (uses no index), XBTwigStack (uses XB tree), and XRTwigStack (uses XR tree – TSGeneric+ in [103]). We have obtained
<table>
<thead>
<tr>
<th>Qid</th>
<th>XPath Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//Entry[PFAM[@prim_id=PF00304]]//@SIGNAL/Descr</td>
<td>3</td>
</tr>
<tr>
<td>Q2</td>
<td>//Entry[Org]//@SIGNAL/Descr</td>
<td>39</td>
</tr>
<tr>
<td>Q3</td>
<td>//Ref/Author=“Moss J”</td>
<td>38</td>
</tr>
<tr>
<td>Q4</td>
<td>//Entry [Species=“Glycine max”][Organe=“Chloroplast”][Org= “Glycine”]</td>
<td>9</td>
</tr>
<tr>
<td>Q5</td>
<td>//Features/DOMAIN[from=“165”][to=”171”][Descr=”POLY-PRO”]</td>
<td>1</td>
</tr>
<tr>
<td>Q6</td>
<td>//Features/[*/from][”171”][”POLY-PRO”]</td>
<td>14</td>
</tr>
<tr>
<td>Q7</td>
<td>//Features[”165”][*/to][”POLY-PRO”]</td>
<td>51</td>
</tr>
<tr>
<td>Q8</td>
<td>//Entry/Features/DOMAIN</td>
<td>46,566</td>
</tr>
<tr>
<td>Q10</td>
<td>//Entry[<em>/”Eukaryota”][</em>/”Metazoa”][<em>/”Chordata”][</em>/”Craniata”]</td>
<td>17,836</td>
</tr>
</tbody>
</table>

Table 2.3: Swissprot workload

<table>
<thead>
<tr>
<th>Qid</th>
<th>XPath Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q11</td>
<td>//inproceedings/*/“Antonin Guttman”</td>
<td>2</td>
</tr>
<tr>
<td>Q12</td>
<td>//article/author=“Antonin Guttman”</td>
<td>3</td>
</tr>
<tr>
<td>Q13</td>
<td>//phdthesis/year</td>
<td>series</td>
</tr>
<tr>
<td>Q14</td>
<td>//phdthesis/year</td>
<td>number</td>
</tr>
<tr>
<td>Q15</td>
<td>//inproceedings/author=“E. F. Codd”</td>
<td>33</td>
</tr>
<tr>
<td>Q16</td>
<td>//book/[AA93][AABM82][AB87a][AB87b][AB88][AB91][ABD+89]</td>
<td>1</td>
</tr>
<tr>
<td>Q17</td>
<td>//article/year=1999</td>
<td>7,408</td>
</tr>
</tbody>
</table>

Table 2.4: Dblp workload

<table>
<thead>
<tr>
<th>Qid</th>
<th>XPath Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q18</td>
<td>//EMPTY/*/LS_OR_JJ</td>
<td>2</td>
</tr>
<tr>
<td>Q19</td>
<td>//S/*/RB_OR_JJ</td>
<td>8</td>
</tr>
<tr>
<td>Q20</td>
<td>//S/SBARQ-1</td>
<td>34</td>
</tr>
<tr>
<td>Q21</td>
<td>//NP/ADJP/IN_OR_RB</td>
<td>1</td>
</tr>
<tr>
<td>Q22</td>
<td>//S[PRT][NP]</td>
<td>2</td>
</tr>
<tr>
<td>Q23</td>
<td>//EMPTY/*/X/VP/PP/NP/S/VP/VP/NP</td>
<td>107</td>
</tr>
<tr>
<td>Q24</td>
<td>//EMPTY/<em>/NP/</em>/X/<em>[VBN]/</em>[WRB]/<em>/[S/</em>/NONE_]/[VBG]]</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2.5: Treebank workload

<table>
<thead>
<tr>
<th>Qid</th>
<th>XPath Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q25</td>
<td>//1/3[/545]</td>
<td>26</td>
</tr>
<tr>
<td>Q26</td>
<td>//2500[7695][2501]</td>
<td>72</td>
</tr>
<tr>
<td>Q27</td>
<td>//1155/[4525][2613/5888]</td>
<td>82</td>
</tr>
<tr>
<td>Q28</td>
<td>//1155/[5996][7834][2586]</td>
<td>85</td>
</tr>
<tr>
<td>Q29</td>
<td>//1155[2586][5996]</td>
<td>3,742</td>
</tr>
<tr>
<td>Q30</td>
<td>//1/271/272/273/276/278/281/284/287/1552</td>
<td>78</td>
</tr>
<tr>
<td>Q31</td>
<td>//5191[8650][8686][8685][8684][9400][8680]</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.6: Cslogs workload
Table 2.7: NLM workload

PRIX from its open source distribution [59] and TwigStack algorithms from the authors of [103]. All the experiments, unless otherwise noted were performed on a system with dual AMD 250 Opteron processors and 8 GB of main memory. We have found that the performance of LCS-TRIM using DFS is quite similar to the results obtained using Prüfer sequences, the ones we present here.

Data sets: We consider five different data sets – Swissprot, Treebank, DBLP, Cslogs, and NLM (see Table 2.2). These five data sets are derived from five different application domains and have different characteristics. Swissprot (curated protein), Treebank (syntactic structure of English sentences), and DBLP data sets are obtained from the University of Washington XML data repository [4]. Cslogs data set, that represents the website access patterns, is obtained from Dr. Zaki’s website [207]. Finally, NLM data set [142] houses abstracts from MEDLINE and other life science journals for biomedical articles. These XML data sets are parsed using a SAX parser [2] and the output is then converted into sequences using methods in Section 2.2.1.

Query workload: The query workloads with which we evaluated our algorithms are shown in Tables 2.3, 2.4, 2.5, 2.6, 2.7. The queries are carefully chosen such that each one has different characteristics – large and small queries, deep and bushy queries, low and highly selective queries. We first show the comparative results against PRIX, then we present the effect of workload characteristics on the performance of our algorithms.

Effect of optimizations: The performance of our algorithm with and without optimizations on the Swissprot query workload is shown in Figure 2.5a. The basic algorithm with no optimizations is labeled as No Opt and the effect of subsequence matching optimizations is shown as LF & DM. Note that our No Opt algorithm is similar in structure, and expected to exhibit comparable performance, to the methods proposed by Zezula et al [211] because one of the main differences between LCS-TRIM and these methods are the use of our optimizations. The recursion overhead is greatly reduced by both LF and DM. They, for example on $Q_9$, showed a 94-fold speedup
that is due to two reasons – the reduction in the average $R$-matrix size from 28KB to 8KB and the reduction in the number of recursive calls to Algorithm 2 from 16 billion to 5 million. The optimizations targeting structure matching in conjunction with subsequence matching, denoted $LCS$-$TRIM$, are very effective across the board. For example, looking at the query $Q_9$ again, we find that of the 3.2 million subsequence matches only 17,836 of them are actual structure matches. A large majority of these false-positives are filtered out by our unifying optimization in the early stages resulting in an overall speedup of 345 for this query against No Opt.

### 2.3.1 Comparison with PRIX

Figures 2.6 demonstrate the performance differences between LCS-TRIM and PRIX. LCS-TRIM achieves a significant speedup over PRIX in enumerating all the twig matches – on average, up to 3 orders of magnitude speedup on DBLP and up to 2 orders on other data sets.

This huge disparity in run times is due to various reasons. First, the subsequence matching step in PRIX is very expensive. Too many range queries issued during the process increase the number of accesses to the index and the large $B^+$-tree index structures (see Figure 2.7) make those accesses very costly. Second, once we factor in the need to use the $E$Pindex (index constructed over extended prüfer sequences) for value-based queries the number of accesses to the index increases dramatically. For query $Q_4$, there are 2,041 subsequence matches requiring 690 accesses to the index. PRIX then further refines these 2,041 matches using a series of post-processing steps.
to determine that all but only 9 of them are false positives. In contrast, LCS-TRIM filters these the false positives very early by combining the subsequence and structure matching phases. Third, by way of comparison our approach relies on an inverted tree index to identify candidate trees and for each candidate tree we rely on the processing of small $R$-matrices – which can often fit in a few cache lines – leading to excellent locality. Drilling down on this disparity even further in Figure 2.5b we see that in all of the queries the base subsequence match step in PRIX is already more expensive than our method. Factoring in the time for refinements (for PRIX to generate the correct results) adds to the disparity. For example, consider the fact that subsequence matching in PRIX for $Q_7$ took 15,300msec and refinements amount to an additional 29msec. For the same query our approach took just about 2msec. The refinement time for queries $Q_4$ and $Q_5$ is small due to the fact that those queries are small in size and also have a small number of matches. Our overall query processing time is usually less than the time that PRIX spent in refining the subsequence matches.

**Effect of pivots and gap constraints**

PRIX relies on user-provided pivots and gap constraints in order to reduce the number of accesses to the index, while finding the subsequences. PRIX leverages pivots in performing a bi-directional subsequence matching that is done using two separate indexes $LIndex$, $RIndex$ [150]. The performance of PRIX is highly dependent on the pivots chosen. Pivots pointing to low selective nodes may actually increase the number of accesses to the index and thereby may hinder the performance. For example, in queries $Q_{18}$ and $Q_{19}$, the nodes EMPTY and S have low selectivity with 49, 416 and 50, 726 matches, respectively. When they are chosen as pivots, the run time for $Q_{18}$ and $Q_{19}$ increased to 0.15sec and 0.8sec – a significant slowdown when compared to the best pivot choice (0.3msec and 0.6msec as shown in Figure 2.6c). All the reported results for PRIX are based on the best choice. Unfortunately, providing the best pivot value can be challenging for the user. The database engine can of course provide some hints based on selectivity estimation but they may not be optimal. Our methods on the other hand do not depend on such choices. In fact, even when we process the entire data set the retrieval times are not very high – 0.014sec, 0.018sec for $Q_{18}$ and $Q_{19}$, respectively.

*Gap constraints* are specified by the user based on the distance between two elements in the database sequence. For example in $Q_{22}$, occurrences of tag $PRT$ in the database is scattered over several documents. Only 2 of these documents have
Figure 2.6: Evaluation of LCS-TRIM and Prix on different data sets.
\(S\) as \(PRT\)’s parent while in others, \(S\) is an ancestor. PRIX, in this case, relies on gap constraints to restrict the search to parent-child axis. With sub-optimal gap constraints, the total number of index page transfers for \(Q_{22}\) have increased from 1,502 to 79,177, reflecting a significant increase in the number of index accesses. Furthermore, inaccurate gap constraints can lead to inaccurate results. In order to provide accurate gap constraints, the user not only needs to know the database tree structure but also its \textit{internal} representation. Even though a level-wise constraint is similar to a gap constraint, the former depends on a high-level document structure where as the latter depends on internal representation. We now evaluate how different workload characteristics affect the retrieval times.

\textbf{Effect of query size}

The number of nodes in the query tree directly affects the size of \(R\)-matrix and hence the performance of LCS-TRIM. In our workload, small sized queries are \(Q_3, Q_{11}, Q_{12}, Q_{19}, Q_{20},\) and \(Q_{25}\) and large queries are \(Q_5, Q_{16}, Q_{24}, Q_{30},\) and \(Q_{31}\). We have observed an average speedup of 60 and 400 on small and large queries, respectively. Smaller queries will have smaller \(R\)-matrices, which mostly fit in the L1 cache resulting in very good locality. A higher speedup on large queries is due to the slowdown in PRIX culminating from an increased number of index accesses. Hence the query size does not affect our simple matrix-based approach as much as it affects PRIX.

\textbf{Effect of recursive structure}

We now examine how a twig’s recursive structure affects the performance. In order to do that, consider the queries, \(Q_{23}\) and \(Q_{30}\), with a deep structure and the queries, \(Q_4, Q_9, Q_{16}, Q_{24},\) and \(Q_{31}\), which are shallow and bushy. Note that the trees in both Swissprot and DBLP data sets have a small depth (see Table \ref{table:depth}). The average speedup on deep queries (240 times) is a little more than the speedup on bushy queries (190 times). This difference can again be attributed to a large number of index accesses made by PRIX in case of deep queries. On query \(Q_{24}\) alone LCS-TRIM achieved up to 3 orders of magnitude speedup.

\textbf{Effect of selectivity}

The query workload has some highly selective queries (\(Q_5, Q_{16}, Q_{21}, Q_{22},\) and \(Q_{31}\)) and some low selective queries (\(Q_8, Q_9, Q_{17}, Q_{27},\) and \(Q_{29}\)) with many matches. LCS-TRIM showed up to 2 orders of magnitude speedup over PRIX on both the
type of queries. For low selective queries, PRIX spent a lot of time in accessing the index while finding many subsequence matches. The performance of PRIX is very sensitive to the selectivity of individual nodes in the twig. When the subsequence matching starts off with a low selective node, the performance gets severely affected due to increased number of range queries issued to the index. The effect of node selectivity on the performance of our algorithm is relatively small because of the way our inverted index is probed (before the matching process starts) and the way we find the subsequence matches.

**Effect of wildcards**

Finally, we analyze the effect of wildcards on the query processing times of LCS-TRIM and PRIX. The queries $Q_6$, $Q_7$, $Q_{10}$, $Q_{16}$, and $Q_{24}$ have a high number of wildcards. As an example, consider the queries from Swissprot workload. The queries $Q_6$ and $Q_7$ are slight modifications of the query $Q_5$ with 3 wildcards introduced. These simple modifications have increased the run time of PRIX by more than 600 times. Such a drastic increase is due to three reasons. First, wildcards increase the number of true and false positive subsequence matches. Second, false positives are detected only in the later stages of query processing. Finally, the process of filtering out a subsequence is very inefficient in PRIX – makes too many scans on the twig. LCS-TRIM, on the other hand, embeds employs the unified approach (detects the false positives as early as possible) and makes a single pass on the twig to filter it out. Thus the effect of wildcards on the performance of LCS-TRIM is considerably small when compared to the effect on PRIX’s performance. Due to these reasons, we have observed an average speedup of 2,500 on the queries with many wildcards.

**Effect of data set size**

We now test the performance of both the algorithms on large data sets by considering $Q_{32}$, $Q_{33}$, and $Q_{34}$. We used a system with 2.8GHz P4 processor and 1.5GB of main memory for this experiment. Figure 2.6e–g shows the performance differences as the data set size (XML file size) is increased from 0.5GB to 2.5GB. An $\alpha$-infrequent ($\alpha=0.5$) index is used for this experiment.

On all the queries, when the data set size exceeds the main memory size (1.5GB), there is a significant increase in the run time of LCS-TRIM. Note that the graphs may be a bit misleading since the y-axis has a log-scale basis. For example on $Q_{34}$ when going from a data set size of 1.5GB to 2GB, it appears as though LCS-TRIM
suffers a significant slowdown while PRIX only suffers a marginal slowdown. However in reality the execution time of LCS-TRIM goes up to 0.1s whereas the PRIX running time goes up from 1.5s to 2.5s. Essentially once we go out of core the disk latency dominates for both the methods and as a result of Amdahl’s law the relative speedup of LCS-TRIM (10% CPU utilization) with respect to PRIX is reduced. LCS-TRIM is still at least one order of magnitude better on all the queries and in some 3 orders of magnitude better. Such huge speedup is observed in case of queries with complex structure and many wildcards (e.g., \( Q_{32} \)). We are currently investigating ways in which the out-of-core performance can be further improved using strategies based on data partitioning and tiling.

**Comparison of Index Size**

In PRIX, the index is constructed based on user-provided *list of tags*, which correspond to labels in the query workload. A \( B^+\)-tree is created for each tag in the list. Therefore the total size of the index grows linearly with the number of tags. Figure 2.7 shows the index sizes along with the number of tags used, for each data set. The figure also shows the size of 1-in frequent index constructed by LCS-TRIM. The huge difference, between PRIX and LCS-TRIM, in index size is clearly evident – compare the 256MB DBLP index in PRIX with our index of size just 48MB. Furthermore, when a 0.5-in frequent index is created the size reduced to 35MB. Even for large data sets like NLM, our \( \alpha \)-infrequent inverted index is quite small. In contrast, index structures in PRIX are very large and are usually more than double the data set size – at 2.5GB data set the index size is almost 5GB. To build a generic query processing engine, the *tag list* should contain a large number of labels resulting in very large index structures. More importantly, each tid-list in our inverted index is *bounded by the number of trees* in the data set whereas the size of each \( B^+\)-tree in PRIX is *proportional to the selectivity of labels*, which is usually very high.

**2.3.2 Comparison with TwigStack-style algorithms**

In this section we compare the performance of LCS-Trim with the TwigStack family of algorithms\[37, 54, 103\]. We should note that TwigStack based algorithms target unordered matches whereas LCS-TRIM targets ordered matches. Later in Section 2.3.3 we discuss some ideas on how to extend LCS-Trim to handle ordered matches. To ensure as fair a comparison as possible, we follow the method employed in PRIX [150],
Figure 2.7: Index size comparison on different data sets

Figure 2.8: Evaluation of \textit{LCS-TRIM} and TwigStack-style algorithms
wherein for each query in our workload the number of ordered matches is exactly the same as the number of unordered matches. This is because only one configuration of the twig is present in the data set.

As found in [103], the performance of XR TwigStack is marginally better than the other two algorithms (see Figure 2.8). We have run this experiment on a Windows PC with a 2.8GHz Intel Pentium 4 processor and 1.5GB of main memory. Note that the y-axis is shown in log-scale. LCS-TRIM performed significantly better than XR TwigStack on all queries – up to 3 orders of magnitude. The speedup is very high, especially on queries with low selectivity, because the TwigStack algorithms make a large number of scans on huge element lists.

2.3.3 Extensions

Unordered matching
The sequencing methods in Section 2.2.1 are based on the tree traversals and hence they preserve the order among sibling nodes. Therefore, Algorithm 4 can only be applied to process ordered twigs. A typical but naive method to support unordered matches is as follows [150]: First, the set of all configurations ($C$) of the given twig is found. Second, for each configuration in $C$, matches are found in each database tree using Algorithm 4. The set $C$ is potentially exponential in the size of the query. However, twig queries are usually small in size and hence processing all the configurations in $C$ may not be very expensive.

There exist several ways to reduce the number configurations which are to be evaluated – by imposing the DTD constraints, by a suitable preprocessing of the data set, and by imposing an artificial ordering on node labels of both the database tree and the query tree. DTD constraints for example can determine that certain configurations are illegal and can thus be eliminated from processing. In case of artificial ordering, the data set is initially pre-processed by rearranging the sibling nodes so that they respect the ordering. Nodes in the given twig query are also rearranged in a similar manner so that the matching can be done. For example, consider the query $Q_{31}$ with 6 different child nodes. There are a total of $6! = 720$ different configurations to evaluate. When a label ordering, say a numerical ordering $8650 < 8680 < ... < 9400$, is placed on both the data set and the query then we need to evaluate only a single configuration. In fact, the label ordering gave a single valid configuration for every query in our workload.
Figure 2.9: Heuristics for Unordered Subtree Matching

When the application of constraints and orderings is not possible, one can employ some heuristics. For example, the naive approach can be improved by doing a loop inversion in the second step – for each tree in the database, process all configurations in $C$ to find their matches. A tiling-like approach can then be used to exploit the fact that many configurations are quite similar. For example, Figure 2.9 shows two configurations $S_1$ and $S_2$. $S_2$ can be eliminated from processing because the position of mismatch in $S_1$ is away from the portion of the sequence that is changed in $S_2$. Such a heuristic complemented with a suitable ordering of configurations in $C$ can greatly improve the performance. We are currently investigating these and other ways to extend LCS-TRIM to handle unordered matches.

**NOT Predicates**

Consider the twig query, $NQ=//supplier/[NOT(//store-)]//part$ [205]. It selects the part elements with an ancestor supplier that has no descendant named store. There exist several ways to incorporate such constraints in query processing [105, 205]. A simple but inefficient approach is to divide the query into two sub queries – one with the tag (store) and one without the tag. Then take the difference between the results from those two sub queries.

LCS-TRIM handles NOT predicates by first constructing the $R$-matrix as described in Section 2.2.2, by ignoring the NOT-predicate on store tag. It checks for following two conditions to find a tree $T$ that does not have the tag store. i) the length of LCS between $T$ and $NQ$ should be $m - 1$. ii) the number of dominant matches in the row, in $R$, corresponding to store should be zero. Moreover, the structure agreement checks ignore the store tag in $NQ$ and proceed to its parent nodes. More involved algorithms ought to be designed to support nested NOT predicates.
Approximate Query Matching

In case a query does not have any matches, instead of returning no results it would be nice to have a query engine that can return approximate results. Approximation can either be structural or content-oriented i.e., values. LCS-TRIM can be modified with a simple cost-based model to retrieve such approximate results. Each mismatch (either in structure or in content) is tagged with certain cost and the tree matching, both subsequence and structure matching, is continued as long as the total cost of mismatch is less than a pre-defined threshold. A domain expert can define the threshold and also the type and cost of mismatches.

2.4 Conclusions

We proposed a indexing mechanism $LCS$-$TRIM$ for querying XML data repositories. It relies on a concise injective transformation between rooted ordered trees and sequences. The query processing phase leverages a dynamic programming based classical longest common subsequence matching algorithm to enumerate query matches from the database. We designed a series of optimizations to prune false candidate matches early in the search process, thereby enabling efficient matching and retrieval. In contrast to state-of-the-art tree indexing methods, we leverage a simple inverted index structure that complements our efficient subtree matching algorithm in pruning the search space. Through a detailed empirical evaluation on several real-world data sets and workloads, we demonstrated that $LCS$-$TRIM$ exhibits up to three orders of magnitude speedup over existing methods. We presented mechanisms that enable the user to specify constraints on the retrieved output and showed how they can be pushed deep into the retrieval process. Our framework can also support the retrieval of approximate matches. Our algorithms exhibit very good data locality as they operate on simple array based data structures, and they are amenable for parallel query processing on emerging multi-core server architectures (discussed later in Chapter 5).
CHAPTER 3
HASHING TREE-STRUCTURED DATA

Advances in data collection and storage technology have led to a proliferation of information available to organizations and individuals. This information is often also available to the user in a myriad of formats and multiple media. With the increasing importance given to semantic web [66] and Web 2.0 technologies, languages such as XML and collaborative community systems like DBLife [68], an increasing number of these data stores are housed in (semi-)structured formats. Examples abound ranging from XML data repositories to directory information on modern file systems, from MPEG-7 repositories [191] to linguistic data [47] and from social network data [183] to phylogenetic data [170]. With the increasing use of such structured datasets, the need for efficiently managing, querying and analyzing such data is thus growing.

A fundamental operation, in a number of domains, for example in database systems, data mining, computational geometry and network science, is that of hashing. A hash function is a procedure that maps a large, possibly variable sized piece of information into a small fixed sized datum. Hash functions are ubiquitous in their use and are primarily used as a tool to improve the efficiency of search, for example in finding the items in a database, detecting duplicate records [74, 112], and localizing related data for subsequent analysis [38] etc. Although the idea of hashing dates more than 50 years ago, much of the work to date has primarily focused on the hashing of (variable-sized) sets [35, 60], documents [36], sequences [9] and geometric objects [193]. Hashing and sketching tree- and graph- structured data is not so well understood although it has been the focus of recent research [22, 21, 87, 203, 83].

In this chapter, we primarily focus on the problem of hashing and sketching tree-structured data. For expository simplicity we focus on rooted trees although many of our ideas apply for free trees and directed acyclic graphs as well (not discussed further). Our approach relies on two operators – a transformation operator that converts a tree-structured dataset into a (multi-)set of pivotal elements, and a signature-sketch operator that converts the (multi-)set into a fixed datum that
can subsequently be used in estimating the similarity among trees. We investigate the theoretical properties (e.g. efficiency, perfect hashing, edit-distance bounds) of several possible transformation operators and describe efficient mechanisms to compute them. For the signature-sketch operator we primarily focus on the use of min-wise hashing [35, 60, 36] although alternate strategies (e.g. locality sensitive hashing [16, 85, 102], bloom filters) may also be options to consider in the future.

In addition to the theoretical analysis we present a comprehensive empirical study to compare and contrast the proposed strategies on multiple synthetic and real datasets. We have evaluated our methods along the axes of efficiency, storage costs and performance from the perspective of the application domain. The benefits of the proposed approach are showcased on different application domains, including XML de-duplication, stratified sampling of tree-structured data for mining frequent patterns on web log data and phylogenetic data analysis.

To reiterate, the key contributions of our study include:

- Novel incremental, transformation operators based on the notion of induced, embedded and constrained pivot structures that map a given tree-structured datum into a multi-set.

- Signature-sketch operators based on min-wise hashing to convert the resulting multi-set from the transformation operator into a fixed sized datum which can subsequently be leveraged in estimating the similarity between different trees.

- Theoretical results showing that one of the transformation operators we propose can form the basis for a perfect hash function and lower bounds relating the proposed operators to traditional tree edit distance measures.

- Empirical results on a range of datasets and application scenarios demonstrate efficacy and efficiency of the proposed methods.

**Chapter Organization:** We first describe our framework for hashing tree-structured data in Section 3.1. In Section 3.2, we conduct a thorough theoretical study on the properties of our hash functions. A detailed evaluation of our algorithms and two application case studies are presented in Section 3.3. We finally conclude this chapter in Section 3.5.
3.1 Sketching a Tree Structure

Our approach for constructing signatures for a given tree relies on two main functions viz. a transformation function and a signature function. While the former is responsible for transforming the tree into meaningful substructures, the latter is accountable for constructing shorthand sketch or fingerprint for the given tree. The produced signatures can then be used in conjunction with methods like locality sensitive hashing [16, 85, 102] to perform a variety of operations such as searching for nearest neighbors and grouping similar structured trees. We now present our transformation and signature functions, and illustrate their use in the context of computing the similarity between two given trees.

3.1.1 Transformation Function (tf)

A tree structure in terms of set theory can be thought of as a partially ordered set of elements that are ordered by a relation namely parent-child. This relation alone induces all other associations among nodes such as ancestor-descendant and sibling. Not only does the tree structure impose different relations among tree nodes but in converse these relations also precisely define a tree structure. Therefore, one possible way to compare different tree structures is to examine the relationships that are preserved among given trees – the fundamental motivation behind our approach. It relies on a transformation function that maps a given tree into a multiset of substructures known as pivots where each pivot individually captures a specific relationship present among the nodes involved in the substructure. For the sake of explanation, we define a particular type of pivot structure here that is called as an embedded pivot. We later show how pivot structures can be adapted to capture the notion of tree similarity that the underlying application demands.

Definition: 3.1.1. Embedded Pivot: An embedded pivot containing two nodes $u, v \in T$ is defined as the tuple $(lca, u, v)$ where $lca$ is the lowest common ancestor of $u$ and $v$.

An embedded pivot encapsulates the association between $u$ and $v$ by capturing the ancestor-descendant relations $lca - u$ and $lca - v$. Since $lca$ may not be the direct parent of $u$ and $v$, the pivot structure is in fact a wedge-shaped ($\wedge$) embedded subtree of $T$. The set of all pivots involving a particular node $w \in T$, denoted $\mathcal{S}_T(w)$, describes the tree structure when seen from the perspective of $w$ (see Table 3.1).
set can be divided into two subsets – the one in which \( w \) is the root and the one in which \( w \) is one of the two child nodes. We thus have:

\[
S^T(w) = S^T_{\text{root}}(w) + S^T_{\text{child}}(w), \quad \text{where}
\]

\[
S^T_{\text{root}}(w) = \{(lca, u, v) | lca = w\} \quad \text{and} \quad S^T_{\text{child}}(w) = \{(lca, u, v) | u = w \lor v = w\}
\]

The set of all pivots associated with a tree \( T \) is then given by \( S(T) = \bigcup_{w \in T} S^T(w) \).

| \( T \) \(|T|\) | A branched unordered tree (and its size) |
| \( S(T) \) | The multiset of all pivots in \( T \) |
| \( \text{sig}(T) \) | The minhash signature of \( T \) |
| \( r(T), d(T) \) | The root and height (or depth) of \( T \) |
| \( T(v) \) | Subtree rooted at \( v \) (including \( v \)) in \( T \) |
| \( A(v) \) \((D(v))\) | Set of all ancestors (descendants) of \( v \) – not including itself |
| \( S^T(v) \) | All pivots in \( T \) that contain a node \( v \) |
| \( S^T_{\text{root}}(v) \) | All pivots in \( T \) in which \( v \) is the root |
| \( S^T_{\text{child}}(v) \) | All pivots in \( T \) in which \( v \) is one of the two children |
| \( d_v \) | Depth or level of a node \( v \). If \( v \) is the root node then \( d_v = 0 \) |
| \( \mathbb{T} \) | Set of all unordered trees |
| \( \mathbb{S} \) | Set of all pivot multisets |

Table 3.1: Notation

In case of labeled trees, each pivot stores the node labels. A pivot would then be described as \((l(lca), l(u), l(v)) \) where \( l(\cdot) \) is a label function. Since multiple nodes can have the same label, there may be repetitions in \( S(T) \), making it a pivot multiset (or a bag) instead of a pivot set. Throughout this chapter, we use the words multiset and set interchangeably. We also ignore the label function \( l(\cdot) \) whenever possible for notational convenience. We use the hyphen ('-') as a wildcard symbol in the pivot. For example, \((w, -, -)\) refers to the set \( S^T_{\text{root}}(w) \). We further abuse the use of – to omit certain fields in the pivot that are not important for the discussion.

A simple method to construct the multiset of embedded pivots is shown as Algorithm 5. \( rml(v) \) in Line 3 denotes the pre-order number of the right most leaf node in \( T(v) \), the subtree that is rooted at \( v \). The algorithm operates on a particular orientation (a fixed arrangement) of the given unordered tree. Every iteration in Line 2 produces the pivots in which \( u \) is one of the child nodes. In Line 3, \( u \) is paired with all the other nodes \( v \in T \) where \( v \) is neither an ancestor nor a descendant of
Algorithm 5 Transformation function that constructs embedded pivots

1: $n \leftarrow |T|$
2: for each $u$ in $T$ in pre-order do
3:   for each $v$ in $T[rml(u) + 1 \cdots n]$ do
4:     $lca \leftarrow$ lowest common ancestor of $(u,v)$
5:     if $lca \neq v$ then
6:       if $l(u) > l(v)$ then
7:         swap($l(u), l(v)$)
8:       add pivot $(l(lca), l(u), l(v))$ to $S(T)$

In case of unordered trees, the node labels are arranged in an order (Lines 5–6) to make sure that two different orientations of the unordered pivot are treated to be identical. Such an artificial order guarantees that all orientations of an unordered tree produce the same (multi)set of pivots. The computational complexity of the algorithm is equal to the output size, which, in a worst case, is quadratic $O(n^2)$ in tree size $n$. We would like to note that it is fairly easy to parallelize Algorithm 5 (see Section 3.3). Such parallel strategies are of great importance in the context of modern day multicore server systems.

The total number of pivots $N(T)$ in $T$ can be computed as a function of tree size $n$ by using Algorithm 5. In the following derivation, $pre(v)$ and $|D(v)|$ refers to the pre-order number and the number of descendants of $v$, respectively (see Table 3.1).

$$N(T) = \sum_v (rml(v) + 1 \cdots n) = \sum_v [n - pre(v) - |D(v)|]$$

$$= n^2 - \frac{n \cdot (n + 1)}{2} - \sum_v |D(v)| - \frac{n \cdot (n - 1)}{2} - \sum_v |D(v)|$$

The number $N(T)$ reaches its maximum when the sum of all descendants $\sum_v |D(v)|$ is minimum, as in the case of a bushy tree shown in Figure 3.1a. For simple chain trees with $n - 1$ edges as in Fig. 3.1b, $N(T)$ reaches its lower bound.

$$\min(\sum_v |D(v)|) = n - 1 \Rightarrow \max(N(T)) = \frac{(n - 1) \cdot (n - 2)}{2}$$

$$\max(\sum_v |D(v)|) = \frac{n \cdot (n - 1)}{2} \Rightarrow \min(N(T)) = 0$$
We handle these chain trees with zero pivots as special cases and we assume, without loss of generality, that all trees are branched i.e., the root has at least two children. Note that a chain tree can be made as a branched tree by introducing a dummy child node. Among branched trees with \( n \) nodes, the lower bound on \( \mathcal{N}(T) \) is \( n - 2 \) (see Figure 3.1c).

The similarity between two trees can be established by comparing their respective pivot sets. This is possible because the structural relationships among nodes, which govern the tree similarity, are effectively captured in pivot substructures. We would like to note that simple intuitive transformations are not useful here. Consider a mapping \( f \) that maps a tree into a set of node edges i.e., \( f : T \to \mathbb{E} \) where \( f(T) = E_T \) is the set of all edges in \( T \). Evidently \( f \) is a surjective function since many structurally different trees can be mapped to the same edge set i.e., \( f \) causes many “collisions”. Hence, these functions are not effective in establishing the similarity among given trees. In contrast, our transformation functions capture interesting non-trivial structural relationships present among tree nodes. Furthermore, they can be made, when specialized with more information, to provide unique mapping from trees to multisets (see Section 3.2). Such unique transformations resemble perfect hash functions which are guaranteed to map distinct elements to distinct hash values. We present several alternative pivot structures in Section 3.1.3. The exact nature of the transformation primarily depends on the notion of tree structure that the underlying application wants to capture.

Due to the nature of our pivot substructures, all the properties that are established for sets can now be provided for trees also. In particular, the Jaccard coefficient that serves as a metric for (pivot) set similarity can now be used as a measure of similarity between the corresponding trees.

\[
\text{sim}(T_1, T_2) = \text{Jaccard}(S(T_1), S(T_1)) = \frac{|S(T_1) \cap S(T_2)|}{|S(T_1) \cup S(T_2)|} \tag{3.1.2}
\]

The intersection (union, respectively) is generalized from sets to multisets by taking
the minimum (maximum, resp.) of the two frequencies of a pivot in the multisets to be intersected (merged, resp.).

3.1.2 Signature Function (sf)

Merging and intersecting the pivot sets as in Equation 3.1.2 can be computationally very expensive because the pivot set sizes can be potentially quadratic in tree size. In a worst case, all pivot sets from a database may not fit in the main memory. The signature functions are designed to deal with these problems. A signature function converts large pivot sets into shorthand summaries or fingerprints so that all expensive operations are performed on these small signatures. The key idea here is to produce signatures such that the similarity between two signatures is roughly the same as the similarity between corresponding pivot sets.

There exist several ways to define a signature function. A fingerprint for a tree $T$ can be produced as a random sample drawn from the pivot multiset $\mathcal{S}(T)$. One may also consider a weighted random sample where the weight of a pivot is proportional to its multiplicity in $\mathcal{S}(T)$. We employ a signature function that is inspired from the popular MinHashing technique (short for Minwise Independent Permutation Hashing) [36, 35]. The similarity between minhash signatures is guaranteed to approximate the similarity between (multi)sets from which the signatures are generated [35]. We thus have,

$$\text{sim}(T_1, T_2) = \frac{|\mathcal{S}(T_1) \cap \mathcal{S}(T_2)|}{|\mathcal{S}(T_1) \cup \mathcal{S}(T_2)|} \approx \frac{|\text{sig}(T_1) \cap \text{sig}(T_2)|}{|\text{sig}(T_1) \cup \text{sig}(T_2)|} \quad (3.1.3)$$

Method: For a given tree $T$, the pivot multiset $\mathcal{S}(T)$ is first constructed as per Algorithm 5. The signature $\text{sig}(T)$ is then constructed via minhashing. To this purpose, each pivot $p=(\text{lca}, u, v) \in \mathcal{S}(T)$ is hashed to a numeric value as follows:

$$\text{ph}(p) = (a_1 \cdot \text{lca} + a_2 \cdot u + a_3 \cdot v) \bmod P \quad (3.1.4)$$

where $P$ is a large prime number, and $a_1, a_2, a_3 \in \mathbb{Z}_P$. Note that the node labels are used in the above multiplication. Alphanumeric labels are handled by using methods like Karp-Rabin algorithm [107]. This pivot hash function is sufficiently random and gives low probability of collision [85]. The pivot multiset can then be treated as a multiset of numbers less than $P$ where each number denotes a single pivot.

The basic idea in signature construction is to randomly permute the universe of pivots, and hash the pivot set under that permutation. For a given permutation $\pi_i$, 

58
Algorithm 6 Similarity between two trees $T_1$ and $T_2$

**Input:** $T_1$, $T_2$, $ph$, hash family $H = \{(a_i, b_i), 1 \leq i \leq k\}$, $M$

1. construct pivot sets $S(T_1), S(T_2)$ using Algorithm 5
2. $\text{sig}(T_1) \leftarrow \text{sketch}(S(T_1), ph, H)$
3. $\text{sig}(T_2) \leftarrow \text{sketch}(S(T_2), ph, H)$
4. compute Jaccard of $\text{sig}(T_1)$ and $\text{sig}(T_2)$

the index of the first pivot that belongs to set $S(T)$ is produced as the set’s minhash value $h_i(T)$. It has been shown that for a random permutation, the probability with which two sets produce the same hash value is equal to their Jaccard similarity [35, 60]. False positives from this probabilistic scheme are minimized by repeating the process $k$ times, resulting in $k$-minhashes.

$$\text{sig}(T) = \{h_i(T) = \min_{p \in S(T)} \pi_i(ph(p)), 1 \leq i \leq k\}$$

where $\pi_i$’s are random permutations over the universe of pivots. For a universe of large size, explicit construction of these permutations is very expensive. Broder et al. [35] showed that when the universe of elements is $\{0, 1, \cdots, P-1\}$ for some prime $P$, one can instead consider a family of permutations of the form:

$$\pi_i(x) = a_i \cdot x + b_i \pmod M$$

where $a_i \in \mathbb{Z}_M^*$, $b_i \in \mathbb{Z}_M$, and $M$ is a prime number that is not smaller than the universe size (i.e., $M \geq P$). They showed that the performance of such linear hash functions is as good as random permutations. Once the signatures are produced as per these linear hash functions, they can be used to estimate the tree similarity, as shown in Equation 3.1.3.

For each signature, along with the hash value we also store the corresponding pivot’s multiplicity in $h_i(\cdot)$. Therefore, the intersection (union, resp.) of signatures is computed using the multiset extension i.e. by considering the minimum (maximum, resp.) of the two multiplicities involved. The entire procedure to compute the similarity between two trees is summarized as Algorithm 6.

3.1.3 Different Transformation Functions

A key benefit in the above described approach is that the pivots (and hence the transformation functions) can be adjusted according to the type of structural relations
Algorithm 7 Tree Sketching (sketch)

Input: $S(T)$, $ph$, hash family $H = \{(a_i, b_i), 1 \leq i \leq k\}$, $M$

Output: $\sigma(T)$

1: for each $p \in S(T)$ do
2:   add $ph(p)$ to $S'(T)$ (Equation 3.1.4)
3: for $i = 1$ to $k$ do
4:   $min \leftarrow 0$
5:   for each $p \in S'(T)$ do
6:     $hash \leftarrow a_i \cdot p + b_i \pmod{M}$
7:     if $min > hash$ then
8:       $min \leftarrow hash$
9:   add $min$ to $\sigma(T)$

one wants to consider while comparing given tree structures. We now exhibit this flexibility in our approach by describing some alternative ways to define a pivot substructure. This list by no means is an exhaustive one.

- **Unordered pivots ($tf^u$)**: The pivots discussed thus far in this chapter are unordered embedded pivots where there is no specific order defined on the two children nodes.

- **Ordered pivots ($tf^o$)**: Several applications including the ones in computational linguistics, bioinformatics, and document-centric XML require the data to be modeled as ordered trees. To cater to such applications, an ordered pivot substructure can be defined where the two children nodes of the pivot are ordered as per the application’s requirements. The transformation function that produces such pivots operates in a similar manner to Algorithm 5 except for the swap operation in Lines 5–6. Note that when the labels are not swapped, different orientations of an unordered tree produce different pivot sets.

- **Constrained pivots ($tf_c$)**: The transformation functions described above produce multisets containing all possible pivots. They present the global structure of the tree since each node is paired up with every other node, whenever possible. Some applications may however require only the local structure. The transformation functions can be adapted to such scenarios by providing additional constraints representing the type of local substructure one wants to focus on. There may exist two types of constraints: node constraints and edge constraints. Node specific constraints allow the user to specify a set of nodes of interest $U = \{u_1, u_2, \cdots\}$ in the tree. The transformation function then produces only those pivots $(lca, u, v)$ where $lca, u, v \in U$. Such
node specific constraints can easily be pushed into the construction process (Lines 2, 4 in Algorithm 5). In Section 3.3.2, we present a specific case study relating to Phylogenetics, where the trees denote the evolutionary relationships among known organisms, which are typically present at leaf nodes (i.e., $U$).

Local tree structure can also be described by specifying the fixed neighborhood around each tree node. In other words, only those pivots where $lca$ is within a certain number of hops from two children nodes $u$ and $v$ i.e., $|d_{lca} - d_u| \leq \theta$ and $|d_{lca} - d_v| \leq \theta$, for a user defined parameter $\theta$. Such constraints can also be incorporated into the construction process (at Line 4 in Algorithm 5). Several applications require such localized structures. Consider a graphics or a vision application [161] where the images are represented using some space partitioning data structure like kd-tree. Level-constrained pivots in this case correspond to specific localized regions on the image. Such localization may be useful, for example, in pruning the search space while comparing multiple images.

- **Induced pivots ($tf_i$):** In contrast to embedded pivots which preserve ancestor-descendant relationships, induced pivots maintain only parent-child relations among nodes. In an induced pivot $(lca, u, v)$, $lca$ is the parent of both $u$ and $v$. Note that, this transformation is a special case of level-constrained pivot transformation $tf_e$ where $\theta = 1$.

- **Embedded pivots with levels ($tf_e^l$):** This transformation produces pivots that are annotated with the node depth information. Such annotation can be done in two ways: each node in the pivot is attached with its depth; or each edge is attached with the level difference in corresponding nodes. In the former case, each pivot is a 6-tuple $(lca, l_1, u, l_2, v, l_3)$ where $l_1 = d_{lca}$, $l_2 = d_u$, and $l_3 = d_v$. In the latter case where the edges are annotated with level difference, each pivot is defined as a 5-tuple $(lca, l_1, u, l_2, v)$, $l_1 = d_u - d_{lca}$ and $l_2 = d_v - d_{lca}$. Notably, these pivots embody more structural information than the pivots described earlier. In fact, this particular transformation function provides a unique mapping between trees and pivot sets. In other words, distinct trees are mapped to distinct pivot sets - see Section 3.2.3 for more details.

### 3.2 Theoretical Analysis

We investigate the theoretical properties of our transformation operators and describe efficient mechanisms to compute them. We first introduce some useful notation. Let
$r(T)$ and $d(T)$ be the root node and the height of a tree $T$, respectively (see Table 3.1). Similarly, let $d_u$ be the depth of a node $u \in T$. The node depth information can be used to characterize a particular orientation of an unordered tree $T$ by defining its level code $d_1d_2 \cdots d_n$ where node $i$ in preorder appears on level $d_i$. Among all possible orientations of $T$, the one with lexicographically largest level code is chosen as the canonical orientation of $T$, and the corresponding code is referred to as the canonical level code of $T$.

### 3.2.1 Bounds on Node Level Pivots

Let $N^T(v) = |S^T(v)|$ denote the number of pivots in which a particular node $v$ is present. As discussed in Section 3.1.1, $S^T(v)$ can be divided into two subsets: $S^T_{\text{root}}(v)$ in which $v$ is the root; and $S^T_{\text{child}}(v)$ in which $v$ is one of the two children. While $S^T_{\text{child}}(v)$ contains the pivots in which $v$ is paired with every other node, except for its ancestors $A(v)$ and descendants $D(v)$, the set $S^T_{\text{root}}(v)$ depends only on $v$’s descendants. We thus have,

$$
N^T_{\text{child}}(v) = |S^T_{\text{child}}(v)| = n - 1 - |A(v)| - |D(v)|
$$

$$
N^T_{\text{root}}(v) = |S^T_{\text{root}}(v)| \leq \frac{|D(v)| \cdot (|D(v)| - 1)}{2} \quad \text{(from Eq 3.1.1)}
$$

Since $N^T(v) = N^T_{\text{child}}(v) + N^T_{\text{root}}(v)$, we have

$$
N^T(v) \leq [n - 1 - |A(v)| - |D(v)|] + \frac{|D(v)| \cdot (|D(v)| - 1)}{2}
$$

$$
\leq n - 1 - |A(v)| + \frac{|D(v)| \cdot (|D(v)| - 3)}{2} \quad \text{(3.2.1)}
$$

### 3.2.2 Incremental Construction

![Figure 3.2: Incremental computation of $N(T)$ using Property 3.2.1](image-url)

Figure 3.2: Incremental computation of $N(T)$ using Property 3.2.1
In many XML applications, the document order that exists among nodes within a single document is very important. Operations like XML canonicalization [198] process the nodes in ascending document order. Such an order is also important in domains like XSLT. When the nodes are considered in document order, there exists an elegant manner in which pivot sets can be computed. It relies on the following observation.

**Property 3.2.1.** Let $T$ be a tree with associated pivot set $S(T)$. If a new leaf node $u$ is attached to a node $v \in T$, resulting in a new tree $T'$ then $S(T') = S(T) + S^{T'}(u)$.

It states that the existing pivots are not altered when new leaf nodes are added to the tree. This is because in a given pivot $(lca, u, v)$, no addition of a leaf node can alter the ancestor-descendant relations between $lca$ and $u, v$. It enables us to decompose a pivot set into multiple disjoint sets. For example, if the root $r = r(T)$ has $k$ children $u_1, \cdots, u_k$ (in that document order) then $S(T)$ can be written as $S^T(r) + \cup_{i=1}^k S(T(u_i))$.

This property also allows us to construct $S(T)$ in an incremental manner when the nodes in $T$ are considered in document order. In other words, pivots can be constructed as and when the new nodes are encountered while scanning the document. If $u$ is the next node in document order then the new set of pivots introduced by $u$ can be simply constructed by pairing $u$ with other existing nodes in $T$, except for its ancestors. The number of such pivots is exactly equal to the difference between the number of nodes which appear before $u$ in document order (i.e., $pre(u) - 1$) and the depth of $u$ ($d_u$) – see Figure 3.2. The size of $S(T)$ can thus be computed in linear time by simply scanning $T$ exactly once (in pre-order), without actually computing the set itself. Such fast size estimation techniques are helpful in reducing the search space in applications involving node-by-node processing. Note that this method of counting helps only when new nodes are added as leaf nodes but not when they are inserted at arbitrary positions within the tree.

### 3.2.3 Relation to Perfect Hashing

Perfect hash functions map distinct keys to distinct values. In case of tree-structured data, such functions map distinct trees to distinct pivot sets. Such a property is primarily governed by the structure of pivots. Not all transformation functions can guarantee this property. For instance, in the context of functions like $tf_i^u$ and $tf_i$, it is easy to construct trees that are structurally different but map to identical pivot sets – see Figure 3.3. However, for a specific transformation $tf_i^u$, where the pivots are
Figure 3.3: Counter examples for Perfect Hashing (see Section 3.2.3): (a) $tf_u$, (b) $tf_i$ & $pq$-grams [22]

annotated with level information, we prove that the mapping is perfect. This result is formally stated as the following theorem.

**Theorem 3.2.1.** The function $tf^d_0 : T \rightarrow S$ is injective i.e., for given $T_1$ and $T_2$, if $T_1 \neq T_2$ then $S(T_1) \neq S(T_2)$.

We first prove this theorem in the context of unordered, unlabeled trees, and subsequently extend this result for trees with node labels.

**Unlabeled Trees**

Before we delve into the details of the proof, we introduce two useful concepts: tree merge; and structural equivalence between nodes within a single tree.

**Definition: 3.2.1.** Tree Merge: Consider two trees $T_1$ and $T_2$ with canonical level codes $a_1 \cdots a_m$ and $b_1 \cdots b_n$, respectively. Say, $\forall i < k ~ a_i = b_i$, and $a_k > b_k$ i.e., $k$ is the first location at which both codes differ. We define a merged tree $T_1 \otimes T_2$ to be a tree with the following level code:

$$Code(T_1 \otimes T_2) = \underbrace{Code(T_1) \otimes Code(T_2)}_{\left[ a_1 \cdots a_{k-1} a_k \cdots a_m \right]}$$

$$\quad = \underbrace{\otimes}_{b_1 \cdots b_{k-1} b_k \cdots b_n}$$

$$\quad = a_1 \cdots a_{k-1} a_k [a_{k+1} \cdots a_m \triangleright b_k \cdots b_n]$$

where $[a_{k+1} \cdots a_m \triangleright b_k \cdots b_n]$ is some valid combination of level codes. The tree merge operation essentially creates a larger tree with $m + n - k + 1$ nodes such that the first $k$ nodes are similar to $T_1$, and rest of the nodes from $T_1$ and $T_2$ are added in some valid combination. A simple case of a tree merge operation is illustrated in
Figure 3.4. In this particular example, the valid combination is obtained through simple block movement of level codes.

**Definition: 3.2.2.** Structural Equivalence: Two nodes $u$ and $v$ in a tree $T$ are said to be structurally indistinguishable (or structurally equivalent) if and only if the following two conditions hold:

- **i)** Subtrees $T(u)$ and $T(v)$ possess the same structure.
- **ii)** Either $u$ and $v$ are siblings or their parent nodes are structurally indistinguishable.

It essentially states that $u$ and $v$ are placed in $T$ in such a way that they can be swapped without affecting the tree structure. By permuting such structurally indistinguishable nodes one can enumerate different automorphisms of $T$. Note that a tree automorphism is an isomorphism from a tree to itself. The recursion in the definition is carried out until the lowest common ancestor of the two nodes is encountered, at which point two siblings are encountered. From the definition, it is easy to see that $d_u = d_v$. A pictorial demonstration of the definition is shown in Figure 3.5. Note that, if $u$ and $v$ (whose LCA is $w$) are structurally equivalent then note only that $T(u)$ and $T(v)$ are identical but the branches of $T(w)$ which contain $u$ and $v$ are also equivalent.

**Theorem 3.2.2.** Two tree nodes $u$ and $v$ in $T$ are structurally indistinguishable if and only if $S^T(u) = S^T(v)$.  

65
Proof. This result stems from the fact that the embedded pivots involving a given node capture the entire tree structure when seen from that node. Also, the pivots now are specialized by annotating edges with level difference.

**Only If part:** Recall that the pivots \( S^T(u) \) involving a single node \( u \) can be partitioned into two subsets \( S^T_{\text{root}}(u) \) and \( S^T_{\text{child}}(u) \). The first condition in Definition 3.2.2 implies that \( S^T_{\text{root}}(u) = S^T_{\text{root}}(v) \). All the remaining pivots in \( S^T_{\text{child}}(u) \) are of the form \((-,-,u,-,x)\) where \( x \) is some node that is neither an ancestor nor a descendant of both \( u \) and \( v \). We then have two cases:

- **\( x \notin T(w) \):** In this case, \( x \) is a node that is present in \( T' \) (see Figure 3.5. Since the node label and the depth are same for both \( u \) and \( v \) due to their structural equivalence, all pivots of the form \((-,-,u,-,x)\) and the pivots \((-,-,v,-,x)\) are equivalent.

- **\( x \in T(w) \):** Here, \( x \) can either be part of \( T'' \) or is a node that is on the path \( w \sim u \) or \( w \sim v \) (i.e., an ancestor node to either \( u \) or \( v \)). If \( x \in T'' \) then one can make the same argument as in previous case (\( x \notin T(w) \)). If \( x \notin T'' \) and \( x \) is on the path \( w \sim u \) then there must exists \( x' \) that is on the path \( w \sim v \) such that \( x \) and \( x' \) are structurally equivalent – second condition in Def. 3.2.2. In such a case, the pivot \((y,-,x,-,v)\) that belongs to \( S^T_{\text{child}}(v) \) will have an equivalent pivot \((y',-,x',-,u)\) in \( S^T_{\text{child}}(u) \) in such a way that \( y \) and \( y' \) are structurally equivalent. We can equivalently argue the case where \( x \) is on the path \( w \sim v \).

We thus proved that \( S^T_{\text{child}}(u) = S^T_{\text{child}}(v) \) if \( u \) and \( v \) are structurally equivalent.

**If part:** We show that this part of the theorem holds by proving its contrapositive i.e., if \( u \) and \( v \) are not structurally equivalent then \( S^T(u) \neq S^T(v) \). If they are not equivalent then one of the two conditions in Definition 3.2.2 must fail.
• $T(u) \neq T(v)$: Let the canonical level codes of $T(u)$ and $T(v)$ be $c_1 \cdots c_m$ and $d_1 \cdots d_n$, respectively. If the subtrees differ in their structure then there must exist $x \in T(u)$ and $x' \in T(v)$ such that $d(x)=c_k \neq d_k=d(x')$ for some $k \leq \min(m,n)$, where $k$ is the position at which both level codes differ. Now consider the specific pivots $p_1$ formed by $u$ and the node in $T(v)$ i.e., $(w, -, u, d_k, x') \in S^T_{\text{child}}(u)$ and similarly $p_2 = (w, -, v, c_k, x) \in S^T_{\text{child}}(v)$. Since $c_k \neq d_k$, we have $p_1 \neq p_2$.

Now consider an instance where the second condition of Definition 3.5 fails. Let the parent nodes of $u$ and $v$ are $z$ and $z'$, respectively. If $z$ and $z'$ are not structurally equivalent then, by a similar argument from previous case, we can find at least one node in each of $T(z)$ and $T(z')$ such that the nodes differ in their depth. By pairing them with $u$ and $v$, as we did in previous case, we can find at least one pivot that is in $S^T_{\text{child}}(u)$ but not in $S^T_{\text{child}}(v)$.

We thus proved that nodes $u$ and $v$ are structurally equivalent if and only if $S^T(u) = S^T(v)$.

We use the above result to prove Theorem 3.2.1 – the function $\text{tf}_e^1$ is injective.

**Proof of Theorem 3.2.1** Consider two cases: $d(T_1) \neq d(T_2)$; and $d(T_1) = d(T_2)$. In the former case, without loss of generality assume that $d(T_1) > d(T_2)$. Since all trees are branched, there must exist at least one pivot of the form $(r(T_1), d(T_1), -, -, -)$ that is in $\mathcal{S}(T_1)$ but not in $\mathcal{S}(T_2)$. Thus, $\mathcal{S}(T_1) \neq \mathcal{S}(T_2)$.

Now consider the case where $d(T_1) = d(T_2) = d$. We prove this case by contradiction i.e., we assume $\mathcal{S}(T_1) = \mathcal{S}(T_2)$ and argue that it is not possible. Let $\text{Code}1$ and $\text{Code}2$ are canonical codes for $T_1$ and $T_2$. Assume that $\text{Code}1 > \text{Code}2$ and $k$ be the smallest index at which both the codes differ. Since $d(T_1) = d(T_2)$, we have $k > d$. This instance is summarized below:

\[
\begin{align*}
\text{Code}(T_1) &= \text{Code}1 = a_1 \cdots a_{k-1} a_k \cdots a_m \\
\text{Code}(T_2) &= \text{Code}2 = b_1 \cdots b_{k-1} b_k \cdots b_n \\
\text{WLOG : } \text{Code}1 &> \text{Code}2 \text{ i.e., } \forall i < k, a_i = b_i; \quad a_k > b_k
\end{align*}
\]

Since the first $k-1$ nodes in both the trees are same and since we assumed $\mathcal{S}(T_1) = \mathcal{S}(T_2)$, we must have $k'$ such that:

\[
\exists k' \in T_2, k < k' \leq n \text{ such that } \mathcal{S}^{T_1}(k) = \mathcal{S}^{T_2}(k') \tag{3.2.2}
\]
Now consider the merged tree $T_3 = T_1 \otimes T_2$ as defined in Def. 3.2.1. Note that both $k$ and $k'$ belong to the merged tree. From Property 3.2.1, Equation 3.2.2 and from the way merged tree is constructed, we can derive that the pivot sets involving $k$ and $k'$ in the merged tree are same i.e., $S_{T_3}^T(k) = S_{T_3}^T(k')$. In other words, both $k$ and $k'$ are structurally indistinguishable in $T_3$ (from Theorem 3.2.2). However, such a node $k'$ can not exist in $T_2$ since we assumed that $Code2$ is canonical and $Code2 < Code1$. Therefore, our assumption $S(T_1) = S(T_2)$ can not hold true when $T_1 \neq T_2$. □

**Labeled Trees**

Definition 3.2.2 can easily be extended to define structural equivalence between $u$ and $v$ in a labeled tree by making two simple changes – the labels of $u$ and $v$ must be same; $T(u)$ and $T(v)$ in condition 1 must not only be structurally identical but the node labels also must match. Based on this modified definition, we provide the following theorem.

**Theorem 3.2.3.** Two nodes $u$ and $v$ in a labeled unordered tree $T$ are indistinguishable if and only if $S^T(u) = S^T(v)$.

**Intuition:** The proof relies on two facts. First, the structural equivalence between $u$ and $v$ in the unlabeled version of $T$ is a necessary but not sufficient condition for their equivalence in $T$. Second, node labels only specialize the pivots by making them more distinct. □

**Theorem 3.2.4.** If $T_1$ and $T_2$ are two labeled trees with different structure then $S(T_1) \neq S(T_2)$.

**Proof** From Theorem 3.2.2, the unlabeled versions of $S(T_1)$ and $S(T_2)$ are not the same. Since the equivalence between corresponding unlabeled pivot sets is a necessary condition, we can derive that $S(T_1) \neq S(T_2)$. □

**Theorem 3.2.5.** Consider two labeled trees $T_1$ and $T_2$ with exact same structure. If $T_1 \neq T_2$ i.e. they differ in node labels then $S(T_1) \neq S(T_2)$.

**Intuition:** Suppose that all structurally equivalent nodes of $T_1$ are clustered into groups $G_1 \cdots G_k$. Permuting the nodes within each group results in different automorphisms of $T_1$ without altering the pivot set. We can then argue that $T_1$ and $T_2$ will have same pivot sets if and only if they have identical group structure i.e., they are automorphisms of each other. □
3.2.4 Relation to Tree Edit Distance

The most commonly used distance measure on tree structured data is the tree edit distance [31, 214], which denote the minimum number of basic edit operations (relabel, insert, and delete) to transform one tree into the other (see Figure 3.6). Some researchers have used a variant of this basic measure that includes subtree moves, which allow a subtree to be moved under a new node in the tree in one step [83]. In this section, we briefly present the relation between this variant of tree edit distance with subtree moves with our tree similarity measure from Section 3.1.1. Later in Section 3.3, we empirically evaluate this relation on different datasets.

**Theorem 3.2.6.** Consider two trees $T_1$ and $T_2$. If $S(T_1)=S(T_2)$ then the maximum edit distance between them is equal to $m + n - 6$, where $m = |T_1|$ and $n = |T_2|$.

**Proof** Since we are interested in the maximum edit distance, we choose two extreme cases discussed in Section 3.1.1 for $T_1$ and $T_2$. Let $T_1$ and $T_2$ be the tree structures similar to the ones shown in Figure 3.1a & c, respectively. We thus have, $\mathcal{N}(T_1) = \frac{(n-1) \cdot (n-2)}{2}$ and $\mathcal{N}(T_2) = m - 2$.

If $S(T_1) = S(T_2)$ then $\frac{(n-1) \cdot (n-2)}{2}$ must be equal to $m - 2$ ($m > n$). Note that the choice of $T_2$ is very important to find the maximum edit distance. We can not choose any other tree structure with more than $m$ nodes for $T_2$ – there can not
exist a larger tree that has the same number of pivots as that of $T_1$. This is because $T_2$ is a $m$-node tree structure with minimum number of pivots. Also, if we consider trees with size less than $m$ then we may not get the maximum distance. Therefore, the choice of $T_2$ is justified to find the maximum edit distance.

Now, consider the edit distance between $T_1$ and $T_2$. To convert $T_1$ in Fig. 3.1a into $T_2$ in Fig. 3.1c, we need $(n - 3)$ deletions and $(n - 3) + (m - n)$ additions. This implies that the edit distance between trees with same pivot sets can not exceed $m + n - 6$.

For each of the edit operations, we construct lower bounds on the number changes in the pivot set, especially in $S(T_1) \cap S(T_2)$. Note that these are not the bounds for our Jaccard similarity. These bounds provide an intuition as to how pivot sets change when different edit operations are applied on a tree.

**Relabel $v_e$**

When a particular node label is changed then only those pivots which contain the modified node are altered. They include pivots both $S_{\text{child}}^{T_1}(v_e)$ and $S_{\text{root}}^{T_1}(v_e)$, whose number is equal to $N^{T_1}(v)$ as given in Section 3.2.1. We can then construct a lower bound on the pivot multiset intersection (the numerator in Jaccard similarity) as follows:

$$
S(T_1) \cap S(T_2) = S(T_1) - S_{\text{root}}^{T_1}(v_e) - S_{\text{child}}^{T_1}(v_e)
$$

$$
|S(T_1) \cap S(T_2)| = N(T_1) - N^{T_1}(v_e)
$$

$$
\geq N(T_1) - [n - 1 - |A(v_e)| + \frac{|D(v_e)| \cdot (|D(v_e)| - 3)}{2}] \quad (3.2.3)
$$

$$
\geq N(T_1) - \frac{|D(v_e)| \cdot (|D(v_e)| - 3)}{2} - n
$$

**Delete $v_e$**

When $v_e$ is deleted, the children of $v_e$ are attached to the parent of $v_e$ (say, $p_{v_e}$) in the resulting tree. Suppose $\overline{T_1(v_e)}$ be the tree that does not contain $v_e$ and its descendants i.e., $\overline{T_1(v_e)} = T_1 - T_1(v_e)$. Note that the pivots that belong to $\overline{T_1(v_e)}$ are not affected by the deletion of $v_e$. Therefore one needs to analyze on the pivots involving nodes from $T_1(v_e)$. Let us first consider $v_e$ itself. All the pivots in which $v_e$ is the child are eliminated, and hence they do not belong to new pivot set i.e., $S_{\text{child}}^{T_1}(v_e) \cap S(T_2) \neq \phi$. For every pivot in $S_{\text{root}}^{T_1}(v_e)$, the root node is modified to $p_{v_e}$.
Now consider other nodes \( x \in D(v_e) \). All the pivots of the form \((z, -, x, -, y)\) where \( y, z \in D(v_e) \) are not affected by the deletion. Only those pivots in which \( x \) is paired up with nodes \( y \in T_1(v_e) - A(v_e) \) are affected. More precisely, a pivot \((-d_x, x, -, y)\) will be changed to \((-d_x - 1, x, -, y)\) in \( T_2 \) — the depth of \( x \) is reduced by 1 due to the deletion of \( v_e \). Let this set be denoted as \( S' \), whose exact size depends on the number of ancestors and descendants that \( v_e \) has in \( T_1 \).

\[
S' = \{(-, -, x, -, y)| x \in D(v_e) \land y \in T_1(v_e) - A(v_e)\}
\]

\[
S(T_1) \cap S(T_2) = S(T_1) - S^{T_1}(v_e) - S'
\]

For convenience, let \( A = |A(v_e)| \) and \( D = |D(v_e)| \). We can then bound the intersection size as follows.

\[
|S'| = ( n - |T_1(v_e)| - A ) \cdot D
\]

\[
|S(T_1) \cap S(T_2)| = N(T_1) - N^{T_1}(v_e) - |S'| \geq N(T_1) - (n - 1 - A - \frac{D \cdot (D - 3)}{2}) - |S'|\]

\[
\geq N(T_1) - (n - A) \cdot (1 - D) - 1 - \frac{D \cdot (D + 5)}{2}
\]

For example, if \( v_e \) is a leaf node then the above expression reduces to \( N(T_1) - (n - A - 1) \), where \( (n - A - 1) \) is exactly equal to the number of pivots that \( v_e \) has in \( T_1 \) i.e. \( |S_{child}(v_e)| \).

**Insert** \( v_e \)

Insertion is a dual to deletion that is discussed above. Therefore, the pivots which get eliminated during deletion, for example \( S^{T_1}_{child}(v_e) \), are introduced into \( T_2 \) due to insertion. Similarly the pivots which get introduced while deletion are eliminated during insertion. Hence, the change in the number of pivots is same as the bounds shown in previous section except that \( A \) and \( D \) now refer to the number of ancestors and descendants of \( v_e \) in the modified tree \( T_2 \), respectively.

**Move** \( v_e \) to \( v_f \)

Now consider the last edit operation, move. Here, a non-root node along with the subtree rooted at node is moved from one tree node to the other. It is depicted in Figure 3.6b. \( T(v_e) \) that is under the node \( w_1 \) in \( T_1 \) is moved under \( w_2 \), resulting in
$T_2$. In such a case, the set $S_{\text{child}}^{T_i}(v_e)$ changes completely, especially when the depth of $v_e$ in both the trees is different. However, the pivots $S_{\text{root}}^{T_i}(v_e)$ are unaffected by the move operation. Similar to delete operation, the pivots involving descendants of $v_e$ and rest of the tree nodes change from $T_1$ to $T_2$. They are of the form $(-, -, x, -, y)$ where $x \in T(v_e)$ and $y \in T_1 - T_1(v_e)$. Therefore, the bound derived for this operation will be similar to the one for delete operation except that the set $S_{\text{root}}^{T_i}(v_e)$ that is eliminated during deletion is unaffected during move operation.

### 3.3 Empirical Evaluation

In our evaluation, we consider several publicly available tree-structured datasets drawn from a range of real-world applications including bioinformatics (Swissprot [4]), Linguistics (Treebank), Web log analysis (CSlogs [207]) and XMark [65] (online auctions). Among these, XMark is a synthetic dataset that models the behavior of an online auction site and is useful for controlled experiments. The size of XMark data trees is driven by the scaling factor, an input to the generator. It produces one large single tree. We remove the top-level XML tags such as `<site>`, `<regions>`, `<closed_auctions>` to generate a number of small trees.

All the results presented in this section are obtained by using signatures with 16 minhashes. Note that there is a tradeoff involving the signature size, the computation cost, and the accuracy. Larger the signature, the higher is the computation cost and higher is the accuracy. In our evaluation, signatures with 16 hash provided a nice balance among these factors. We primarily evaluate our transformation functions which are input to minhashing – the effectiveness of minhashing scheme has already been well studied in the literature [35, 60, 36].

**Basic Construction Costs:** We present the overall time spent in constructing signatures (including the cost of transformation and signature sketching operators) for different datasets in Figure 3.2. The cost associated with an induced pivot function $tf_i$ is linear with the tree size whereas the cost associated with $tf_e$ is quadratic in tree size. Note that $tf_e$ is used to denote a general transformation that produces embedded pivots, where the order and the level information on pivots is irrelevant.

In a controlled experiment, as the size of a XMark tree is changed from 1,000 to 10,000 to 30,000 nodes, the construction cost for $tf_e$ increased from 0.02sec to 10sec to 112sec – see Figure 3.7a. The construction cost grows linearly with the database size – see Figure 3.7b. Furthermore, the signature construction is an embarrassingly
parallel process and one can leverage modern multicore architectures in speeding up the process. For example, $\text{tf}_e$ on Swissprot on a dual quad core system has taken about 67.3 seconds – a 7.98-fold improvement over a sequential version with 535 seconds (see Figure 3.2).

**Information Content:** If we treat a pivot set as a message that describes the associated tree then the amount of information contained in the message (i.e., pivot set) can be quantified as the *information entropy* ($\sum p \log p$) of the set. We can use this measure to compare multiple pivot set strategies. Consider an XMark tree $T$ whose labels have been removed. We assign new labels to $T$ chosen from a set $L$ using the following scheme. We first randomly select a special label $\lambda \in L$. We then assign new labels to tree nodes by biasing the distribution towards the special symbol $\lambda$ with probability $b$. The remaining probability $1 - b$ is equally divided among other labels i.e. each label other than $\lambda$ is chosen with uniform probability $\frac{1 - b}{|L| - 1}$. Figure 3.7c depicts the change in entropy of $S(T)$ as the label bias $b$ is varied – $T$ is a XMark tree with 31,000. The main trends, along expected lines are: i) that the information content of embedded pivots with labels dominate the other two strategies across the board; ii) even when the label bias is 1.0 (all nodes have the same label) the entropy of this strategy is not zero suggesting there is still important information in the levels to help disambiguate amongst trees; and iii) for low label bias values the difference between embedded with levels and embedded without levels becomes insignificant.

We have observed similar results for a range of trees of different sizes and shapes.

Figure 3.7: (a & b) Construction costs for $\text{tf}_e$; (c) Information content in pivot multisets
<table>
<thead>
<tr>
<th>Data set</th>
<th># of trees</th>
<th>Tree size</th>
<th>Tree depth</th>
<th>Sketching time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>max</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>Swissprot</td>
<td>50,000</td>
<td>7018</td>
<td>271.5</td>
<td>5</td>
</tr>
<tr>
<td>Dblp</td>
<td>328,858</td>
<td>4477</td>
<td>32.1</td>
<td>6</td>
</tr>
<tr>
<td>Treebank</td>
<td>52,851</td>
<td>648</td>
<td>68.0</td>
<td>35</td>
</tr>
<tr>
<td>Cslogs</td>
<td>59,691</td>
<td>428</td>
<td>12.9</td>
<td>85</td>
</tr>
<tr>
<td>XMark</td>
<td>280,000</td>
<td>28,000</td>
<td>59.7</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.2: Datasets and their characteristics

Figure 3.8: Approximating Edit Distance
Edit Distance

In this experiment, we empirically relate the edit distance with four operations (relabel, insert, delete, and subtree move) and the Jaccard distance \( (1 - \text{similarity}) \) obtained via our signatures. Unfortunately the computation of exact edit distance with moves is \( NP \)-Hard [164]. Also, to the best of our knowledge, there are no efficient algorithms which can approximate such edit distance. We therefore adopt a mechanism where a given tree \( T \) is subjected to a series of random perturbations. As we perform these perturbations, we monitor the change in Jaccard similarity between the original tree \( T \) and the perturbed tree \( T_p \).

For the purposes of this experiment, we select a tree at random from each of our datasets. For each tree, we prepared an edit script \( ES \), which is a sequence of edit operations that are denoted as \((\text{node}, \text{operation})\) pairs [83]. In order to avoid any redundant operations, we make sure that no node is deleted twice, no relabeled node is selected for relabel operation again, and no subtree is moved twice. Such a valid edit script is applied on the selected tree resulting in a new perturbed \( T_p \). We then compare the edit distance \(|ES|\) with our signature-based distance, \(1 - \text{Jaccard}(T,T_p)\) (see Section 3.1.1). Note that these experiments are representative and similar results were obtained for other trees and other random perturbations.

We compared the performance of the three basic strategies we propose, embedded with levels, embedded without levels and induced with two alternative approaches from the literature. The first strategy is based on \( pq \)-grams proposed by Augsten et al. [22, 21]. We have obtained the source code from the authors. Since our data trees are unordered, we have computed windowed \( pq \)-grams using their 3-step process [21] – sort the unordered tree, extend the tree with dummy nodes, and compute the windowed \( pq \)-gram profile. The results shown in Figure 3.8 are obtained for the default setting of \( p = q = 2 \) and \( w = 3 \). We found this parametric setting to work the best, in a manner similar to those reported by the authors[21]. In addition to \( pq \)-grams we also examined the performance of a path-based strategy denoted \( \text{paths} \). Here, the transformation function maps the tree into a set of root-to-leaf paths, which are then hashed using standard string hashing methods such as PJW hash [9]. The resulting set of hash values is then used for computing minhash signatures.

Figure 3.8 reports on the performance of these five transformation strategies for trees from three different datasets. The trend-line labeled “linear” refers to a hypothetical technique that can compute the exact edit distance, hence a linear relation. The closer we are to this line, the better is the approximation of the algorithm.
For all data sets and all perturbation scripts we have evaluated, the transformation function $\text{tf}_\text{u}$ using embedded pivots without levels dominates, and always provides the best approximation across the board. Similarly the weakest strategy is almost always the path based strategy. In the experiments we have observed the second best strategy is usually a toss up between embedded pivots with levels and the strategy based on $pq$-grams. As noted earlier, $\text{tf}_\text{e}$ behaves like a perfect hash function. Therefore, even a small change in the tree is likely to be exaggerated in the corresponding pivot set leading to a loss in accuracy in the estimation of edit distance. Since $pq$-grams are induced substructures (with some additional structural information) they can only capture the local structure around nodes, they are unable to approximate the actual edit distance as well as the embedded strategy without levels. We should also point out that in terms of execution time, the path-based and the basic induced strategies were inexpensive. The embedded strategies were more expensive but were not far off. Computing the $pq$-gram profiles was found to be the most expensive.

3.3.1 Case Study I: De-duplication of XML Documents

For our first case study, we consider the application of detecting duplicate documents in an XML repository [91, 190]. This problem is related to finding different representations (i.e. duplicates) of a same underlying object.

We perform a detailed evaluation by adopting an approach that is commonly used in evaluating deduplication techniques [190]. We add random noise or artificial duplicates to the data and then we observe how well our tree signatures can detect these duplicates. For this purpose, we implemented a tool that takes in a data set and generates a data set that is polluted with duplicates. Our tool takes two main parameters $p_t$ and $p_c$, where $p_t$ is the percentage of trees that are to be duplicated and $p_c$ is the amount of noise that is added while creating a duplicate. While $p_t$ is denoted as the percentage of the database size, $p_c$ is represented as the percentage of the number of nodes in a tree that is selected for duplication. For a given $p_t$ and $p_c$ values, the tool carefully generates duplicates by introducing various forms of errors: node deletions representing missing data; node modifications denoting typographical errors or corrupted data; node movements denoting copy-paste errors; and node insertions corresponding to extra noise.

We compute the signatures for each document in the resulting dataset containing both clean and dirty (or noisy) documents. By comparing the computed signatures,
Figure 3.9: Accuracy in XML Deduplication Study
we determine top $K$-nearest neighbors for each dirty (i.e., duplicate) document. We then check if the corresponding original document is one of the $K$ nearest neighbors.

Figures 3.9 show the results on Swissprot, Dblp, and XMark datasets for a variety of $p_c$ and $p_t$ values. The y-axis in these figures shows the percentage of duplicates that are detected (accuracy) by our algorithms. When $K$ is set to 1, we compare a duplicate document with its top nearest neighbor. For such a setup, the accuracy reduces as $p_c$ increases because the duplicate is no longer “similar” to the original document. For all three datasets, the accuracy is not affected by the number of trees that are altered ($p_t$). For a given $p_c$, the accuracy is roughly the same for all values of $p_t$. Notably, the accuracy obtained by $\text{tf}_e^t$ transformation is less than that of $\text{tf}_u^t$ that does not embody the level information. The performance difference between them is roughly the same for all three data sets. Also, note that the accuracy increases marginally with $K$, and it reaches its plateau after $K = 4$, for all datasets.

In this case study, induced pivots produced from $\text{tf}_i$ did not perform very well. For instance, when $p_t = 30\%$, $p_c = 5\%$, and $K = 1$, $\text{tf}_i$ detected only 28\% of duplicates in Swissprot and a mere 5\% of duplicates in Dblp. In contrast, embedded transformations $\text{tf}_e^u$ and $\text{tf}_e^t$ detected 75\% and 61\% on Swissprot, and 81\% and 75\% on Dblp, respectively. We did not consider $pq$-grams in this evaluation due to its poor run time behavior – a deduplication task on a small XMark dataset of 21,750 trees with 10\% duplicates has taken more than 24 hours\(^1\), especially due to expensive

\(^1\)On a Windows system with 2.4GHz Pentium 4 processor and 1.5GB RAM. The $pq$-gram profiles are stored as tables in MySQL database.
pq-gram table joins. However, we expect the effectiveness of pq-gram based approach to be similar to that of $\text{tf}_i$.

3.3.2 Case Study II: Phylogenetic analysis

For our final case study, a qualitative one, we consider the application domain Phylogenetics. Phylogenetic trees are typically unordered, and they help biologists in studying the evolutionary relationships among a given set of organisms (taxa). Leaf nodes denote the taxa whereas the internal nodes denote some ancestor organisms. Biologists often construct expensive consensus trees to extract common relations from a collection of phylogenies built over the same taxa. The process can be made efficient by dividing the set of all phylogenies into groups of structurally similar trees. We demonstrate the use of our constrained transformation $\text{tf}_e$ in order to perform such a grouping.

Methodology: We consider 12 aligned protein sequences from Chitinase (digestive enzyme) genes [64] in plants. We construct 20 different phylogenies by using different algorithms from the Phylip [182] toolbox.

Since the phylogenies are leaf labeled, we use $\text{tf}_e$ to construct node constrained pivot sets. The signatures computed from these pivot sets are used in grouping those 20 phylogenies into clusters. A domain expert manually examining these clusters, found that the trees in each cluster present very similar evolutionary relationships. Two such clusters (each with two trees) are shown in Figure 3.10. The 12 protein sequences are shown as $A, B, \cdots, L$ for simplicity. Evidently, the trees in each cluster preserve a significant number of common relationships among organisms.

For these particular phylogenies, even an induced transformation function $\text{tf}_i$ provided good clustering arrangements. However, in general, constrained embedded pivots are preferable for domain experts since the trees are leaf labeled. As Stockham et al. recently pointed out, the consensus trees constructed over groups of structurally similar phylogenies are more resolved than those that are produced using all phylogenies [170]. Furthermore, computation of consensus trees over these clusters is cheaper and effective than computing a single, global, often non-informative consensus tree suggesting that our hashing methods can be an effective preprocessing step for such methods.
3.4 Related Work

Tree similarity has been studied extensively in the context of tree editing distance [31, 214], which is a natural extension of string edit distance. Similar to the case of string comparisons, there exist a number of ways to compare different trees – largest common subtree [12], smallest common super-tree [158], tree alignment [104], to name a few. Exact computation of conventional tree edit distance is computationally expensive. The fastest known solution takes at least $O(n^3)$ time for ordered trees [31] and it is $\mathcal{NP}$-hard for unordered trees [214]. Even for ordered trees, the problem is $\mathcal{NP}$-hard when the “subtree move” operation is introduced [164]. There has been some efforts in developing approximate algorithms for tree edit distance [22, 21, 87, 83, 203].

Yang et al. match two ordered trees by using $L_1$ distance between corresponding vectors of binary branches. Binary branches are similar to induced pivots produced by our transformation $\mathbf{tf}_i$, and their performance is likely to be similar to $\mathbf{tf}_i$ as shown in Section 3.3. Augsten et al. measure the tree similarity using pq-grams, which are small subtrees of specific shape that is controlled by parameters $p$ and $q$ [22]. They recently extended their approach to unordered trees by considering different permutations of sibling nodes [21]. However, as we showed in Section 3.3, both the run time performance and the effectiveness of their approach is not as good as our algorithms. Furthermore, the parameters $p$ and $q$ must be tuned based on the data. A parameter setting that is suited for a tree in the database may not suit others. Guha et al. presents a framework for approximate XML joins based on edit distance between ordered trees [91]. Garofalakis and Kumar proposed XML stream processing algorithms for embedding tree edit distance into $L_1$ space while providing some bounds on the distortion [83]. However unlike our methods, their algorithms focus only on ordered trees. In general, approximating edit distance is a hard problem. Andoni and Krauthgamer have recently proved that the computational hardness of estimating string edit distance itself is very high [17] – for tree structures, the it is likely to be even higher. Recently, Gollapudi and Panigrahy proposed sketching techniques for hierarchical data, which are subsequently leveraged in providing LSH methods under Earth Mover’s Distance measure [87]. These methods, however, are developed only for leaf labeled trees. It is not easy to extend them for general node labeled trees.
3.5 Conclusions

We have presented a simple and effective framework for hashing tree structured data. The synopsis of the proposed approach entailed transforming the tree-structured datum into a multi-set of pivot structures and subsequently relying on min-wise hashing to yield a fixed length datum suitable for standard operations like nearest neighbor search, clustering etc. We examined the performance of induced, embedded without levels, embedded with levels, and constrained pivots from a theoretical perspective and proved that one of them can form the basis for constructing a perfect hash function and also proved lower bounds connecting these strategies with tree edit distance operations. To further enhance the efficacy of the proposed transformations we realized parallel implementations on modern multicore systems with linear time speedups. We demonstrated the utility of the hashing framework on several applications and case studies drawn from the domains of XML deduplication, and phylogenetic data analysis. Another case study on frequent pattern mining is shown later in Chapter 5.
CHAPTER 4
MINING TREE-STRUCTURED DATA

Frequent pattern mining is a fundamental task in the knowledge discovery process that deals with mining useful and common patterns from massive data sets. While most of the early work in this context has focused on mining simple transactional datasets, recently there is a significant shift towards analyzing data with complex structure such as trees and graphs. This problem is often popularly referred to as frequent itemset mining while dealing with transactional data sets. An equivalent problem for tree-structured data is often cited as frequent subtree mining.

Recently frequent pattern mining has gained a lot of interest in a number of application domains that produce large amounts of semi-structured data. For example in bioinformatics, the secondary structure of a RNA molecule is represented as a rooted ordered tree \[213\]. Common substructures discovered from a database of such trees helps in discovering new functional relationships among corresponding RNAs \[80\]. These substructures are known to be useful in predicting RNA folding \[119\] and in functional studies of RNA processing mechanisms \[165\]. Similar techniques can be extended to other biological molecules such as glycans whose basic structure is denoted as trees. Glycans are covalent assemblies of sugar that play crucial roles in many cellular processes \[199\]. Significant subtree patterns obtained from a group of such trees help biologists in classifying glycan molecules \[93\], and hence in deeper understanding of their functionality.

In Phylogenetics, a branch of biology, scientists study evolutionary relationships among various groups of organisms (or taxa), which are discovered through molecular sequencing data and morphological data matrices. These relationships are naturally denoted as a branching structures known as phylogenetic or evolutionary trees. Due to uncertainties in the target phylogenetic tree structure, biologists often induce multiple trees for the same taxa by executing several different algorithms. Subsequently, they search for most stable and meaningful relationships that are present across different
trees – a computational problem that can be solved via frequent subtree mining techniques [166, 208].

Consider the case of web usage mining. A number of visitors (often in thousands) maneuver through the popular web-sites like Google, Amazon, and Yahoo!. The pages in these web-sites are usually arranged in an hierarchical fashion i.e., in a tree structure. The web page access histories of individual visitors are also modeled (with some approximations) as trees [208], typically using the information retrieved from web server log files which contain session-level data. Common browsing patterns extracted from such a collection of trees are shown to be useful in many applications: in making recommendations; in personalizing the web as per the end user needs; in better organizing the web-site structure; and also in business intelligence [73, 208]. Similarly, consider the web pages (e.g., wiki pages) whose content is well organized (e.g., sections and subsections). Mining the implicit (DOM) structure can be used in clustering and classification of documents [116].

Linguists can leverage pattern mining algorithms in developing and training of automatic probabilistic language parsers over a large corpus of sentences. The syntactic structure of sentences is often modeled using tree structures known as *treebanks*. Treebanks are created either completely manually, where linguists annotate each sentence with its syntax, or semi-automatically, where a parser assigns some syntax which linguists then evaluate for correctness. Frequent substructures obtained from such annotated corpus can help in estimating the probability that a particular nonterminal in the language grammar is expanded according to a given rule. The estimated probabilities are useful in building probabilistic context-free grammars for languages [47, 118].

Tree pattern mining is also useful in several other application domains. Techniques to mine frequent substructures are useful in analyzing XML repositories, in constructing effective database indices [125, 216], in designing caching policies for XML indices [202], in examining parse trees [25], in automatically building mediated schema [70, 180], and in many other applications. The essential problem in these instances can be abstracted to the one that of *discovering frequent subtrees from a set of rooted ordered trees*. It is referred to as *frequent subtree mining*.

**Chapter Organization:** Rest of this chapter is organized as follows. We formally define the problem and provide required background details in Section 4.1. The related research and existing algorithms are presented in Section 4.2. We then present
two sequential frequent subtree mining algorithms in Section 4.3. Empirical evaluation of proposed methods is presented in Section 4.4, and we finally conclude in Section 4.5.

4.1 Background and Problem Definition

In graph theoretic terms, a rooted tree \( T = (V, E, r) \) is a connected acyclic graph where \( V \) and \( E \) are the set of vertices and edges, respectively. The node \( r \in V \) is called as the root node of \( T \). Since \( T \) is acyclic, \( |E| = |V| - 1 \). A smaller tree \( S_i = (V_i, E_i) \) is said to be an induced subtree of \( T \) if \( S_i \) is connected, \( V_i \subseteq V \), and \( E_i \subseteq E \). In other words, \( \forall e = (v_p, v_c) \in E_i, v_p \) is the parent of \( v_c \) in \( T \). Any induced subtree of \( T \) can be obtained simply by deleting vertices and edges from \( T \). Similarly, a tree \( S_e = (V_e, E_e) \) is said to be an embedded subtree of \( T \) if \( S_e \) is connected, \( V_e \subseteq V \), and \( \forall e = (v_a, v_d) \in E_e, v_a \) is the ancestor of \( v_d \) in \( T \). In other words, an induced subtree preserves parent-child relationships from \( T \) whereas an embedded subtree respects the ancestor-descendant relationships. Each occurrence of a subtree \( S \) in a tree \( T \) is called as an embedding of \( S \) in \( T \). An embedding refers the set of vertices in \( T \) that are matched with vertices in \( S \).

Let \( T \) be a tree and \( S \) be a small subtree. Let \( \delta_T(S) \) be the number of occurrences of the subtree \( S \) in the tree \( T \). Assume that \( d_T \) is an indicator variable that is defined in such a way that \( d_T(S) = 1 \) if \( \delta_T(S) > 0 \) and \( d_T(S) = 0 \) if \( \delta_T(S) = 0 \). The support of the subtree \( S \) with respect to \( T \) can then be defined in terms of \( \delta_T(S) \) and \( d_T(S) \).

There are primarily two ways to define the support of \( S \) — transaction-based and occurrence-based. The former counts the number of trees in which \( S \) occurs and the latter counts the total number of embeddings (or matches) in the database. Consider a database \( D = \{T_1, T_2, ..., T_n\} \). The support of the subtree \( S \) can be computed as follows:

\[
\text{Transaction Support} : \sup^t(S, D) = \sum_{i=1}^{n} d_{T_i}(S)
\]

\[
\text{Occurrence Support} : \sup^o(S, D) = \sum_{i=1}^{n} \delta_{T_i}(S)
\]

In this article, we mainly consider the transaction-based support even though our algorithms and optimizations are not limited by the way support is defined.
Definition: 4.1.1. **Frequent Subtree Mining:** Given a database of rooted ordered trees, enumerate the set of all frequent embedded subtrees (FS) i.e., the subtrees whose support is greater than a user defined minimum support threshold \( \text{minsup} \).

![Diagram of database trees and patterns](image)

(a) Database Trees and Patterns

(b) Pattern Growth

**Figure 4.1:** Frequent patterns and pattern growth mechanism

The pattern \( P_1 \) in Figure 4.1 has one embedding (i.e., matching) in both database trees, whereas \( P_2 \) occurs only in \( T_1 \) with 2 embeddings. If the minimum support threshold is set to 2 then only \( P_1 \) is considered frequent. A variant of this problem mines for induced subtrees which preserve parent-child relationships as opposed to embedded subtrees which preserve ancestor-descendant relationships.

The process of pattern mining can be divided into two logical phases — candidate generation and support counting. The first phase generates all candidate subtrees, which are then evaluated for their frequency in the second phase. The key challenge in the first phase is to efficiently traverse the search space for generating the candidates in a non-redundant fashion i.e., no pattern must be generated more than once. The support counting phase, on the other hand, mainly deals with performing efficient subtree isomorphism checks for determining the frequency of candidate patterns – non-trivial in the context of tree-structured and graph-structured data.

Pattern mining algorithms can be classified into two categories. The first class of algorithms process all patterns of size \( k \) before generating any pattern of size larger than \( k \). The other class of techniques are known as pattern-growth approaches, in which a frequent subtree \( S \) is repeatedly grown with new edges (equivalently, new nodes) to yield larger candidate subtrees. The newly added edge is called an extension, and the process of edge addition is referred to as point growth. In Figure 4.1b, an
extension that adds a node \( D \) is shown as a shaded circle. The set of all subtrees generated from a single pattern \( S \) via one or more pattern growths is called as its equivalence class, which is denoted by \([S]\). If \( S \) is a single node \( v \) then the set \([S]\) contains all subtrees whose root node is \( v \).

4.2 Related Work

In this section, we first describe a general high-level structure any algorithm that mines for frequent subtrees from a database of trees. We then briefly review the literature related to the problem.

There are two important phases in any tree mining algorithm, namely candidate generation and support counting. In the first phase, the algorithms traverse the search space in order to enumerate candidate subtrees, which are then evaluated for their frequency in the second phase. The fundamental challenge in candidate generation is the search space traversal. It involves an adoption of efficient mechanisms to systematically enumerate all possible subtrees, and also efficient methods to quickly prune the search space. Depending on whether the algorithms traverse the search space in breadth-first or depth-first manner, they can be classified in to two categories, apriori-style and pattern-growth approaches. The apriori-style, also known as level-wise, algorithms generate the candidate patterns of size \( i \) (at level \( i \)) by making use of frequent patterns of size \( i-1 \), which are previously discovered. Once the candidates in a level are generated, the database is scanned to evaluate each candidate for frequency. On the other hand, pattern-growth approaches, starting from a seed pattern, generate the candidates by incrementally growing the seed pattern edge-by-edge. Such a process partitions the search space in to equivalence classes, where the patterns in each class are grown from a single seed pattern. Efficient search space traversal is important also because of its size. For example, consider the number of different subtrees in a \( n \)-node tree where the root node has \( n-1 \) distinctly labeled children nodes.

In order to prune the search space, all the pattern mining algorithms leverage the anti-monotone property – according to which, all the sub patterns of a frequent pattern are frequent. The contra-positive is “no super pattern of an infrequent pattern can be frequent”. As soon as a pattern that is not frequent is discovered, the entire portion of the search space that is underneath the infrequent pattern can be pruned.
from processing. Pattern-growth algorithms, in general, are known to be more efficient when compared to level-wise algorithms because of their superiority in terms of locality and pruning power.

The key challenge in support counting phase is to quickly check for the existence of a subtree in a given database tree. These isomorphism checks are known to be expensive, especially in case of more involved structures like graphs. A majority of subtree mining algorithms avoid the repeated executions of isomorphism checks by collecting and by maintaining some state information while generating the candidate patterns.

### 4.2.1 Mining an Ordered Forest

Here, we are given a forest of rooted trees with a specific order imposed on the set of sibling nodes. Beyer et al proposed an enumeration technique for generating all rooted ordered trees with $n$ vertices [30]. Nakano generalized this result to enumerate all rooted ordered trees with at most $n$ vertices [136]. Incidentally, Asai et al [19] and Zaki [208] also proposed similar enumeration techniques independently. All these techniques employ the following idea: *define a successor tree for each tree so that by repeatedly finding the successor tree of a derived successor tree one can generate all trees*. Such a strategy is often referred to as *reverse search* [23]. Both the approaches proposed by Zaki and Asai et al define the successor tree via *rightmost expansion*. A candidate subtree is generated by attaching a vertex to the right most branch of a subtree that is already known to be frequent. Starting from a single node $v$ (a seed}

<table>
<thead>
<tr>
<th>Type of subtrees</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordered and Induced</td>
<td>FreqT [19], AMOIT [97]</td>
</tr>
<tr>
<td>Ordered and Embedded</td>
<td>Treeminer [208], MB3 [172], iMB3 [172], Chopper and XSpanner [185], PrefixTreeESpan [218], Trips and Tides [178]</td>
</tr>
<tr>
<td>Unordered and Induced</td>
<td>uNot [20], uFreqT [141], PathJoin [196], CMTTreeMiner [53], HybridTreeMiner [52]</td>
</tr>
<tr>
<td>Unordered and Embedded</td>
<td>Treefinder [180], zaki2005emf [208], Wti-Miner [75]</td>
</tr>
<tr>
<td>Free Trees</td>
<td>FreeTreeMiner [159], Gaston [141], HybridTreeMiner [52], CFFTree [215]</td>
</tr>
<tr>
<td>Closed and Maximal</td>
<td>Treefinder [180], PathJoin [196], CMTTreeMiner [52], DryadeParent [179], CFFTree [215]</td>
</tr>
</tbody>
</table>

Table 4.1: Classification of frequent subtree mining algorithms
rightmost expansion computes the equivalence class of \( v \), where each tree
in the class contains \( v \) as its root node.

Zaki’s algorithm \textit{TreeMiner} [208] mines for frequent ordered \textit{embedded} subtrees
(see Table 4.1). It maintains a \textit{scope-list} for every subtree (or pattern) to store the set
of all its embeddings. The frequency of a given pattern can thus be determined simply
by counting the number of embeddings in the corresponding scope-list. These lists
are maintained to avoid repeated invocation of isomorphism or minor containment
checks. The functionality of rightmost expansion is incorporated in \textit{scope-list joins}
which produce the new candidate subtrees. The redundant information stored in
scope-lists greatly increases the memory usage of this algorithm, especially when the
patterns have a large number of embeddings. In addition, the run time performance
of this algorithm is significantly affected due to the use of such large scope lists in
performing the joins, and also because of pointer-based data structures.

Wang \textit{et al} proposed two algorithms \textit{Chopper} and \textit{XSpanner} [185]. Chopper re-
codes the trees in to sequences and mines for frequent subsequences using PrefixSpan
[149]. Each resulting subsequence is then evaluated against the database in order
to check if it is also a frequent subtree. This method suffers from a very high overhead
that is incurred due to processing of false positive frequent subsequences – a common
case while mining embedded subtrees. XSpanner, on the other hand, overcomes the
problem of false positives by generating the frequent patterns without \textit{explicit} candidate
generation. It grows the pattern in hand by recursively projecting the database
and by finding the frequent nodes in the projected database – an idea that is explored
in the context of itemset mining [92, 210]. However, its performance is limited due
to excessive use of pointer-based data structures and also due to the complexities
involved in the process of projection.

Zou \textit{et al} proposed \textit{PrefixTreeESpan} [218] that builds upon the idea of projection in
XSpanner and adopts a divide-and-conquer technique to enumerate the subtrees. The
main idea here is to examine the prefix-tree subtrees and divide (i.e., project) their
corresponding project-instances in to the projected database. Though it performs
marginally better than XSpanner, the fundamental limitations are similar to the
ones in XSpanner.

Tan \textit{et al} proposed an embedded subtree mining algorithm called as \textit{MB3} [172].
They later extended it to propose an algorithm called \textit{iMB3} that limits the embedding
level (\( L \)) of the mined patterns depending on user preferences [172]. It can be used
to mine both embedded subtrees (\( L=\infty \)) and induced subtrees (\( L=1 \)). Though their
initial algorithm relies on the occurrence-based support, they recently developed similar methods which use transaction-based support – referred to as $iMB3\cdot T$, hereafter. It encodes the given data set into a Dictionary data structure that is used in rest of the algorithm. It constructs an embedding list for each frequent internal node $v$ in the database in order to keep track of all the descendants of $v$. Furthermore, similar to scope-lists in TreeMiner, it maintains the set of all occurrence coordinates (i.e., embeddings) in the form of vertical occurrence lists. Both dictionary and embedding list data structures can potentially increase the memory space and they, in addition, are maintained throughout the course of execution. Occurrence lists can further increase the memory consumption if the patterns have a large number of matches, a typical case in most of the real-world data sets. Excessive memory usage coupled with poor cache performance and large amount of computation leads to a mediocre performance.

We now describe algorithms which mine for rooted, ordered, induced subtrees. In terms of computational complexity, induced subtree mining is much easier than mining embedded subtrees primarily due to the simplistic nature of parent-child relationships when compared to complex ancestor-descendant relationships. $FreqT$, proposed by Asai et al, generates the candidates in a similar manner to TreeMiner. It reduces the number of rightmost expansions by employing two techniques, node-skip and edge-skip, which rely on infrequent nodes and edges, respectively. It keeps track of embeddings of a pattern by maintaining the occurrences of its right-most leaf. Such simple strategies are difficult to extend for mining embedded subtrees.

Hido et al [97] proposed AMOIT that employs a level-wise approach to mine induced subtrees from a single large tree using the occurrence-based support. This algorithm shares a lot of similarities with the classical Apriori itemset mining algorithm [7]. For example, the candidates are enumerated using what they call as right-left-joins, which essentially produce a $(k+1)$-size subtree by joining two subtrees of size $k$ which differ only in their right and left most leaf nodes. Such a method can not generate chain trees (i.e., paths), and AMOIT employs a separate mechanism to enumerate them. For each of the generated candidate, database is scanned to determine its frequency. Authors show that $FreqT$ generates some infrequent candidates which are not generated by AMOIT, thereby argue about its performance. However (as the authors admit) this algorithm, like any other level-wise algorithm, has a large memory footprint since it has to store a large number trees in memory at any given time.
4.2.2 Mining an Unordered Forest

Here, we are given a database (forest) of unordered trees, where the children of a given node form a set of siblings instead of sequence of siblings. Therefore, ordered trees that only differ in permutations of the ordering of siblings are to be considered the same unordered tree. Every unordered forest corresponds to a unique ordered forest if we sort the members of each family appropriately, using an ordering mechanism introduced by Scions in 1968 [114, 162]. Clearly, for a given unordered tree $U$, there exist several ordered trees corresponding to $U$. Our of all, a single ordered tree can be chosen as $U$’s canonical representation. It is the one that is lexicographically smaller (or larger) among all possible orderings. Such a canonical representation can easily be obtained by leveraging Scions’s family ordering methodology.

All the unordered subtree mining algorithms follow Scions’s approach and they differ only in the way they represent the ordered trees. A significant difference between ordered tree mining and unordered tree mining is that while generating candidates, unordered tree mining algorithm has to make sure that the generated candidates are canonical. Otherwise, it may either generate wrong patterns or generate redundant patterns which add to the computational overhead.

Asai et al have extended their ordered tree mining algorithm (FreqT) to develop an unordered induced subtree mining algorithm $u$Not [20]. It relies on the canonical ordering proposed by Nakano and Uno [137], and generates candidate subtrees by reverse search, in a similar manner to FreqT. They represent a given ordered tree using a depth-label sequence, where each node is denoted using a pair (depth,label) and all the pairs are arranged in the depth-first order. Associated with each canonical pattern $P$, $u$Not stores its depth-label sequence (say $DL(P)$), and a stack. Each entry in the stack corresponds to a node $v$ that is on the right most path of $P$, and contains the two pointers pointing to $v$’s last two siblings in $DL(P)$ and also their label information. $u$Not leverages the information in this stack to make sure that each generated candidate from $P$ is also canonical.

A technique similar to $u$Not was independently proposed by Nijssen et al, which is called as $u$FreqT [141]. They show that the complexity of enumerating unordered trees is not higher than the complexity of enumerating ordered trees. $u$FreqT also represents each ordered tree using the depth-label sequence. They determine the frequency of a pattern by formulating the problem as a variant of bipartite matching problem. For each node in the pattern, they store the set of all its mapping nodes in
the pattern tree in a data structure called Map. The entries in Map are inter-linked (using pointers) such that the link structure resembles the pattern tree. If \( n \) is the number of nodes in the largest frequent pattern tree, and \( m \) is the length of longest mapping list, the memory demand of uFreqT is of order \( O(nm) \) [141]. \( m \) may be large for patterns with large number of patterns, thereby increasing the memory footprint of the algorithm.

Chi et al have proposed CMTTreeMiner [53] and HybridTreeMiner [52], which essentially operate in a similar manner except that their representation of ordered trees are different. They rely on breadth-first canonical forms (BFCF) and depth-first canonical forms (DFCF). These forms are obtained by annotating the breadth-first and depth-first traversals with some special characters, which denote the backtracks while traversing the tree. They allow extensions on all leaf nodes, and therefore the evaluation of valid range of labels (for generating canonical candidates) has to be carried out for each extension on each leaf node – making it an expensive process. Similar to TreeMiner, they also store the embeddings in an occurrence list with a lot of redundant information. These algorithms also store the embeddings for each automorphism, further increasing the memory usage.

We now turn our focus to algorithms which mine unordered embedded subtrees. Here, the key challenge is that the minor containment check in unordered subtrees is NP-Complete. The first algorithm in this category is TreeFinder due to Termier et al [180]. It uses Inductive Logic Programming approach and represent the trees using relational encoding that can potentially be very long, since it captures all ancestor-descendant relationships. It operates in two steps. First, it treats the relational encodings of trees as transactions and computes maximal frequent item sets. These itemsets are used to cluster the input trees in which the same pair of labels occur together frequently enough in ancestor relation. It then uses Least General Generation (LGG) and \( \theta \)-subsumption to generate maximal common patterns from each cluster. However, it is not a complete method, i.e, it can miss many frequent subtrees, especially as support is lowered or when the different trees in the database have common node labels. Further, the LGG operation is expensive and NP-complete.

Feng et al proposed WTIMiner [75] that uses canonical representation based on DFS encoding of TreeMiner. It is similar to TreeFinder in the sense that the frequent subtree mining is re-casted in to itemset mining. The motivation is that the problem of combinatorial explosion of the search space while mining itemsets is much smaller than that of the one while mining subtrees. While TreeFinder uses apriori-style itemset
mining algorithm, WTIMiner uses FP-Growth [92] style algorithm. Once the itemsets are found, it scans the database for each itemset to count all the corresponding subtrees. The number of database scans is dependent on the number of frequent itemsets that are generated. Though this method is complete, it is inefficient since the structural information is lost while mining for frequent itemsets. Further, the number of false positives may potentially reduce the performance (similar to Chopper [185]).

Zaki proposed a pattern-growth method zaki2005emf [208] for mining unordered embedded subtrees. Its canonical representation is built on top of TreeMiner’s sequences. Similar to TreeMiner, it enumerates the candidates by following the rightmost expansion. Starting from vertices with distinct labels, using prefix extensions, and retaining only canonical forms for each automorphism group, it enumerates all unordered trees. It uses prefix equivalent classes and scope-list joins to generate candidates which are most likely to be frequent. Since the overall structure of the algorithm is very similar to TreeMiner, zaki2005emf also suffers from large memory footprints and poor run times.

4.2.3 Mining a Forest of Free Trees

As mentioned earlier, multiple ordered trees may correspond to a single unordered tree. Similarly, multiple unordered trees rooted at different nodes correspond to a single free tree. Unordered trees are handled by enforcing an order on each set of siblings (see Section 4.2.2). Similarly, free trees are tackled by finding an artificial root node(s). This is done by iteratively removing the leaf nodes until a single node (the center) or an edge (bi-centers) is obtained. Li an Ruskey gave an algorithm to enumerate free trees [121] (exercise 90 in [114]). All the free tree mining algorithms leverage such a strategy to systematically enumerate the candidates.

FreeTreeMiner [159] proposed by Rückert et al encodes a given free tree in two steps. First, it finds the canonical center(s) of the tree and builds the rooted unordered tree from the free tree. Second, it orders the nodes in the rooted tree to obtain an ordered tree, that is, the canonical form. This algorithm stores the tree in the form of depth-label pairs, which are arranged in a level-wise or breadth-first manner. It keeps track of leaf nodes which are farthest from the root node, and use them while growing the given pattern. It computes the frequency of a pattern by scanning the database, and it also collects frequency of possible extensions. However, it mines only for induced subtrees.
Nijssen et al proposed a graph mining algorithm called Gaston [141] that can systematically generates all frequent sequences, trees, and then graphs. They have developed an encoding method that is similar to uFreqT and uNot, and it relies on the concept of backbone paths which are essentially the longest paths in the tree. This algorithm also uses depth-label sequences and employs rightmost expansion to generate candidate trees. Necessary evaluations are done so that no candidate is generated that is not canonical. This again mines only for induced subtrees.

Chi et al proposed HybridTreeMiner [52] that relies on the breadth-first canonical form (see Section 4.2.2). They extended the BFCF ordering by finding the centers, like all the other approaches. Once the canonical forms are obtained, they simply leverage the algorithm that they developed for mining unordered induced trees.

4.2.4 Other Variants

There exist various other algorithms which formulate the problem in a slightly different manner. An important class of such algorithms are the ones which mine for closed and maximal subtrees. Both maximal and closed sets are much smaller in size yet represent the same information as that of the set with all frequent subtrees. However, one can not retrieve the support values of individual subtrees from maximal sets whereas closed sets can provide support information also. Closed and maximal sets are usually employed in order to limit the memory requirements of the mining algorithms. We now briefly mention about different algorithms which mine for such sets.

Chi et al proposed CMTreeMiner [52] that mines both closed and maximal sets. Their method relies on a concept called blanket. The blanket of a tree provides the set of immediate super trees that are frequent. The right-blanket consists of trees which are obtained by adding a new node to one of the node that is on the right most path of the pattern. The left-blanket consists of all other immediate super trees. By comparing the occurrences of a given subtree with the occurrences of its blanket subtrees, it determines whether the original subtree is closed or not. However, they mine only for induced subtrees, extending them to embedded subtrees is not trivial.

Xiao et al has proposed PathJoin [196] that finds maximal unordered induced subtrees. It first discovers the set of maximal frequent paths, then it finds the subtrees by joining the paths. Once all the frequent subtrees are found by joining the paths, the ones which are not maximal are pruned by employing a post-processing step.
Such a strategy will suffer from a significant overhead if the if the number of false positive paths is very high. Furthermore, this algorithms mines only the induced subtrees, and in addition, it assumes that no two siblings have the same label—an unrealistic assumption.

Balcazar et al proposed an approach for mining closed induced sets using natural representations [24]. The most interesting part of their approach is the way the trees are represented. A natural representation of a tree $t$ corresponds to a pre-post-order traversal of $t$ where each number of the sequence represents the depth of the current node in the traversal. Such a sequence starts and ends with a zero, and the difference between any two consecutive entries would be equal to one. It employs a pattern-growth approach that is similar to TreeMiner and GSpan [200] for generating the candidate trees. It mines the closed subtrees by establishing a Galois connection between the power sets of transactions and corresponding subtrees [181].

Termier et al proposed DryadeParent [179] to mine closed unordered embedded subtrees. In order to deal with hard minor containment problem for unordered embedded subtrees, they simplify the problem by assuming that no two siblings can have the same label (similar to PathJoin). Such an assumption on one hand is unrealistic, and on the other, significantly reduces the complexity of the problem. Furthermore, they encode the given database in a matrix format where the matrix stores the set of all ancestor-descendant relationships. The matrix representation is very similar to the embedding lists employed in iMB3 [172], which we show in later sections that it greatly increases the memory footprint. Their occurrence lists further add to the space overhead.

Zhao et al recently proposed CFFTree [215] that mines closed frequent free trees. By mining only the closed trees, they show that CFFTree outperforms other free tree miners [159, 52], which mine the complete set of frequent trees, by up to 10 times. They employ techniques like equivalent occurrence pruning, safe position pruning, and safe label pruning, which in spirit, are very similar to the ones proposed in CMTreeMiner.

Shasha et al have proposed algorithms for discovering the set of all cousin pairs in a given tree whose distance is within $maxdist$ and which occur more than $minoccur$ times, where $maxdist$ and $minoccur$ are two user defined parameters [166]. In order to find frequent cousin pairs in a database of trees, their algorithm first finds the cousin pairs in each tree and then evaluates the frequencies of each of the resulting
pair, evolutionary trees, evaluating the consensus of equally parsimonious trees, and finding kernel trees of groups of phylogenies.

Katsaros et al proposed an algorithm Mabers that discovers the schema in the given semi-structured data by finding frequent substructures [108]. A key contribution here is to treat each rooted ordered tree as a type-graph in programming languages, and thereby to leverage the hashing mechanism introduced by Katzenelson et al [109] for encoding the trees as magic numbers. The resulting hashes are used in reducing the overhead of containment checks. However, they employ an apriori-style algorithm for enumerating the subtrees.

4.3 TRIPS and TIDES: New Algorithms for Tree Mining

Our algorithms operate in three important phases: data transformation, candidate generation, and support counting. During the first phase, we rely on novel injective mappings from trees to sequences in order to sequentialize the given database trees. The later two phases operate completely on the sequential representation of the database to produce frequent subtrees. Our mining algorithms are not tied to any specific encoding method. They can work with any type of sequences as long as the sequences follow specific properties identified in later sections.

The framework of our subtree mining algorithms is shown in Figure 4.2. The given database \( D \) is first transformed in to sequences \( T(D) \) using our data transformation methods. Once the database is transformed, the set of frequent 1-nodes (i.e., \( F1 \)) is determined in the frequency counting step. Each node \( f \) in \( F1 \) is treated as a seed pattern for partitioning the search space in to equivalence classes. Each such seed pattern along with \( T(D) \) is then fed in to the mining block that consists of both candidate generation and support counting phases. The mining block operates on \( T(D) \) to generate candidate subtrees, which are evaluated in the supporting counting phase to produce frequent extensions. Each frequent extension correspond to a new frequent subtree that is generated, and is sent to the output. The mining block also produces a projected database, a subset of \( T(D) \), that contains the set of trees in which the pattern that is currently being mined has at least one embedding. The idea of projected database is essentially to leverage the anti-monotone property of frequent patterns. If a pattern \( P \) has non-zero embeddings in \( T'(D) \subseteq T(D) \) then all the subtrees generated from \( P \) has embeddings only in the projected database
Both the projected database, and the frequent extensions which are generated are fed back in to the mining block to produce larger subtrees recursively.

4.3.1 Data Transformation

Given database trees are first converted into sequences. Our tree mining algorithms are built upon two sequencing methods described in Section 2.2.1 – Prüfer sequences and depth first sequences. As discussed earlier, both types of sequences are equivalent. Any algorithm that is built on top of one type of sequences can be modified to leverage sequence representation of other type. For the sake of exposition, we first describe all our algorithms in the context of Prüfer sequences. The resulting Tree mIining algorithm using Prüfer Sequences is referred to as TRIPS. We then summarize the methods for depth first sequences in Section 4.3.5.

To summarize from Section 2.2.1, Prüfer sequence representation of a database tree $T$ relies upon the post-order traversal of its tree nodes. It consists of two sequences – Numbered Prüfer Sequence ($NPS_T$) and Label Sequence ($LS_T$). The label sequence can be obtained by concatenating tree node labels in post-order. The $NPS$ is then given by the post-order traversal numbers of parent nodes. Two example trees ($T_1$ and $T_2$) and corresponding Prüfer sequences are shown in Figure 4.3b.
4.3.2 Candidate Generation

Given a frequent subtree $S$, candidate generation process generates the subtrees which are potentially frequent. We employ a pattern-growth approach that operates on all the sequences in which $S$ has at least one embedding to discover edges (or nodes) with which $S$ can be extended (or grown). Each such edge together with $S$ defines a new candidate subtree. Candidate subtrees whose support is greater than or equal to $\text{minsup}$ are processed further to generate bigger subtrees.

Assume that $S$ is grown with an edge $e$. There are a number of growth points in $S$ at which $e$ can be added. As such $e$ can be attached to any node in $S$, and on top of that, it can be added at any position in the list of a given node's children. Note that in case of unordered trees, there is exactly one way in which $e$ can be attached to $v$. In order to limit the number of growth points, we stick to a systematic way of generating candidates – we always attach $e$ to one of the nodes which are on the Left Most Path (LMP) of $S$. Furthermore, $e$ is always added as the first child. It is fairly simple to see that such a mechanism indeed generates every subtree (i.e., complete) in the search space and generates each subtree only once (i.e., non-repeating).

Each candidate subtree that is generated from a pattern $S$ is uniquely represented using what are called as extension points. They uniquely identify different subtrees generated from $S$ by attaching nodes (with different labels) to nodes (at different positions) on the LMP of $S$. Each extension point is of the form $(l, p)$, where $l$ is the label of the node that is being added to $S$, and $p$ is the PON of the node in $S$ to which $l$ is attached. In other words, $l$ identifies what is being added to $S$, and $p$ denotes where in $S$ it is added.
Example: Figure 4.3a illustrates the candidate generation process where two patterns $S_1$ and $S_2$ are extended to produce three new subtrees $S_{11}$, $S_{12}$, and $S_{21}$. The new edges with which the patterns are extended are shown as dashed lines. In case of $S_1$, the LMP is just the edge $(A,B)$. New edges can be attached either to $A$ or to $B$. Therefore, $S_{11}$ and $S_{12}$ are valid extensions from $S_1$, and are denoted as $(A,3)$ and $(A,1)$, respectively. $S_{21}$ is not a valid extension of $S_1$ because the node $D$ has to be attached to node $C$ that is not on the LMP of $S_1$. However, $S_{21}$ defines an extension $(B,3)$ for $S_2$ since it can be obtained by attaching $B$ to node $A$ (PON=3) that is on the LMP of $S_2$.

Given such a candidate generation procedure, a naive way to generate candidates from $S$ is to attach nodes with every single label to a node on the LMP of $S$. The number of generated candidates would then be equal to (cardinality of the label set)* (number of nodes on the LMP of $S$). However, it is highly likely that a majority of these candidates are redundant, meaning that their support is zero i.e., they never occur in the data set. For example, with respect to the database in Figure 4.3b, both $S_{11}$ and $S_{12}$ are redundant patterns because they neither occur in $T_1$ nor in $T_2$. On the other hand, $S_{21}$ is a subtree of $T_1$, and therefore, is a non-redundant pattern. Redundant patterns can significantly increase the computation overhead and thus should never be generated. Therefore, the set of subtrees generated not only have to be complete and non-repeating but should also be non-redundant. We employ a schema-driven approach that generates extensions by looking at the nodes in those trees in which the pattern that is being extended has at least one embedding. Since such an approach is guided by the database tree nodes, it generates only the non-redundant patterns. More interestingly, our approach can work off the sequential representation of trees by transforming the growth on trees to the growth on corresponding sequences.

Lemma 4.3.1. Let $S$ be a pattern with its sequential encoding $CPS(S) = (NPS, LS)$. If $(l, p)$ is an extension to $S$ that results in a new candidate $R = S \cup (l, p)$ then the prefix of $R$’s Prüfer sequence corresponds to the extension $(l, p)$.

Proof. Say that the node to which $l$ is attached is $v$ i.e., PON($v$) = $p$. Since $(l, p)$ is an extension of $S$, node $v$ is on the LMP of $S$ and $l$ is attached as the first child to $v$. $l$ is thus visited first, before any other node that was in $S$, in the post-order traversal on $R$. Since $CPS(R)$ is ordered by the post-order numbers, $(l, p)$ is its first
entry. Therefore, extension points on the LMP of $S$ corresponds to prefixes of the resulting pattern’s (i.e., R’s) prufer sequence.

**Lemma 4.3.2.** Let $S$ be a pattern that is being extended, and has an embedding $E$ in a database tree $T$. The set of all extensions to $S$ in $T$ are present to the left of $E$ in $T$’s Prüfer sequence.

*Proof.* From Lemma 4.3.1, valid extensions to $S$ are the prefixes of the Prüfer sequences of resulting patterns. If the resulting pattern is in $T$ then the extension point has to be on the left of $S$’s embedding in $T$’s sequence.

**Lemma 4.3.2** essentially limits the portion of tree nodes which need to be scanned while generating the extensions from $S$ – the ones which are on the left of $S$’s embedding in $T$. There are two important things to notice here: (i) extensions have to be defined with respect to a particular embedding of $S$ in $T$, and (ii) not all the nodes which are on the left of $S$’s embedding are valid extensions. Consider an example pattern $S$ in Figure 4.3b that has two embeddings ($E_1$ and $E_2$) in $T_1$ and one embedding ($E_3$) in $T_2$. A valid extension with respect to $E_1$ need not be a valid extension with respect to $E_2$ since the two embeddings might not have any node in common. The node with PON=3 (in $T_1$) is an extension to $E_1$ but not for $E_2$. This is because the two embeddings might not have any node in common. An extension point with respect to one embedding might not even be connected to a node in another embedding. Even when two embeddings have a common node, the position of the common node might not be the same. Hence, an extension point is always defined with respect to a particular embedding. Similarly, even though the node with PON=1 appears on the left of $E_1$, it is not a valid extension point because it is not connected to any node in $E_1$. We capture this notion in Theorem 4.3.1 that precisely identifies the set of valid extensions for a pattern.

**Theorem 4.3.1.** Let $S$ be a pattern that is being extended, and has an embedding $E$ in a database tree $T$. Also, say that the node $v$ is on the left of $E$ in CPS($T$). If $v$ is connected to a node $u$ that is in $E$, then $v$ is a valid extension for $S$ in $T$ with respect to $E$.

*Proof.* From the definition of an extension point, the new node has to be connected to a node that is on the LMP of $E$. Since $v$ is on the left of $E$, it is a possible extension to $S$ (from Lemma 4.3.2). The node $u$ must be on the LMP on $E$ because $v$
is connected to $u$. If not, then the node that is connected to $u$ can not be on the left of $E$. Thus, $v$ is connected to a node that is on the LMP of $E$ and therefore $v$ defines a valid extension for $S$ with respect to $E$.

The entire process of finding extensions for a given pattern $S$ can thus be summarized as follows:

(i) Find the set $S_D = \{ T \mid T \in D \land T \text{ has at least one embedding of } S \}$.

(ii) For each embedding $E$ of $S$ in $T \in S_D$, determine the set of nodes $E_{S,T} = \{ v \mid v \in CPS(T) \land v \text{ is a potential extension to } E \}$ by leveraging Lemma 4.3.2.

(iii) For each $v \in E_{S,T}$, check if it is connected to a node that is in $E$. If yes, then $v$ defines a valid extension (from Theorem 4.3.1).

(iv) Repeat steps (ii) and (iii) for each embedding in $T \in S_D$.

In step (iii), we need to evaluate if $v$ is is connected to any node in $E$ (see Theorem 4.3.1). We refer to these checks as connectivity checks. For the sake of simplicity, consider the case where extensions are generated for induced subtrees. Say that node $v$ that is on the left of $E$ has PON=$k$ with corresponding CPS entry $(NPS_T[k], LS_T[k])$. The Connectivity check evaluates if $v$’s parent is in $E$ or not. More precisely, it examines whether or not $NPS_T[k]$ is in $E$ by traversing the left most path of $E$. The exact position $p$ at which $v$ is connected to $E$ can be computed simply by maintaining a simple counter that is updated while scanning the LMP. The resulting extension point would then be $(LS_T[k], p)$. If $S$ has more than one embedding, then the connectivity check for $v$ has to be evaluated against all the embeddings for which the node $v$ is on the left. Since such a connectivity check evaluates only $v$’s parent (i.e., $NPS_T[k]$), it is useful in finding induced subtree extensions. However, while mining for embedded subtrees, the node $v$ is considered as a valid extension point if any of its ancestors is part of the embedding. A connectivity check thus evaluates each ancestor of $v$ (by traversing the path from $v$ to the root node) until an ancestor that is part of the embedding is found. These connectivity checks essentially explores the parent-child relationships while mining induced subtrees, and the ancestor-descendant relationships while mining embedded subtrees.

**Example:** Consider the running example in Figure 4.3b, assume that the $S$ is being extended to generate new candidate subtrees. The embeddings of $S$ in $T_1$ and $T_2$ are marked as dashed lines in red color. In this example, we denote a node as $[l, n]$,
where \( l \) is the label of the node and \( n \) is its PON. From Lemma 4.3.1, the connectivity checks have to be performed on nodes from \([D, 5]\) to \([B, 1]\). The node \([D, 5]\) is a valid extension to \( E_1 \) because its parent node \([C, 6]\) is part of \( E_1 \). Since it is attached to \( E_1 \) at position 1 (PON of \( D \)’s parent in \( S \)), \((D, 1)\) defines an extension point for \( S \) in \( T_1 \) with respect to \( E_1 \). Consider the next node \([F, 4]\) whose parent \([D, 5]\) is not part of any of the two embeddings. However, \( F \)’s ancestor \([C, 6]\) is part of \( E_1 \) and as a result, \((F, 1)\) is a valid extension. Note that, \((F, 1)\) is not considered as an extension while mining for induced subtrees as the connectivity check, in which case, stops with \( F \)’s parent \([D, 5]\). Similarly, the nodes \([B, 3]\) and \([B, 1]\) result in extension points \((B, 2)\) and \((B, 1)\) with respect to \( E_1 \) and \( E_2 \), respectively. Note that, the node \([C, 2]\) must be evaluated against \( E_1 \) even though it is part of \( E_2 \). Furthermore, \([B, 1]\) is evaluated against both \( E_1 \) and \( E_2 \) because it is to the left of both the embeddings. In total, the candidate generation process discovers: four extensions in \( T_1 \) – \((D, 1), (F, 1), (B, 2), (B, 1)\); and two extensions in \( T_2 \) – \((D, 1), (B, 2)\) (see Figure 4.3b).

**Embedding Lists**

In the first two steps of above described candidate generation process, one needs to know the set of all embeddings of \( S \) so that the extensions can be found against them. Instead of discovering these embeddings by invoking subtree isomorphism checks, we store the list of embeddings of the pattern that is being mined in a special data structure called as an **Embedding List**(EL). Most of the existing algorithms leverage such a trade off between time and space by storing the embeddings in some or the other form. However, the data structures used in these algorithms suffer from various issues: some employ pointer-based data structures which limit the ILP; and some store huge amount of redundant, and even, persistent data that significantly increases the memory overhead.
Unlike state-of-the-art approaches, we design a simple array-based embedding structures which grow and shrink depending on the size of the pattern that is being mined. The benefits are two fold: they store exactly the information that is required, nothing more nothing less; and they never keep any type of persistent information. Such array-based designs also help in improving the locality of overall mining algorithms.

The entries of the list are divided into different sections where each section stores the matching nodes of a particular node in the pattern that is being mined (see Figure 4.4). Different sections are separated by a special entry that is shown as a shaded cell in the figure. All the other entries are of the form \((m, \text{prnt})\) where \(m\) is the PON of the matching node and \(\text{prnt}\) is a pointer that points to the parent node in the embedding. Since the entries in the first section correspond to matches for the root node, they do not point to any other entries. The complete embedding of a given pattern can thus be enumerated by following the pointers starting from an entry in the last section. The number of total embeddings of the pattern in hand is thus equal to the number of entries in the last section. We do not need to store the entire embedding since the candidate generation process requires only the nodes on the LMP. We thus store only those nodes which are on the LMP of the embeddings, which greatly reduces the amount of space occupied by these lists.

The size of the embedding lists grow and shrink proportionally with the size of the patterns being mined. The method in which these lists are maintained is described in Algorithm 8. For a given pattern \(S\), new candidate subtrees are generated in line 4. Each newly generated subtrees is mined recursively in line 7. Before mining an extension \(S + e\), the embedding list of \(S\) i.e., \(EL(S)\) is transformed into \(EL(S + e)\) simply by appending the matching nodes for the newly added label (from \(e\)). Once
the mining of \( S + e \) is complete, \( EL(S + e) \) is transformed back into \( EL(S) \) by deleting the last section of \( EL(S + e) \). The list of \( S \) can be reused for mining the next extension. These transformations improve the locality, maximizes the reuse, and are easy to implement. For example, line 7 involves a simple pointer retraction. **Example:** Figure 4.4 shows an example pattern and its embedding list with respect to a database tree \( T_1 \) (from Figure 4.3b). The first section of the list hosts the matches for root of the pattern \( P_1 \) (node \( A \)). Matching nodes for node \( C \) are stored in the second section along with appropriate pointers into the first section. \( P_1 \) is grown into a bigger pattern \( P_2 \) by adding an extension \( (B, 2) \). Before processing \( P_2 \), its embedding list is constructed by appending entries to the list associated with \( P_1 \). \( EL(P_2) \) is constructed by appending matches for node \( B \) to \( EL(P_1) \). Note that not all the nodes in \( T_1 \) with label \( B \) correspond to the extension \( (B, 2) \). By traversing the pointers from the last section, we can enumerate the LMP nodes (3 and 7, in this example) in all embeddings.

### 4.3.3 Support Counting

This step evaluates the frequency of each extension found in the candidate generation step. While generating the extensions of a subtree \( S \), a hash table \( H \) that is known as Support Structure is maintained. \( H \) stores extensions and associated counts. As soon as a new extension \( e \) is found, it is hashed into \( H \). If \( e \) is hashed to an empty bin (i.e., \( e \) is not there in \( H \)) then the associated count is initialized to 1. Otherwise, the count is incremented by 1 to indicate its new support. Once all the trees are scanned for extension points, \( H \) contains the set of all extension points along with
their supports. The resulting set of frequent patterns generated from \( S \) is then given by, \( Frq = S \cup \{ e | e \in H \land support(e) \geq \text{minsup} \} \). Each pattern in the set \( Frq \) is considered again by the candidate generation process to produce larger patterns.

**Example:** The support structure for the example pattern in Figure 4.3b is shown in Table 4.2. The extension points found while extending \( S \) for \( T_1 \) are all hashed into the structure. Each extension is created with support 1, denoting its occurrence in \( T_1 \). When \((D, 1)\) is generated from \( T_2 \), associated count is incremented because it is already present in \( H \). Similarly, count associated with the extension point \((B, 2)\) is incremented by 1. If the minimum support is 2 then only two, \((D, 1)\) and \((B, 2)\), of the four extension points would be frequent and will be considered again for candidate generation.

### 4.3.4 Putting it All Together

The complete subtree mining algorithm is described in Algorithm 9. Given database of trees are first transformed into sequences \((D)\). While transforming, the set of all frequent nodes \((F_1)\) is constructed. Each frequent node \( f \in F_1 \) act as a seed pattern and its equivalence class is recursively mined by invoking the procedure \texttt{mineTrees}.

\texttt{mineTrees} enumerates all frequent subtrees whose root node is \( f \). At every level of recursion, a new pattern \texttt{newpat} is generated by extending \texttt{pat} with an extension point \((\text{lab}, \text{pos})\) (line 1). The input parameter \texttt{tidlist} is a projection of the database with respect to the pattern \texttt{pat}. It gives the list of trees in which \texttt{pat} has at least one embedding. Only the trees in this projected data set are considered in extending \texttt{pat}. Before starting the candidate generation process, the embedding list of \texttt{pat} i.e., \texttt{EL(pat)} is transformed into the list of \texttt{newpat} (lines 3-6). In order to do that, it scans each tree in \texttt{tidlist} to determine the matching locations of \((\text{lab}, \text{pos})\) – obtained locations are appended to \texttt{EL(pat)}. It also constructs the projected data set \((\text{newtidlist})\) for the new subtree \texttt{newpat}. Once the embedding list for \texttt{newpat} is constructed, extensions are found in each tree in \texttt{newtidlist} with respect to each embedding of \texttt{newpat} (lines 8-12). In the actual implementation, the loop in line 3 (that finds the embeddings of \texttt{newpat}) and the loop in line 8 (that finds the extensions) can be combined thereby making a single scan on the projected database. For each frequent extension point, \texttt{mineTrees} is called recursively to mine larger patterns (lines 13-15). For a given pattern, this algorithm generates and finds the frequent patterns simultaneously by performing a single scan on the projected database.
Algorithm 9 Trips and Tides: Algorithms for frequent subtree mining

Input: \( \{T_1, T_2, \ldots, T_N\}, minsup \)

\((D, F_1) = \text{Transform}(T_i): 1 \leq i \leq N\)

for each \( f \) in \( F_1 \) do

\( \text{mineTrees} (\text{NULL}, (f, -1), D) \)

\text{mineTrees} (pat, extension \( e=\text{(lab, pos)} \), tidlist)

1: \( \text{newpat} = \text{extend} (\text{pat}, e) \)
2: output \( \text{newpat} \)
3: for each \( T \) in tidlist do
4: \hspace{1em} if \( (l, pos) \) is an extension point for \( \text{pat} \) in \( T \) then
5: \hspace{2em} update embedding list of \( T \)
6: \hspace{2em} add \( T \) to \( \text{newtidlist} \)
7: \( H = \text{NULL} \)
8: for each \( T \) in \( \text{newtidlist} \) do
9: \hspace{1em} for each node \( v \) in \( T \) do
10: \hspace{2em} for each match \( m \) in \( \text{EL}[T] \) do
11: \hspace{3em} if \( v \) is a valid extension to \( m \) then
12: \hspace{4em} add the extension point to \( H \)
13: for each \( \text{ext} \) in \( H \) do
14: \hspace{1em} if \( \text{ext} \) is frequent then
15: \hspace{2em} mineTrees (\( \text{newpat} \), \( \text{ext} \), \( \text{newtidlist} \))
4.3.5 TIDES: Tree mining using DEpth first Sequences

Our algorithms are not tied to any specific tree encoding method. We demonstrate the generality of our approach by developing a mining algorithm TIDES that is based on DFS codes. In TIDES, the database trees are first transformed into depth first sequences by performing a pre-order traversal on the trees. Similar to the properties derived in the context of Prüfer sequences, we can derive the following properties.

Lemma 4.3.3. Let $S$ be a pattern with depth first order sequence $DFS(S)$. If $(l, p)$ is an extension to $S$ that results in a new candidate $R = S \cup (l, p)$ then the postfix of $R$’s DFS code corresponds to the extension $(l, p)$.

Lemma 4.3.4. Consider a pattern $S$ that is being extended and has an embedding $E$ in a database tree $T$. The set of all possible extensions to $S$ are present to the right of $E$ in the depth first sequence of $T$.

Theorem 4.3.2. Consider a pattern $S$ that is being extended and has an embedding $E$ in a database tree $T$. Also, say that the node $v$ is on the right of $E$ in $DFS(T)$. If $v$ is connected to a node $u$ that is in $E$, then $v$ is a valid extension for $S$ in $T$ with respect to $E$.

The candidate generation process described in Section 4.3.2 can be tailored to DFS codes by making few alterations. (i) While TRIPS restricts the pattern-growth to the Left Most Path of the pattern, TIDES restricts it to the Right Most Path (RMP). (ii) An extension point $(l, p)$ now refers to a new subtree that is obtained by attaching a node with label $l$ to a node whose pre-order number is $p$. The support counting step in TIDES would be same as the one in TRIPS since this step does not require any information on sequences.

To summarize, our sequence-based candidate generation technique can be applied to any type of sequential encoding schemes as long as one can relate the growth in the trees to the growth in the corresponding sequences. This involves building relationships similar to the ones mentioned in Lemma 4.3.1 and Lemma 4.3.3.

4.4 Empirical Evaluation

We empirically evaluate the tree mining algorithms presented in previous section, on both synthetic and real-world data sets. The data sets we consider are shown in Table 4.3. All the experiments were performed on an Itanium 2 based system with
4 GB of main memory and a 1.3GHz processor. Whenever the experiments require more than 4GB, we used a system with 32GB of main memory instead of relying on virtual memory. We compare our algorithms against both TreeMiner [208] and iMB3-T [172], whose source code is obtained from the respective authors. We express the size of data sets in terms of the number of trees, and the minimum support \((\text{mins}\text{up})\) as an absolute number instead of as a percentage. Hereinafter, A data set \(D\) with \(\text{num}\) trees is denoted as \(D-\text{num}\).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DS1)</td>
<td>-T 10 -V 100</td>
</tr>
<tr>
<td>(DS2)</td>
<td>-T 10 -V 50</td>
</tr>
<tr>
<td>(DS3)</td>
<td>-f 15 -d 10 -n 100 -m 100</td>
</tr>
<tr>
<td>(DS4)</td>
<td>-f 15 -d 1000 -m 100</td>
</tr>
<tr>
<td>Cslogs</td>
<td>59, 691 trees</td>
</tr>
<tr>
<td>Treebank</td>
<td>52, 581 trees</td>
</tr>
</tbody>
</table>

Table 4.3: Data sets for evaluating TRIPS

### 4.4.1 Evaluation on synthetic data sets

In the first set of experiments, we evaluate our TRIPS algorithm on four different synthetic data sets. As shown in Table 4.3, the synthetic data sets \(DS1\) and \(DS2\) are generated using the PAFI toolkit developed by Kuramochi and Karypis (\(PafiGen\)) [90]. The data sets \(DS3\) and \(DS4\) are generated from the tree generator created by Zaki (\(TreeGen\)) [207]. The table also shows the parameter settings used for creating these data sets. The trees generated from \(TreeGen\) generator mimic the website browsing patterns of different users. The generator first creates a master website browsing tree based on the parameters such as fanout \((f)\), depth \((d)\), number of labels \((n)\), and number of nodes in the master tree \((m)\). Once the master tree is generated, it creates smaller subtrees whose number is given by another user defined parameter \(t\). The \(PafiGen\) generator, on the other hand, makes synthetic graphs using several different parameters. Since it can only create the graphs, we have extracted the spanning trees from the generated graphs and used for our purpose. The user parameters just serve as the guidelines while generating trees, the generated data set may not have the statistics exactly same as the provided parameters.
Figure 4.5: Evaluation of TRIPS on synthetic data sets

Figure 4.5 demonstrates the performance differences between our algorithm TRIPS and TreeMiner on the synthetic data sets. TRIPS consistently performs better than TreeMiner at all data set sizes by showing a speedup of up to 11.6-fold on Pafi data sets and up to 8.6-fold speedup on TreeGen data sets. We observed that the associativity in the data set plays a significant role in the overall mining time. We have observed that the trees in TreeGen data sets are much more skewed when compared to the ones generated by PafiGen. The skew refers to the distribution of number of frequent patterns over different equivalence classes. In case of DS3 and DS4, a significant number of frequent patterns are resulted from a very few equivalence classes i.e., mining these equivalence classes would be much more time taking than mining most of the other equivalence classes. Since the skew is less in PafiGen data sets, the mining times may go up quickly even at smaller data set sizes because all equivalence classes are equally tough to mine. For example, consider DS1 and DS4 data sets with with 40,000 trees. TRIPS (respectively, TreeMiner) spent around 790 (resp., 8042) seconds in mining trees in DS1-40K whereas they had to spend only 315 (resp., 2711) seconds for mining the trees in DS4-40K. Higher speedups from the Pafi data sets shows that TRIPS can handle the difficulty posed by (lesser) skewness better than TreeMiner.

Figures 4.6 (a) and (b) depict the effect of minsup on the mining time and the number of frequent patterns found in the PafiGen data sets. Not surprisingly, the execution time and the number of patterns increases as minsup decreases. Notably, the performance difference between TRIPS and TreeMiner continuously goes up with
the decrease in \( \minsup \). On \( DS2-40K \), as the value of \( \minsup \) is decreased from 5 to 1, the number of frequent patterns increased by 339 times. In this case, while the execution time of TreeMiner increased by 65 times, the mining time of TRIPS increased only by 15 times. This is because of two reasons: large number of join operations; large number of false positive patterns. First, at low support levels, TreeMiner has to perform costly join operations on a very large number of scope lists. Though TRIPS has to perform more connectivity checks (at low \( \minsup \)), the increase in run time is not very high. The time taken for a connectivity check, in general, is very small because it is performed only on nodes which are on the LMP of embedding. Moreover, for a given pattern the set of nodes for which the connectivity check is performed does not depend on the support level. It only depends on the tree size and the matching nodes of the pattern (Theorem 4.3.1), whereas in TreeMiner, the number of scope lists and hence the number of joins performed is highly dependent on the value of \( \minsup \). Second, TreeMiner naively performs the join operation on scope lists and hence it has the potential for generating a large number of false positive patterns. For example on \( DS1-10K \), at \( \minsup = 1 \), TreeMiner produced 1041 million candidates out of which only 173 million (approximately 16%) are frequent. On the other hand, TRIPS will never generate redundant candidate patterns since the process of candidate generation is non-redundant and guided by the database trees themselves (see Section 4.3.2). Thus, the strategies adopted by TRIPS enable us to mine large data sets at very low support levels efficiently. The performance of TIDES is similar to that of TRIPS on all synthetic data sets.
4.4.2 Evaluation on real-world data sets

In the second set of experiments, we evaluated the proposed algorithms on two different real data sets Cslogs and Treebank. Cslogs [207] contains web logs collected over a month in the Computer Science Department at the Rensselaer Polytechnic Institute. It contains 59,691 user browsing access patterns for 13,361 different web pages. On average, a tree in Cslogs data set has 12.94 nodes and the largest tree is of 428 nodes. The maximum depth of a tree in this data set is 85 levels. The Treebank data set [4] is derived from the field of computational linguistics, and each tree corresponds to a language treebank. A treebank is a XML document that captures the syntactic structure of English text, and provides a hierarchical representation of a sentence in the text by breaking it into syntactic units based on part of speech. This data set contains a total of 52,581 trees, which are narrow and deep recursion of element names. The largest and average size of a tree in this data set is 648 and 68.03 nodes, respectively. The maximum depth of a tree here is 36 nodes. Furthermore, Treebank has higher associativity than Cslogs data set because of more common elements. For example, elements like *Noun Phrase* (NP) occurs almost in every sentence. The distribution of label frequencies for both the data sets is shown in Figures 4.7a&b. In case of Treebank, we show only the labels whose frequency is greater than 1. As it is evident, there are more number of nodes with high frequency in Treebank than in Cslogs. For example, top 10 frequencies in Treebank are greater than 73% whereas top 10 in Cslogs are greater than just 4%. Such high associative data sets lead to patterns with large number of matches, thereby affecting the run time and memory performance.

Figure 4.7c shows the run time comparison of TRIPS against both TreeMiner and iMB3-T, on Cslogs data set. The run time in all the algorithms increases gradually as the support is decreased since more and more patterns need to be discovered. However, the rate of increase in TRIPS is small when compared to competing algorithms. When minsup is decreased, time spent by TreeMiner increases much more quickly than TRIPS. With the decrease in minsup from 1000 to 900, the mining time of TreeMiner and iMB3-T increased by 300 and 46 times, respectively. In contrast, it increased just by 6 times in TRIPS. At minsup=700, TreeMiner was aborted since the run time was more than 100 hours whereas TRIPS found the frequent patterns in 300 sec – giving a speedup of more than 1200.

Figure 4.7d shows the amount of virtual memory used by all three algorithms.
Figure 4.7: Evaluation of Trips on real-world data sets
We approximate the amount of memory usage based on the Resident Set Size (RSS) obtained from “top” command in Linux. As mentioned, real-world data sets have a skewed frequency distribution among labels. Therefore, there exist some patterns with a really large number of matches and some with very few matches. Since all the three algorithms rely on embedding lists, the memory usage shoots up (and hence the RSS) at the time of mining patterns with large number of matches. However, our algorithm uses much less memory when compared to other algorithms because of the concise representation of embeddings, we store only the LMP (see Section 4.3.2). Furthermore, the lists employed in TRIPS grow and shrink depending on size of the pattern that is being mined. At $\text{minsup} = 800$, TreeMiner and iMB3-T used 2505MB and 1638MB of memory, respectively where as TRIPS completed the mining process by using just 60.84MB of memory. After that, the memory usage of iMB3-T increases exponentially due to the way they maintain embedding lists and occurrence lists. For example, at support threshold of 700, iMB3-T used more than 25GB of main memory whereas TRIPS used a mere 336MB – showing an improvement of more than 76-times in the size of memory footprint. Note that experiments requiring more than 4GB are conducted on a system with 32GB of main memory. Note that at $\text{minsup}=600$, iMB3-T was aborted since the memory usage exceeded 32GB.

The trends observed in case of Treebank are same as that of in Cslogs (see Figures 4.7e&f). The time spent in mining and the amount of virtual memory used increased very sharply when using TreeMiner or iMB3-T. However, the amount of
memory is high when compared to the amount in Cslogs. As noted earlier, the associativity in the data set is high in Treebank. This results in larger embedding lists, even at high support thresholds – high support thresholds are chosen due to high associativity in the data set. Due to the same reason, the memory usage and run time of both TreeMiner and iMB3-T increases exponentially to unacceptable levels. At a support level of 40,000, iMB3-T is aborted after running for 47 hours since the memory footprint grew larger than the size of main memory. In contrast, TRIPS finished the mining, using 221MB of memory, in just $273sec$ – giving a speedup of more than 615-times. Similarly, TreeMiner was aborted at $minsup=35K$ after running for 100 hours whereas the run time of our algorithm was around $1125sec$ – a 316-fold speedup.

4.4.3 Comparison of TRIPS and TIDES

We now compare the performance of the mining algorithm that uses Prüfer sequences (TRIPS) against the one that uses depth first sequences (TIDES). Algorithms and optimizations designed in the context of Prüfer sequences can also be applied to depth first sequences (see Section 4.3.5). We thus expect the performance of both algorithms to be same. However, as shown in Figure 4.8, the run time of TRIPS is marginally better than the run time of TIDES. For mining Cslogs at the support threshold of 800, TRIPS took $24.6sec$ whereas TIDES took $73.7sec$. Similarly for mining Treebank data set at $minsup=35K$, TRIPS and TIDES have taken around $1,125sec$ and $1211sec$, respectively.

![Figure 4.9: Example tree in Cslogs: Distribution of P’s matches](image)
In order to reason about this difference in run times, we have looked at the specific characteristics of data set trees and frequent patterns. We have observed that most of the trees in Cslogs and Treebank are deep and oriented towards the right-most branch. For example, consider a tree $T$ that is taken from Cslogs data set – shown in Figure 4.9. The right most branch in $T$ is 71 levels deeper whereas the rest of the tree is relatively bushy and shallow. The figure also shows an example frequent 3-node pattern $P$ and the distribution of its embeddings on the nodes of $T$. The right most node $C$ in $P$ is distributed all over the right most branch, where each occurrence corresponds to an unique embedding of $P$. Now consider the process of generating candidate subtrees from $P$. TRIPS evaluates the nodes which are on the left of $P$’s embedding in order to find the extension points (from Lemma 4.3.2), whereas TIDES evaluates the nodes which are on the right of the embedding (from Lemma 4.3.4). These nodes are represented as $E$ in the figure – see the box in the bottom left corner. From the distribution of $P$’s embeddings in $T$, it is evident that the number of nodes evaluated in TRIPS is less than the number of nodes evaluated in TIDES – resulting in larger mining times for TIDES.

To further corroborate this observation, we created a synthetic data set with exactly opposite characteristics. The data set trees are created with deep left-most branches and the node labels are given in such a way that the left-most node of the frequent patterns is distributed on the deep branch. On such data sets, we found that the mining time of TRIPS, for a given value of $minsup$, is more than the run time of TIDES because the number of nodes evaluated while generating new candidates in TRIPS is more than that in TIDES. We therefore conclude that the performance differences between TRIPS and TIDES are completely dependent on the data set characteristics. There were not any significant differences in the memory usage of both the algorithms because the representational complexity is exactly the same in both the algorithms.

It is worth mentioning that the performance of TIDES, on both Cslogs and Treebank, is significantly better when compared to other algorithms like TreeMiner and iMB3Miner-T – in similar lines to the results observed in case of TRIPS (see Section 4.4.2).
4.4.4 Performance Characterization

We have used the Intel VTune Performance Analyzers [62] in order to analyze the cache performance of Trips. This tool profiles the program execution at the level of source code and provides performance characteristics for each function in the implementation. Results presented in this section are obtained using the Treebank data set at $\text{minsup} = 45K$. Both TRIPS and TIDES exhibit excellent cache performance with ($L_1$ hit rate, $L_2$ hit rate, $CPI$) of (99.6, 99.94, 0.72) and (99.7, 99.96, 0.7), respectively. Whereas, TreeMiner produced hit rates and CPI of (96.59, 99.96, 1.2).

We observed that TreeMiner has slightly lower $L_1$ hit rate. It also suffers from memory management issues. 49% of TreeMiner’s time is spent in the library, libc-2.3.4.so which contains several memory management routines like `malloc()` and `free()`. TRIPS and TIDES, on the other hand, spent only 2.3% and 3.7% of time in managing the memory. The simple array-based data structures (sequence encodings and embedding lists) employed in our algorithms facilitate efficient memory management and make our algorithms cache-conscious. On the other hand, TreeMiner’s pointer-based dynamic structures make the memory management a difficult task. This problem worsens with decreasing values of $\text{minsup}$ because of the exponential increase in memory size that is occupied by scope lists.

In all the three algorithms, candidate generation is the most expensive step. TreeMiner spent about 75 seconds (22.8% of run time) in scope-list joins (functions `check_ins`, `check Outs`, and `compare`). Both TRIPS and TIDES spent only about 31 seconds (63.7% of execution time) in traversing the embedding list and performing the connectivity checks. High percentage of time spent in user-level code depicts the better CPU utilization by TRIPS and TIDES. Similar results are observed on synthetic data sets, Figure 4.10. For example, CPU utilization of TRIPS (TreeMiner) is 99.3% (90.0%) and 99.3% (72.5%) on $DS_1 - 50K$ and $DS_4 - 1M$, respectively.

As part of our analysis, we also compared our approach against XSpanner [185], using the binary provided by the authors. Please note that the following results are obtained on an Intel P4 based system with a 2.4GHz processor and 4 GB of main memory. While we expected our approaches to outperform XSpanner, we also expected XSpanner to outperform TreeMiner as demonstrated in [185]. The results we obtained are surprising in that XSpanner achieves performance that is typically much worse than that of TreeMiner for many data sets. One possible explanation for this
behavior is the fact that the memory footprint can be very large for such pattern-
growth projection based approaches, potentially resulting in memory thrashing and
large I/O costs [84]. Additionally, XSpanner is likely to suffer from poor cache per-
formance due to the complexity of the pseudo-projection step. For example, on DS1
XSpanner took 1166 seconds whereas TreeMiner took only 38.31 seconds. On DS3,
the mining time of XSpanner and TreeMiner are 245 and 3.1 seconds, respectively.
On all the data sets, TRIPS ran faster than TreeMiner even on this architecture
(DS1 : 24 seconds and DS3 : 1.1 seconds). We observed similar results on both
DS2 and DS4. All the results were obtained from data sets with 50,000 trees and
at minsup = 25. Only on some toy data sets with 5 to 10 trees, the run times of
XSpanner are comparable to TreeMiner. We could not evaluate XSpanner on the two
real data sets because XSpanner binary expects the number of distinct labels to be
less than 10,000. In the Cslogs and Treebank data sets, the maximum label number
is 13,361 and 1,387,266, respectively.

4.5 Conclusions

We proposed algorithms that rely on injective sequential representation of trees to
enumerate the set of all frequent subtrees from a given forest of trees. All candidate
subtrees are generated by traversing the search space of subtrees via simple operations
on sequences. A key innovation here is to seamlessly recast the operations on trees into
those on sequences. Furthermore, the efficiency of support counting phase is improved
by leveraging special array-based embedding list structures that do not house any redundant information. All matches of a given pattern are stored compactly in these lists. We demonstrated that our algorithms and data structures provide significant performance improvements when compared to extant techniques – more than two orders of improvement in both run time and memory performance.
CHAPTER 5
PARALLEL ALGORITHMS

Over the past two decades, processor speeds have increased tremendously whereas the access times of main memory and disk have not kept up with the pace. Memory latency has thus become a pivotal bottleneck to application performance – the classical problem of memory wall [195]. Efforts to bridge this performance gap between CPU and memory subsystem have led to complex architectures with deep memory hierarchies consisting of multiple levels of caches, main memory, and disk. Ineffective use of this deep memory can lead to poor response time [96]. Modern architectural innovations have led to new capabilities in single-core processor systems and high end compute clusters. Examples include hardware prefetching, simultaneous multithreading (SMT), and more recently true chip multiprocessing. Several research studies showed that most data mining and data management algorithms fail to leverage these architectural features of modern computing systems, and as a result, they exhibit suboptimal performance [84, 113, 128].

Architecture-conscious data mining and management is a design philosophy that emphasizes the importance of understanding the capabilities as well as the limitations of underlying computer systems, and subsequently exploiting the specific architectural features to improve the application performance. There has been some success in this regard [26, 49, 128, 157]. In this chapter, we consider the development of such architecture-conscious approaches for mining and indexing tree databases. We specifically focus on techniques targeted at multicore processor systems and distributed shared-nothing compute clusters. We briefly review the relevant literature on this topic.

There has been some notable efforts in database systems towards dealing the problem of memory wall problem. Ailamiki et al [10, 11] demonstrated that poor cache utilization severely hurts the query processing times. Optimizations like blocking, partitioning, loop unrolling and fusion are shown to be effective in improving the
data locality of database operators like joins and aggregations [167]. Several studies [128, 217] have analyzed the performance of database workloads on processors that support SMT. They showed that, while most database memory footprints tend to be large, working sets (when properly organized) can often fit in caches. Improving the cache performance is shown to be vital to appropriately leverage multiple execution threads via SMT. Chen et al improved the performance of index and range searches by leveraging cache prefetching. Prefetching helps in reducing the penalty of long memory access times. Similar techniques have also been applied to Hash-Join operations [49].

Along with these techniques that target several database operators, it is also necessary to make accesses to large data structures employed in these workloads as efficient as possible. Cache-oblivious $B$-trees have been studied by Bender et al [26]. Cache-oblivious algorithms and data structures are designed to exploit the CPU cache without having the size of the cache (or the length of cache lines, etc.) as an explicit parameter [67, 79]. Bender et al have proposed recursive placement and layout strategies for $B$-trees that are cache-oblivious.

Rao and Ross have developed cache-sensitive search trees and cache-sensitive $B^+$-trees [156, 157]. They propose to set the optimal tree node size equal to the natural data transfer size. They also suggest to store all child nodes of a given node in $B^+$-tree contiguously in an array with only a pointer to the first child node. Akin to memory paging in operating systems, individual child nodes are located by adding an offset to the first child pointer. Such a strategy is shown to exhibit good cache line utilization. Chilimbi et al [55] determined that cache prefetching can be used even in the case of dynamic data structures, when appropriately remapped. They describe *ccmalloc* a malloc-based library that allows the user to provide hints to the memory allocator regarding candidates for co-location.

There has been some work done also on architecture-conscious data analysis. Several researchers have characterized the memory performance of various data mining algorithms. Bradford and Fortes [33] presented the performance and memory-access characterization of decision tree induction algorithms, such as C4.5. Parthasarathy et al [147] have studied the memory behavior of Apriori, a popular itemset mining technique. They point out that the data structure used in Apriori exhibit poor data locality, and as a result, it suffers from false sharing in a parallel setting. Ghoting et al [84] presented a detailed characterization of frequent itemset mining algorithms, and they proposed cache-conscious algorithms for novel data structure remapping.
and computation reorganization. Architecture-conscious solutions are shown to be effective in scaling up association rule mining workloads to tera-byte data sets [42].

Several works in the literature have focused on the parallelization of data mining algorithms shared-memory multiprocessor (SMP) systems. Parthasarathy et al have studied several performance issues such as synchronization overhead, degree of parallelism, and data locality [147]. They developed adaptive parallelization strategies that rely on a bitonic workload partitioning scheme to achieve good speedup. Zaki has proposed methods for mining sequence databases on SMP systems [206]. There exist also work that focus on parallelizing other data mining tasks such as classification [209] and clustering [143]. Jin et al have proposed several general-purpose parallelization strategies like full replication and cache-sensitive locking, which are applicable to a wide class of data mining algorithms [106]. While all these studies offer important learnings, they focus on problems that are much simpler than semi-structured data mining. Furthermore, these techniques are not readily applicable for multicore systems.

Buehrer et al. have developed techniques for adaptive state management and adaptive parallelization in graph mining workloads on CMP systems [39]. Since the subgraph isomorphism problem is hard, adaptation of the trade-off between space and run time is the key to efficient performance. In the context of frequent itemset mining, Liu et al have pointed out that existing algorithms developed for single-core processor systems fail to deliver good performance on CMPs due to poor data locality and processor under-utilization [126]. They have proposed new lock-free parallelization mechanisms for improving the parallel efficiency. There has been some efforts in adopting map-reduce style paradigms to multicore systems. Such frameworks, however, are mainly suited for simple data-parallel applications. Most data mining applications, on the other hand, are highly irregular and they require a combination of task-parallel and data-parallel approaches for good parallel efficiency. There is some effort in designing new libraries, programming languages, compilers, and tools for multicore systems like Intel’s TBB [32] and Microsoft’s PCP [151]. They however are not sufficient to solve the problem of interest. For example, Intel’s TBB and related libraries in PCP (designed for Visual Studio) provide new programming interfaces similar to pthreads and OpenMP for easier development of scalable and portable applications. They do not quite support the type of adaptive scheduling strategies that we require to parallelize irregular tree mining workloads. More details on such
architecture-conscious techniques can be found in a recent article by Parthasarathy et al. on next generation data mining [146].

In the first part of this chapter (see Section 5.1), we argue that the existing tree mining algorithms that suffer from poor data locality and large memory footprints are not suitable for multicore systems. We develop a series of memory-conscious optimizations to make the sequential algorithms amenable to these architectures. We then present a multi-level parallel algorithm that demonstrates that moldable and adaptive task partitioning and distribution is of utmost importance to good load balance. We also discuss ideas focusing on parallelizing tree indexing and matching workloads. Based on our algorithms, we design a general-purpose scheduling service for parallelizing data mining and data management applications on multicore chips.

In the latter part of this chapter (see Section 5.2), we discuss parallel tree mining and tree indexing algorithms for distributed compute clusters. We develop a hash-based data partitioning strategy that ensures each partition is a valid representative sample of the entire data set. A stratified sampling mechanism that respects the content as well as structure is also developed. We discuss similar strategies in the context of distributed XML database indexing. Based on these algorithms, we design a hash-based distributed data placement service targeted at large scale data mining and management on compute clusters.

5.1 Algorithms for Multicore Systems

5.1.1 The Multicore Challenge

In multicore or Chip Multi-Processor (CMP) systems, multiple processors operating at low frequencies are packed on a single chip to deliver better performance at smaller power envelopes. Designs range from the general purpose (AMD, Intel) to the specialized (Cell, Sun) to the niche markets (GPUs). Although current designs have 4-8 cores, Intel’s 2015 Processor road map proposes CMPs with hundreds of cores [63]. We now discuss the challenges in deploying large scale data mining and data management algorithms on these modern architectures.

Memory bandwidth: The disparity of speed between CPU and memory outside the CPU chip has been growing constantly over the last two decades. Therefore, the memory latency has been one of the primary bottleneck for efficient computer performance. Techniques like caching, data prefetching, relaxed memory consistency
have been proposed to hide the impact of memory latency from end applications. Multicore systems aggravates this classic memory-wall problem as they further widen the gap between processor and memory performance. For these architectures, along with memory latency one must also develop mechanisms that reduce the memory bandwidth i.e., off-chip traffic. Since all cores in a CMP system share the same front side bus, bandwidth to main memory is likely to be a precious shared commodity. A recent research report [135] from Sandia National Laboratories has shown that the limited memory bandwidth in multicore systems degrades the performance of data-intensive informatics applications. The report argues that a big boost in memory bandwidth must be required to realize scalable applications.

The control of algorithm’s memory footprint is an important consideration in this context. Excessive memory usage may force OS to rely on virtual memory thereby slowing down the application. It may also increase the bus contention and consequently the number of stalls – likely to be aggravated on CMPs since all cores share a common bus to the main memory. Furthermore, memory management activities like allocation, de-allocation, and sharing data or meta-data among processors becomes extremely difficult. Therefore, restricting the memory usage of mining algorithms is very critical in achieving good performance on CMPs.

**Locality of reference:** Multi-core machines typically have shared caches, though other variants are possible [117, 131]. Since a sizable portion of the chip’s real estate is occupied by the cores themselves, the amount of space available for on-chip caches is fairly limited. Algorithms with poor locality likely to exhibit heavy off-chip traffic amplifying the pressure on the (shared) front side bus. Poor locality along with excessive memory usage can result in **thrashing**.

It is, however, non-trivial to devise techniques that improve the locality of large scale data mining algorithms. As most of these algorithms are highly irregular in nature, it is hard to predict the access patterns and to optimize for them. Moreover, while the spatial locality is deterred by pointer-based data structures, the temporal locality is hindered by huge search space.

**Working set size:** The size of a working set is defined as the amount of data that is actively used by the program during a particular phase of computation. If the working set of an algorithm is not effectively held in the cache the the off-chip traffic increases due to constant data swapping between cache and memory. Since the amount of cache available per core is small in CMP systems, algorithms that maintain small-sized working sets must be developed for efficient performance. Note
that each core that is used in the computation maintains its own separate working set. Therefore, the sum of all per-core working set sizes must be smaller than the amount of cache available.

**Load balance:** Efficient parallelization techniques that achieve good load balance are of great importance in the context of CMP systems. Paramount to leveraging the additional compute capability is an effective task partitioning mechanism to distribute the work among individual processing elements. This can be particularly challenging in the presence of data and control dependencies. Moreover, the data and parameter dependent aspects of data mining workloads makes estimating the lifetime of a task difficult. In order to achieve good scalability for irregular data mining algorithms one needs to devise strategies which can: (i) adaptively modulate the type and granularity of the work being shared; and (ii) efficiently handle the skew present in the data set.

Recent research has shown that algorithms developed for multicore systems must expose and subsequently exploit fine-grain parallelism [160]. The time spent in task creation and task distribution must therefore be very small. The tasks themselves must be small and independent. The dependency among different tasks can be reduced by tagging each task with required state information (i.e., meta-data), if any. This however results in replication of meta-data, bulky tasks, and poor efficiency since a lot of time is spent in creating and distributing of these tasks. Therefore, sequential algorithms that limit the use of meta-data structures play an important role in this context.

**Dynamic data structures:** Since the algorithms developed for multicore systems operate in shared-memory environments, the use of dynamic data structures must be limited. The system calls to dynamic memory allocation and de-allocation on the heap are serialized. Excessive use of dynamic data structures thus hinders the parallel performance.

### 5.1.2 Parallel Tree Mining

**Limitations of Existing Tree Mining Algorithms**

Most of the existing tree mining algorithms leverage embedding lists as an attempt to trade space for improved execution time. However, in practice, these lists can grow arbitrarily large in size. And, we contend that these large lists not only affect the memory performance but also run time performance. This behavior can easily be observed by performing a worst case analysis on the tree size. For the purpose of this
Table 5.1: Characterization of Tree Mining Algorithms

<table>
<thead>
<tr>
<th>Working set¹ (KB)</th>
<th>TreeMiner</th>
<th>iMB3-T</th>
<th>Trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory usage² (GB)</td>
<td>7</td>
<td>32</td>
<td>4</td>
</tr>
</tbody>
</table>

¹On Treebank data set at minsup=45K (85%) – see empirical evaluation, later in this section
²Maximum memory footprint observed in all our experiments later in this section

Discussion, let us consider the embedding lists used in Trips algorithm – see Figure 4.4. It is important to note that when a particular pattern \( S \) is being processed in Trips, the embedding list contains the matches for all those patterns whose equivalence class contains the current pattern \( S \). This is due to the pattern-growth approach employed in Trips.

Now, consider a worst case scenario of a chain tree (a path) \( T \) of size \( n \), where every node has the same label (say, \( A \)). When a single node pattern \( S = A \) is being processed, the embedding list holds exactly \( \binom{n}{1} = n \) entries where each entry corresponds to a match of \( S \) in \( T \). This single node pattern is then extended to an edge pattern \( S = A - A \). The corresponding list must then record information about all \( \binom{n}{2} \) different matches of the edge pattern i.e., the list holds a total of \( \binom{n}{1} + \binom{n}{2} = n(n+1)/2 \) different entries. Similarly, when \( S \) is grown into a pattern of size \( n \) (i.e., the complete path), the number of entries in the list is equal to \( \sum_{i=1}^{n} \binom{n}{i} = 2^n - 1 \). At this point, the list holds the information about all matches for all patterns of all size. From this discussion, it can be seen the worst case size of an embedding list is exponential in tree size.

We observe that such extreme cases often occur in real-world data sets. For example in Cslogs data set [207], when a 3-node pattern is extended with a single edge, the number of matches sharply increased from 141,574 to 2,337,127. In terms of physical memory, the increase in the number of matches is amounted to an increase in the size of embedding list from 1.2MB to 19.02MB. When the same pattern is extended to a subtree with 6 nodes, the number of matches increased to 474 million. All state-of-the-art algorithms diligently store the location of each of those 474 million matches, thereby resulting in very high memory usage.

Table 5.1 presents a detailed characterization of memory performance of existing algorithms. Large embedding lists maintained in these algorithms lead to several
limitations—(i) they increase the memory footprints, and hence make it impossible to realize efficient performance when the algorithms are executed on multicore chips (see Section 5.1.1); (ii) since embedding lists are dynamic data structures, they complicate the memory management, especially when implemented in a shared-memory environment; and (iii) they severely limit the parallelization of algorithms as they lead to extra dependencies among different tasks.

Memory Conscious Optimizations

![Diagram of Memory Conscious Tree Mining Algorithm](image)

Figure 5.1: Overview of Memory Conscious Tree Mining Algorithm

We now present our memory-conscious optimizations that aim to reduce the memory footprint without having a significant impact on the run time performance. Figure 5.1 gives an overview of our Memory Conscious Trips (MCT) algorithms. Given database $D$ of trees is first transformed into sequences ($T(D)$). Simultaneously, the frequency of each node label in the database is counted. The transformed database is then pruned by removing infrequent nodes to produce $T'(D)$. Both $T'(D)$ and the set of frequent nodes $F1$ are fed to the mining block that produces all frequent subtrees. Processing in the mining block can be divided into three phases: on-the-fly embedding lists $OEL$, candidate generation $CG$, and support counting $SC$. Unlike in existing algorithms, MCT does not explicitly maintain any embedding lists. It instead computes the frequent subtrees by performing $CG$ and $OEL$ simultaneously. Generated
frequent patterns are fed back to the mining block to produce larger patterns. We
now describe our optimization in more detail.

**Tree Pruning and Recoding (PRUNE)**

This optimization avoids superfluous computations by eliminating those parts of the
tree that do not help in finding frequent subtrees. It draws inspiration from a well-
known technique in itemset mining [92]. Given database trees are pruned by removing
all infrequent nodes, and subsequently, the remaining node labels are recoded for com-
putational convenience. While the pruning step alters the numbered Prüfer sequence,
recoding changes the label sequence (see Algorithm 10). Once the size of pruned tree
is computed in Lines 4-8, labels are recoded by scanning the tree from root to leaves.
The *map* array maintains the list of nodes that are not pruned. Whenever an in-
frequent node is encountered, it is replaced by a closest ancestor node that was not
pruned (Line 21). Top-down processing guarantees that a node is processed only af-
after all of its ancestors are processed, and hence it is easy to find ancestor nodes that
are not pruned. The resulting *map* array is used to build new NPS (Lines 17-18).
Note that the root nodes are never pruned in the entire processing (Lines 13-15),
only infrequent non-root nodes are removed from the database. The computational
complexity of both pruning and recoding steps is $O(n)$, where $n$ is the tree size.

**On-the-fly Embedding Lists (NOEM)**

![Example Database Tree](image1)

![Pattern and its R-matrix](image2)

Figure 5.2: Subtree matching example
Algorithm 10 Tree pruning and recoding algorithm

Input: Database $D$

1: Recoded label set $newLabels \leftarrow \phi$
2: for each tree $T$ in $D$ do
3: \hspace{1em} $n \leftarrow$ size of $T$; $T_{\text{recoded}} \leftarrow \text{null}$; $count \leftarrow 0$
4: \hspace{1em} for each node $v \in T$ do
5: \hspace{2em} if $v$ is the root or $v$ is frequent then
6: \hspace{3em} increment $count$
7: \hspace{2em} end if
8: \hspace{1em} end for
9: \hspace{1em} $ind \leftarrow count - 1$
10: \hspace{1em} for $i$ from $n$ to 1 do
11: \hspace{2em} \hspace{1em} $v \leftarrow \text{$i^{th}$ node in the Prüfer sequence of } T$
12: \hspace{2em} \hspace{1em} if $v$ is root then
13: \hspace{3em} \hspace{1em} add $v$ to $newLabels$, if not present
14: \hspace{3em} \hspace{1em} add $v$ to $T_{\text{recoded}}$ with recoded label
15: \hspace{3em} \hspace{1em} $map[n] \leftarrow count$
16: \hspace{2em} \hspace{1em} else if $v$ is frequent then
17: \hspace{3em} \hspace{2em} $u \leftarrow (newLabels(v.label), map[v.parent])$
18: \hspace{3em} \hspace{2em} add $u$ to $T_{\text{recoded}}$
19: \hspace{3em} \hspace{2em} $map[i] \leftarrow ind$; \hspace{1em} $ind \leftarrow ind - 1$
20: \hspace{2em} \hspace{1em} else
21: \hspace{3em} \hspace{2em} $map[i] \leftarrow map[v.parent]$
22: \hspace{2em} \hspace{1em} end if
23: \hspace{1em} end for
24: end for
Our MCT algorithm, instead of storing embedding lists (EL) explicitly, adopts a strategy that dynamically constructs the list, uses it, and then de-allocates it. In graph-theoretic terms, constructing a dynamic EL is equivalent of finding the set of all (embedded) subtree isomorphisms of a given pattern in the database—a core problem in XML indexing. We construct EL on demand by employing a dynamic programming based approach that takes inspiration from subtree matching algorithm in XML indexing (see Section 2.2.2). There are however some important differences—(i) in XML indexing, there is no notion of embedding lists which are employed to save time on repeated subtree isomorphisms, (ii) each mining run here comprises of many tree matching queries, and our subsequent optimizations. Unlike in XML indexing, as we show later in empirical evaluation, a straight application of these techniques in fact increases the execution time. We devise techniques reorganizes the computation to improve data locality and run time performance. Note that dynamic list construction affects only the lines 3-6 of Algorithm 9—correctness of the algorithm is still intact.

Let $S=(LS_S, NPS_S)$ be a subtree whose matchings need to be found in database tree $T=(LS_T, NPS_T)$. Assume that $|S| = m$ and $|T| = n$. We now describe the method in which the matches for a subtree $S$ are found in the database tree $T$. It is similar to the algorithm presented in Section 2.2.2. We briefly highlight the important aspects of the algorithm here.

Prüfer sequences, due to the way they are constructed, possess an important property that if $S$ is an embedded subtree of $T$ then the label sequence $LS_S$ is a subsequence of $LS_T$ (see Theorem 2.2.1). In other words, being a subsequence is a necessary but not sufficient condition for subtree isomorphism.

First, we check if $LS_S$ is a subsequence of $LS_T$ or not by computing the length of their longest common subsequence (LCS) using a traditional dynamic programming approach [184] (see Algorithm 11). The algorithm constructs a matrix $R$ using Equation 5.1.1. The bottom-right corner entry $R[m, n]$ in the matrix gives the length of LCS between $LS_T$ and $LS_P$. If $R[m, n] \neq m$ then we conclude that $S$ is not a subtree of $T$.

$$R[i, j] = \begin{cases} 0, & \text{if } i = 0, j = 0 \\ R[i - 1, j - 1] + 1, & \text{if } LS_S[i] = LS_T[j] \\ \max(R[i - 1, j], R[i, j - 1]), & \text{if } LS_S[i] \neq LS_T[j] \end{cases} \quad (5.1.1)$$

Second, if $LS_S$ is a subsequence of $LS_T$ then we enumerate all subsequence matches of $LS_S$ in $LS_T$ by backtracking from $R[m, n]$ to $R[1, 1]$ (lines 10-16 in Algorithm 11).
Algorithm 11 On-the-fly embedding list construction

Input: $P = (LS_P, NPS_P), T = (LS_T, NPS_T)$

$R \leftarrow \text{computeLcsMatrix}(LS_P, LS_T)$;

say $m \leftarrow |LS_P|, n \leftarrow |LS_T|$

if $R[m][n] \neq m$ then

return \{Subsequence check failed, $P$ is not a subtree of $T$\}

else

processR $(m, n, 0)$

end if

processR $(p_i, t_j, L)$

1: if $p_i=0$ or $t_j=0$ then
2: return \{terminating condition for recursion\}
3: end if
4: if $L = m$ then
5: if SM[..] corresponds to a subtree then
6: update $EMList[T]$ with $SM$ \{found a structure match\}
7: end if
8: return
9: end if
10: if $LS_P[p_i] = LS_T[t_j]$ then
11: $SM[m - L] \leftarrow t_j$
12: processR $(p_i - 1, t_j - 1, L + 1)$
13: processR $(p_i, t_j - 1, L)$
14: else if $R[p_i, t_j - 1] < R[p_i - 1, t_j]$ then
15: processR $(p_i, t_j - 1, L)$
16: end if
A subsequence match SM is denoted by \((i_1, \ldots, i_m)\), where \(i_k\)'s are the locations in \(T\) at which the match occurs i.e., \(LS_P[k] = LS_T[i_k]\) for \(1 \leq k \leq m\). It is worth noting that, unlike in classical sequence matching problem, here we are interested in obtaining all matches. Since backtracking is performed in backwards, the matches are generated from right-to-left. Since subsequence matches are enumerated by considering only the label sequence, there may exist some false positives. They are pruned in the next step by taking the tree structure into account.

Third, we filter the false positive subsequences by matching the structure (given by \(NPS\)) of \(SM = (i_1, \ldots, i_m)\) with that of \(S\) (Line 5 in Algorithm 11). Such a structural match \((map)\) maps every parent-child relation in \(S\) into an ancestor-descendant relation in \(SM\) i.e., in \(T\) (see Algorithm 3). We first set \(map[m] = i_m\) (root node). For \(k = m-1 \ldots 1\), we check if \(map[NPS_S[k]]\) is either equal to \(NPS_T[i_k]\) or is a nearest mapped ancestor of \(NPS_T[i_k]\) i.e., parent of \(k^{th}\) node in \(S\) is mapped to an ancestor of \(i_k^{th}\) node in \(T\). Since nodes are considered in reverse post order, structure match is also established from right-to-left (i.e., root-to-leaf). Resulting match is finally added to the dynamically constructed embedding list (Line 6 in Algorithm 11).

**Example:** Figure 5.2 shows the \(R\)-matrix constructed from the database tree \(T\) and a 3-node pattern \(P\). The length of LCS between two label sequences \(LS_P\) and \(LS_T\) is given by the bottom right corner entry, whose value is 3. Since this value is equal to the \(|LS_P|\), we enumerate all six subsequence matches of \(LS_P\) in \(LS_T\). Out of these six matches, only \(M1, M2,\) and \(M4\) are actual subtree matches. For \(M3\): at \(k=3\), the root node is mapped to node \(i_3 = 9\) in \(T\) i.e., \(map[3] = 9\). At \(k=2\) \((NPS_S[k]=3)\), we set \(map[2] = i_k = 2\) because \(map[3] = NPS_T[i_k]\). However at \(k=1\) \((i_k = 1)\), \(map[3] \neq NPS_T[i_k]\) and \(map[3]\) is not the nearest mapped ancestor of \(i_1\) in \(T\). Since the check fails, \(M3\) is declared as a false positive. For \(M5\) and \(M6\), the check fails at \(k=1\) and \(k=2\), respectively.

**Tree Matching Optimizations**

The following three optimizations reduce the amount of redundant computations in Algorithm 11. The first two reduce the recursion overhead incurred while backtracking whereas the third one reduces the overhead due to false positives.

1) **Label Filtering (LF):** Before constructing the \(R\)-matrix, we remove those nodes in \(T\) which do not appear in \(S\). In Figure 5.2b, the columns corresponding to nodes \(D, E,\) and \(F\) can be safely deleted as they do not help in establishing the subsequence match.

130
2) **Dominant Match Processing (DOM):** Observe that a subsequence match is established only at the entries (called as *dominant matches*) where both \(LS_S\) and \(LS_T\) match (condition 2 in Equation 5.1.1). Backtracking on rest of the entries is redundant and must be avoided. In Figure 5.2b, dominant matches are encircled. For example, \(R[2,6]\) and \(R[1,3]\) are dominant and all the other shaded cells simply carry LCS value from one to the other. Recursion from \(R[2,6]\) can directly jump to \(R[1,3]\) avoiding all the other shaded cells.

3) **Simultaneous Matching (SIMUL):** Here, we leverage the fact the both subsequence and structure matching phases operate from right-to-left in reverse post order. Therefore, instead of performing the structure matching after generating all subsequence matches, we can do both the matchings simultaneously. As soon as a subsequence match is established at position \(k\), we perform the structure match at that position. Such an embedding of structural constraints into subsequence matching detects the false positives as early as possible and never generates them completely.

**Computation Chunking (CHUNK)**

Since the size of EL is proportional to the number of matches, the dynamic embedding lists can grow exponentially, in the worst case. This optimization completely eliminates the lists by coalescing both tree matching and tree mining algorithms. It operates in three steps: loop inversion, quick checking, and chunking. The computation in Algorithm 9 is reorganized by inverting the loops in lines 9-10 i.e., \(T\) is scanned for each match \(m\) instead of processing \(m\) for each node in \(T\). The second step *Quick checking* notes that the extensions associated with two different matches \(m_i\) and \(m_j\) \((i < j)\) are independent of each other. Thus, \(m_i\) need not wait till \(m_j\) is generated and thus it *need not be stored* explicitly in EL. Finally, *chunking* improves the locality by grouping a fixed number of matches into chunks. The tree \(T\) is then scanned for each chunk instead of for each match \(m\). Once the extensions against all the matches in one chunk are found, we proceed to the next chunk. This optimization implicitly leverages all the other optimizations described above. In our empirical study, we define chunks to contain 10 matches.

The complete Memory Conscious Trips (MCT) is shown as Algorithm 12. Since it always keeps a fixed number of matches in memory, MCT maintains a *constant-sized* memory footprint throughout the execution. Further, chunking *localizes* the computation to higher level caches, improving both locality and working sets.

**Complexity analysis:** Like other pattern mining algorithms, MCT belongs to
Algorithm 12 Memory Conscious Trips (MCT)

\textbf{mineTrees} \( \text{pat, extension } e, \text{ tidlist} \)

\begin{itemize}
  \item A: \textbf{for each} \( T \) in tidlist \textbf{do}
  \item B: \hspace{1em} \text{construct} \( R \)-Matrix for \( T \) and \textit{newpat}
  \item C: \hspace{1em} \text{processR} \( (m, n, m) \)
  \item D: \textbf{for each} \( ext \) in \( H \) \textbf{do}
  \item E: \hspace{1em} \text{mineTrees} \( \textit{newpat, ext} \) recursively
\end{itemize}

\textbf{processR} \( (p_i, t_j, L) \)

\begin{enumerate}
  \item \textbf{if} \( p_i = 0 \) or \( t_j = 0 \) \textbf{then} \textbf{return}
  \item \textbf{if} \( L = 0 \) \textbf{then}
  \item \hspace{1em} \text{add} \( SM \) \textbf{to} \( EMList \) \textbf{and add} \( T \) \textbf{to} \( \textit{newtidlist} \)
  \item \textbf{if} \( |EMList| \% 10 = 0 \) \textbf{then}
  \item \hspace{1em} \textbf{for each} \ match \( m \) \textbf{in} \( EMList \) \textbf{do}
  \item \hspace{2em} \textbf{for each} \ node \( v \) \textbf{in} \( T \) \textbf{do}
  \item \hspace{3em} \textbf{if} \( v \) \textbf{is a valid extension with} \( m \) \textbf{then}
  \item \hspace{4em} \text{add the resulting extension to} \( H \)
  \item \hspace{1em} \text{\( EMList \leftarrow null \)}
  \item \textbf{return}
  \item \textbf{for} \( k = t_j \) \textbf{to} \ 1 \textbf{do}
  \item \hspace{1em} \textbf{if} \( R[p_i][k] \) \textbf{is dominant} \& \( R[p_i][k]=L \) \textbf{then}
  \item \hspace{2em} \( SM[k] \leftarrow (LS_T[t_j],NPS_T[t_j]) \)
  \item \hspace{1em} \textbf{if} \( \text{agreeOnStructure} \ (P, SM, k) \) \textbf{then}
  \item \hspace{2em} \text{processR} \( (p_i - 1, t_j - 1, L - 1) \)
\end{enumerate}
$\#P$ complexity class as it has to count and enumerate all frequent subtrees. mineTrees in Algorithm 12 is invoked exactly once for every frequent pattern ($pat+e$) that is discovered. For a given $S$ and $T$ (of sizes $m$ and $n$), the maximum number of recursions on processR ($c_{m,n}$) can be approximated as follows (see Section 2.2.4):

$$c_{m,n} = \begin{cases} 
1 + \sum_{i=1}^{n-m+1} (1 + c_{m-1,n-1}), & \text{if } n > m \\
n, & \text{if } n = m \lor m = 1
\end{cases}$$

(5.1.2)

c$_{m,n}$ has a closed form of $\frac{n+1}{n-m+1}$. The branch conditions in lines 12 and 14 take constant time and the run time of lines 2-10 is governed by the number of matches for $S$ in $T$.

**Extensions**

An important advantage of our algorithms is that they can be easily modified to mine different types of subtrees. We now briefly mention about those modifications.

**Induced subtrees**: Induced subtrees, a small subset of embedded subtrees, maintain parent-child relationships among nodes. They can be mined by making a simple modification to our algorithm that generates the new candidate subtrees. Recall from Section 4.3.2 that the new candidate subtrees are generated by performing the connectivity checks for all the ancestors of a given node until the one that is part of a given embedding is found. While mining induced subtrees, this procedure needs to be restricted to parent nodes i.e., connectivity checks need to be performed only for parent nodes. Therefore, a particular node is considered as an extension point if only if its parent is part of the given embedding. However, this modification alone is not sufficient if our memory-conscious optimizations are in place. The optimization tree pruning and recoding attaches the children of pruned nodes to a nearest ancestor that is not pruned. The children of a node thus may actually correspond to its descendants. Therefore, we need to distinguish between induced children and embedded children. In other words, each edge in the pruned tree has to be annotated to indicate whether it represents a parent-child or ancestor-descendant relation in the original database tree. We leverage the property of our sequencing methods that each entry in CPS or DFS corresponds to an edge in the tree. We thus maintain another sequence induced, along with LS and NPS, that is populated while the tree is pruned and recoded. While generating extensions, we only consider those nodes which are induced children and are also on the left (or right) of embedding. Furthermore, our structure matching
algorithm in Section 5.1.2 needs to be modified so that the structure agreement checks look at only parent nodes but not ancestors. More precisely, check at position $k$ succeeds only if $LS_P[k] = LS_T[i_k]$ and $strMap[NPS_P[k]] = NPS_T[i_k]$.

**Unlabeled subtrees:** It is fairly simple to modify our algorithms to mine for unlabeled subtrees. A straightforward approach is to label every node in the data set with the same label and apply TRIPS and TIDES without any modifications. Instead, only can modify those algorithms to ignore edge labels. For example, a structure agreement check at position $k$ does not check for the equality between $LS_P[k]$ and $LS_T[i_k]$. A potential problem when the nodes are unlabeled is that the number of false positive subsequence matches can be really high. However, our simultaneous subsequence and structure matching detects such false positives in early stages. Therefore the impact on the overall performance is minimal.

**Edge-labeled subtrees:** If the data set have edge labels, then our sequences have to be updated so that each edge is annotated with its label. This again is similar to induced sequence that is used while mining for induced subtrees. A new sequence has to be added to CPS and DFS to indicate the edge labels. In addition, wherever the node labels are considered in the algorithm, the edge labels also have to be considered. Note that, this change in the representation is minimal.

**Unordered subtrees:** The only change that is required while mining for unordered subtrees is in Algorithm 9 at line 9, where $T$ is scanned for extension points. TRIPS, for example, finds extensions by evaluating the descendants of the left-most child since it considers only the nodes which are on the left of a given embedding. While mining for unordered subtrees, connectivity checks have to be evaluated for descendants of all leaf nodes (not just the left-most node). We thus need to consider all the nodes which are on the left of a node that is matched to the root of the pattern. In order to guarantee non-redundant candidate generation i.e., to avoid the generation of duplicates, a strict total order must be enforced on the set of labels, e.g., lexicographic ordering.

**Adaptive Parallelization**

We now consider the parallelization of MCT for multicore systems. Note that directly parallelizing Trips algorithm is the first approach we considered. However, embedding lists led to a large memory footprint resulting in significant contention overhead and pressure on the front-side bus. The inherent dependency structure of lists pose difficulties in sharing them, leading to a coarse grained work partitioning and poor
load balance (see Section 5.1.2). Essentially, parallelization without identifying the memory optimizations, presented in the previous section, is extremely inefficient.

Our parallel framework employs a multi-level work sharing approach that adaptively modulates the type and granularity of the work that is being shared among threads. Each core $C_i$ in the CMP system runs a single instantiation (i.e., a thread) of our parallel algorithm. Henceforth, the terms core, thread, and process are used interchangeably, and are referred by $C_i$. A job refers to a piece of work that is executed by any thread. The set of all threads consume jobs from a job pool (JP) and possibly produce new jobs into it. The jobs from a job pool are dequeued and executed by threads on a “first come first serve” basis.

**Control flow:** As pointed out by Leung et al. [120], if the threads are allowed to share the work asynchronously then detecting a global termination would be non-trivial – since the jobs could be shared while a termination detection algorithm is being executed. Instead, we implement a simple lock-based algorithm that is driven by the amount of remaining work in the system. Whenever a thread $C_i$ finds the job pool to be empty, it votes for termination by joining the thread pool (TP), and detaches itself (i.e., blocks itself) from execution. Each thread monitors TP at pre-set points during its run time, and if it is not empty then it may choose to fork off new jobs onto JP, and notify the threads waiting in TP. The mining process terminates when all threads vote for termination. We implemented TP using simple locks (akin
to semaphores) and condition variables. Similar strategy can be used when multiple job pools are maintained based on thread groups (e.g., distributed and hierarchical job pools) – job pools here act as implicit channels for communication between running and waiting threads.

In our multi-level approach, threads operate in three different levels. Each level corresponds to a different execution mode, which dictates the type and granularity of the jobs in that mode. The three execution modes are task-parallel, data-parallel, and chunk-parallel. The first one exploits the parallelism across different portions of the search space. The data-parallel mode parallelizes the work required to mine a single pattern. Finally at the finest level of granularity, the chunk-parallel mode obtains the matches of a pattern within a single tree in parallel. For a simpler design, we used different job pools for different modes: task pool ($JP_T$), tree pool ($JP_D$), and column pool ($JP_C$), respectively. Alternatively, one can implement a single job pool with prioritized jobs. Shared access to these pools is protected using simple locks. Jobs in these job pools are uniquely identified by job descriptors. Each job descriptor $J$ is a 6-tuple as shown below.

$$J = (J.t, J.i, J.f, J.c, J.o, J.r)$$

$$J.t = \begin{cases} 
\text{task}, & \text{if } J \in JP_T \\
\text{data}, & \text{if } J \in JP_D \\
\text{chunk}, & \text{if } J \in JP_C 
\end{cases}$$

Job type $J.t$ corresponds to the execution mode, and it defines the remaining entries. Given $J$, a thread starts with the inputs $J.i$, applies the function $J.f$ to produce an output $J.o$. The control is then returned to the job that created $J$ if return flag $J.r$ is set to true. A condition $J.c$ is evaluated at pre-set points to determine whether or not to spawn new jobs from $J$.

$J.t$ also determines the type of new jobs which $J$ can spawn. A task-level job can either create new tasks or a single job of type data. A chunk-level job in $JP_C$ can only be created by a data-parallel job in $JP_D$. And, jobs in $JP_C$ can not create new jobs i.e., $\forall J \in JP_C, J.c = false$. The granularity of jobs in $JP_T$ is more than that in $JP_D$, which in turn is greater than the granularity of jobs in $JP_C$. We integrate different execution modes and termination detection as shown in Algorithm 13. Such a design adaptively adjusts the granularity by switching between the execution modes.
Algorithm 13 Parallel Tree Mining

1: initialize( ) // I1
2: identifyGranularities( ) // I3, I4, I5
3: while true do
4:   if JP\textsubscript{T} is empty then
5:     if JP\textsubscript{D} is empty then
6:       if JP\textsubscript{C} is empty then
7:         vote for termination
8:       else
9:         if { all threads voted } break
10:     else
11:       process JP\textsubscript{C} // chunk-parallel (I8, sync)
12:     else
13:       process JP\textsubscript{D} // data-parallel (I8, sync, I9)
14:   else
15:     mine a task from JP\textsubscript{T} // task-parallel (I8, I9)
16: finishUp( ) // I2

Task-parallel mode
In this mode, each thread processes jobs from the task pool JP\textsubscript{T} where each task corresponds to the process of mining full or a portion of an equivalence class [S]. Therefore, every job J ∈ JP\textsubscript{T} is associated with a subtree J.i=S. The output J.o is the set of subtrees produced from S by invoking J.f (mineTrees in Algorithm 12). Further, J.r is always set to false in this mode.

Each strategy in this mode differs in the way the search space is partitioned into tasks. A naive strategy is to partition the space by equivalence classes – EQ in Figure 5.3, and schedule different classes (F1 in Algorithm 9) on different cores. More precisely,

\[ JP_T = \{J | J.i \text{ is a seed pattern } \land J.c = false \} \]

Since J.c is set to false, each job is processed till its completion to produce all subtrees from the equivalence class of seed pattern J.i. Such a coarse grained strategy, which is referred to as **Equivalence class task partitioning (EqP)**, likely to perform poorly because most real-world data sets are highly skewed and the variance in |J.o|’s is usually high.

Another strategy is to partition the search space such that each pattern is treated as a different job – P in Figure 5.3. Each extension that is produced is enqueued into
the job pool as new tasks (i.e., \( J.c \) is a tautology). Such a technique is referred to as **Pattern-level task partitioning** (PaP). It can be formally denoted as:

\[
JP_T = \{ J \mid J.i \text{ is a frequent subtree } \land J.c = \text{true} \}
\]

Here, \( JP_T \) is initialized with frequent nodes from \( F_1 \). If \( |F_1| < |C| \) then it is initialized with frequent edges. One can continue to mine in levels until \( |JP_T| \) is sufficiently greater than \( |C| \). For better efficiency, the projected database of the subtree is also included in \( J.i \). This strategy suffers from locality issues since the subtrees may not be mined at the place they were created. Also, aggressive job sharing often results in memory management and computation overheads, motivating the need for an adaptive approach.

In an **adaptive task partitioning** (AdP) strategy, the search space is partitioned *on demand*. New tasks are created only when there are idle threads waiting (for work) in the thread pool \( TP \). Unlike EqP and PaP, this method *adaptively modulates* the task granularity at run time. It can be described as:

\[
JP_T = \left\{ J \mid J.i = \text{a frequent subtree } \land \begin{array}{l}
J.c = (TP \neq \Phi \land |Ext| \geq 1) \\
\end{array} \right\}
\]

\( |Ext| \) is the number of extensions that are *yet to be* processed. Note that, \( TP \neq \Phi \) implies that the job pool is empty i.e., new jobs are created only if the job pool is empty and some threads are in wait state. Instead, one can choose to spawn new jobs when the size of the job pool falls below a pre-defined threshold value. The spawning condition in this strategy is evaluated before processing each extension, between lines D-E of Algorithm 12. Since it dynamically modulates the task granularity, it not only achieves good load balance but also exhibits good locality since extensions are mined, whenever possible, on the processor that created them.

**Data-parallel mode**

The task partitioning strategies primarily process the search space, in parallel. They do not take the underlying *data distribution* into account. For example in case of a website, one access pattern \( P_1 \) can be more dominant and popular than another pattern \( P_2 \). Task-parallel strategies *can not* exploit this difference as they implicitly assume that all patterns are of similar complexity. Efficiency can be improved by *dividing* the work associated with the popular i.e., more expensive pattern \( P_1 \).

We parallelize the job of mining a single subtree \( S \) by looking at its projected database \( PD_S \), 8-12 in Algorithm 9 (PD in Figure 5.3). We treat each tree in \( PD_S \)
as a different job, and schedule them on to different cores. The pool of database trees $JP_D$ can be denoted as:

$$JP_D = \{ J \mid J.i = T : T \in PD_S \land J.c = false \land J.r = true \}$$

Note that all jobs in $JP_D$ (unlike $JP_T$) are defined in the context of a subtree ($S$) that is currently being mined. The trees in $PD_S$ are processed simultaneously by multiple cores. Each core produces a subset of extensions, which are then combined to produce a final set of extensions for $S - J.r$ is set to true.

We devise an adaptive strategy by combining this basic method that takes the data distribution into consideration with the best task partitioning strategy AdP. It is called as **Hybrid work Partitioning (HyP)**. Here, a core $C_i$ that is currently mining a task-level job $J \in JP_T$ with $J.i=S$ forks off new jobs on to $JP_D$ only when it finds any idle threads while finding extensions from $S$. Once all trees in $JP_D$ are processed, the core $C_i$ performs a reduction operation to combine the partial sets of extensions. If needed, $J$ may now proceed to create new tasks according to AdP. Therefore, a task-level job may either create new tasks or new jobs of type data – spawning condition thus needs to be augmented as follows.

$$\forall J \in JP_T, \quad J.c = \begin{cases} 
\text{add tasks to } JP_T, & \text{if } TP \neq \Phi \land |Ext| \geq 1 \\
\text{add jobs to } JP_D, & \text{if } TP \neq \Phi \land \frac{c(J,i)}{s(J,i)} < \theta 
\end{cases}$$

While the first condition is evaluated between lines D-E of Algorithm 12 (same as AdP), the second one is checked between lines A-B. The second condition governs the creation of data-parallel jobs and it depends on the amount of work that is remaining to complete the task $J.i$. A rough estimate for the amount of remaining work is given by $\frac{c(J,i)}{s(J,i)}$, where $c(J,i)$ is the number of matches found so far and $s(J,i)$ is the support of $J.i$ (known from line 14 in Algorithm 9). If this ratio is smaller than a threshold $\theta$ (we use $\theta = 20\%$ in our evaluation) then it means that there is a lot of work to be done, and can be shared with others. Such a method essentially decides whether it is worth dividing the work into jobs of finer granularity.

Once the tree pool is created, we sort the trees in the decreasing order of their size. This is similar to classical job scheduling where the jobs are sorted in the decreasing order of their processing time. We sort based on tree size because the mining time that depends on the number of matches in a given tree is likely to be proportional to the tree size.
Chunk-parallel mode

Even the hybrid strategy HyP may not always achieve full efficiency in practice. This is because the trees themselves can be skewed. For example in Bioinformatics, one Glycan or RNA structure may be very large when compared to the other. Such large trees and the trees with large number of matches will introduce load imbalance while using HyP. To deal with such a skew, the job of mining a single tree i.e., the process of finding matches and corresponding extensions from a given tree should be parallelized. This fine grain parallelism is obtained by parallelizing at the level of chunks, which are generated in lines 3-4 of Algorithm 12. Since chunks are created from individual columns of the \( R \)-matrix, we treat each column as a separate job and schedule them on to different cores.

This mode is entered only when all the available parallelism in data-parallel mode is fully exploited. A job of type data in \( JP_D \) switches to this mode based on the following condition:

\[
\forall J \in JP_D, J.c = \begin{cases} 
\text{spawn jobs onto } JP_C, & \text{if } TP \neq \Phi 
\end{cases}
\]

One can also design \( J.c \) based on pattern size, number of matches found so far, and the portion of \( R \)-matrix that is yet to be explored. This condition is evaluated between lines 13-14 of Algorithm 12.

For each job \( J \) in column pool \( JP_C \), the input is a column from \( R \)-matrix, and the partial match that is constructed so far (by \( J \)'s parent job in \( JP_D \)). \( J.f \) backtracks from the input column to discover the remaining part of the match, and extensions from that match (\( J.o \)). \( J.r \) in this mode is always set to true so that extensions generated from different column jobs can be combined at the parent job. Also, \( J.c \) is always set to false.

Cost analysis

A key factor to the performance of our parallel framework is the amount of overhead incurred in creating, sharing, and managing jobs and job pools. This overhead is minimal due to following reasons: (i) we avoid any type of meta data structures, making it is easy to fork off new jobs from current computation; (ii) all jobs have very small sized inputs (a small pattern, a tree id, or a column id), and so it is easy to create and share them; (iii) all jobs are shared using simple queueing and locking mechanisms; and (iv) all job spawning conditions can be evaluated in constant time.
Another source of overhead is the number of context switches between different execution modes. We now develop some theoretical bounds on that number by analyzing various job spawning conditions. Let \( N(t, S) \) be the number of times the spawning condition that results in jobs of type \( t \) is evaluated to \( \text{true} \), while processing \( S \). Similarly, let \( N(S) \) be the number context switches (of any type) while mining \( S \), and \( N \) be the total number of context switches during entire execution. We now have,

\[
N(S) = N(\text{task}, S) + N(\text{data}, S) + N(\text{chunk}, S)
\]

\[
N = \sum_s N(S)
\]

We now construct the worst case bounds for \( N(t, S) \) for each \( t \). While mining \( S \), new \textit{tasks} are created only through adaptive task partitioning. It is performed only after all extensions are produced from \( S \) (see Section 5.1.2). Any subtree can thus produce new tasks \textit{at most} once. We now have,

\[
\forall S, N(\text{task}, S) \leq 1 : \sum_s N(\text{task}, S) \leq \sum_s 1 = |FS|
\] (5.1.3)

where \( FS \) is the set of all frequent subtrees. When a task \( J \) spawns jobs onto tree pool, each unexplored tree in \( J \)’s projected database is created as new job. Once the tree pool is processed, it is \textit{guaranteed} that all trees in the projected database are processed for extensions. Thus for any subtree, the switch from task parallel mode to data parallel mode can happen \textit{at most} once.

\[
\forall S, N(\text{data}, S) \leq 1 : \sum_s N(\text{data}, S) \leq |FS|
\] (5.1.4)

Finally, \( N(\text{chunk}, S) \) is equal to the number of trees in \( S \)’s projected database which spawn the chunk-level jobs. From Section 5.1.2, jobs of type \textit{chunk} are created only when TP is empty. We can thus infer that \( N(\text{chunk}, S) \) is always less than the number of cores. If \( N(\text{chunk}, S) \geq |C| \) then TP can not be empty. Therefore,

\[
\forall S, N(\text{chunk}, S) \leq |C| - 1 : \sum_s N(\text{chunk}, S) \leq |FS| * (|C| - 1)
\] (5.1.5)

From Equations 5.1.3- 5.1.5,

\[
N = \sum_s N(S)
\]

\[
= \sum_s N(\text{task}, S) + \sum_s N(\text{data}, S) + \sum_s N(\text{chunk}, S)
\]

\[
\leq |FS| + |FS| + |FS| * (|C| - 1)
\]

\[
\leq |FS| * (|C| + 1)
\]
Thus, the number of context switches per pattern is bounded by a constant, and the total number $N$ is in the order of $\vert FS \vert$. However in practice, these numbers are very very small since the algorithm moves to a lower granularity only when the parallelism at current granularity is completely exploited. For example, many subtrees would have already been enumerated by the time the first data-parallel job is created i.e., $\sum_s N(\text{data}, S) \ll \vert FS \vert$.

**Empirical Evaluation**

We evaluate our algorithms using two commonly used real-world data sets, Treebank (TB) [4] and Cslogs (CS) [207] – derived from computation linguistics and web usage mining, respectively. The number of trees and the average tree size (in number of nodes) in CS and TB are (59691, 12.94) and (52581, 68.03), respectively. We use a 900 MHz Intel Itanium 2 dual processor system with 4GB RAM, and if more memory is required (typically by extant algorithms), we use a system with 32GB RAM (same processor) instead of relying on virtual memory.

We consider two data set characteristics which affect the performance – number of frequent patterns NFP (affects the run time), and average number of matches per frequent pattern NM (affects the memory usage). While NFP depends on $\text{minsup}$, we find that many frequent patterns found in both datasets have a large number of matches in the data. While the trees in TB possess a very deep recursive structure, the tree nodes in CS exhibit a high variance in their label frequencies. This high variance makes NM to increase at a much faster rate in CS as we decrease the support (see Figure 5.4). We will pinpoint the influence of these properties when discussing the relevant experimental results. Hereafter, $DS$-$\text{minsup}$ denotes an experiment where $DS$ is a data set and $\text{minsup}$ is the support.

**Sequential Performance**

**Effect of optimizations:** We highlight the benefits from our optimizations in Figure 5.5 by considering the run time and memory usage of Trips as the baseline. Note that the Y-axis in 5.5b & 5.5d is shown in reverse direction to indicate the reduction in memory usage. The memory footprint of algorithms is approximated as its resident set size (RSS) obtained from the “top” command. The results shown for each optimization include the benefits from all the other optimizations presented before that. So, $\text{CHUNK}$ refers to fully optimized Algorithm 12 (MCT).
Figure 5.4: Change in NM and NFP as a function of \textit{minsup}

Figure 5.5: Performance comparison with Trips as the baseline
Even though the dynamic lists from NOEM decrease the memory consumption of Trips, they add to the run time overhead due to redundant recursions in Algorithm 11. In case of TB-40K alone, NOEM slowed down Trips by 3.6 times – due to 10 billion recursions in finding just 413 million subsequences, which include about 289 million false positives (i.e., about 7 out of 10). While LF and DOM streamline the backtracking process to reduce the number of recursions to mere 554 million, SIMUL eliminates all 289 million false positives – giving a 23% run time improvement over Trips. More importantly, these optimizations improve the run time without affecting the memory benefits from NOEM. Subsequently, CHUNK (or MCT) by reorganizing the computation, improves the locality and reduces the working sets resulting in a very good run time and memory performance. When compared to Trips, on TB – 30K, MCT performs 24% faster and uses 45-times lesser memory. Similarly on CS-600, our optimizations improve the memory usage by 366-folds and run time by 3.7-em times.

![Figure 5.6: Effect of Chunk Size on the Run Time Performance](image)

The effect of chunk size on the performance is evaluated in Figure 5.6. We show the change in execution time as the value of chunk size is changed from 1 to $10^4$. While Cslogs data set is mined at support threshold of 1%, patterns from Treebank data set are mined at 56% threshold. As the chunk size increases, the time spent in mining all frequent patterns reduces initially due to improved data locality – more number of matches will fit into the cache. Such a reduction in run time is observed until the chunk size reaches an optimal value, beyond which the performance degrades
due to increased number of cache misses. Note that the optimal value depends not only on cache parameters (capacity, line size, etc.) but also on data set properties. Evidently, the optimal chunk size is different in both the data sets considered in Figure 5.6. We observed similar trends with varying support thresholds. The size of memory footprint naturally increases with the chunk size. However, we observed that the overall effect on memory footprint is minimal, especially when compared to the change in run time performance.

**Comparison with TreeMiner.** The performance of TreeMiner is limited by the number and the size of scope-lists, which depend upon the data set properties like NM (see Figure 5.4). For example, when a frequent edge in Cslogs is grown into
a 6-node pattern, the number of matches increased sharply from 11,339 to 141,574 to 2,337,127 to 35,884,361 to 474,716,009 – resulting in large scope-lists which are later used in expensive joins. Due to such patterns, as the support is changed from 1000 to 800, the memory and run time performance degraded by more than 300 times and 18.5 times, respectively (see Figure 5.7). In contrast, MCT always maintains a constant sized footprint – 10.72MB on Cslogs & 34MB on Treebank – irrespective of the support threshold. Since chunking keeps a fixed number of matches in memory at any given point in time, MCT is able to regulate the memory usage – a significant result for CMPs where the bandwidth to memory is precious. On CS-700, while TreeMiner ran for more than 100 hours with a footprint that is larger than 7GB, MCT took about 50sec exhibiting a 7200-fold speedup along with 660-fold reduction in memory usage. Even if we factor out the algorithmic benefits from Trips, the benefits from our optimizations are quite significant.

In case of Treebank, the deep recursive structure among trees limits the performance of TreeMiner (see Figures 5.7c & 5.7d). As a result, even a small change in support (from 50K to 35K) degrades the performance significantly (by more than three orders). On TB-35K alone, MCT exhibits more than 400-fold speedup and 120-fold smaller memory footprint.

**Comparison with iMB3-T**: iMB3-T takes a parameter “level of embedding” (L) that controls the type of subtrees that are mined. When L is left unspecified, it mines embedded subtrees – Figure 5.7 obtained using this setting. Multiple large data structures and apriori-style mining of iMB3-T results in very large memory footprints. Note that, its memory is affected by both NM and NFP, which increase exponentially with the decrease in support (see Fig 5.4). On CS-700, memory and run time performance of MCT is better than iMB3-T by 66-times and 2,300-times, respectively. iMB3-T is aborted at CS-600 as its memory usage exceeded 32GB – no corresponding data point in Figure 5.7a. It stores the set of all descendants for every frequent node, and hence the deep recursive structure in TB results in very large footprints even at high support values (e.g., 8.5GB at 50K support). On TB=40K, MCT is 780-times faster than iMB3-T while using 480-times lesser memory.

**Comparison with FreqT**:

We now perform an evaluation of our MCT algorithm modified to mine induced subtrees (see Section 5.1.2) against two other algorithms: FreqT and iMB3-T (L=1) (see Figure 5.8). We obtained the source code of FreqT from a publicly available implementation [115]. Since induced subtrees are structurally much simpler than
embedded subtrees, overall run times in Figure 5.8 are much smaller than that of in Figure 5.7. When compared to FreqT, our MCT algorithm demonstrated up to 10-fold improvement in memory usage, and more than 15-fold speedup in run time. The embedding lists stored by FreqT stores only a small amount of information (location of right most leaf node), and as a result, its memory footprint is comparable to MCT. In contrast, iMB3-T, even at very high supports, keeps large memory footprints and exhibits very poor run time performance. For example at minsup=50%, the memory usage iMB3-T was more than 8GB (>> 62MB of MCT), and the mining time was more than 7,500sec (>> 64sec of MCT). The memory footprint of iMB3-T on Cslogs is approximately 30 times larger than that of MCT.

---

**Figure 5.8: Induced Subtree Mining: MCT vs. Existing Algorithms**

(a) Cslogs - Execution Time

(b) Cslogs - Memory Footprint

(c) Treebank - Execution Time

(d) Treebank - Memory Footprint
Finally, a note on out-of-core performance – a straightforward realization of our memory-conscious optimizations results in an out-of-core implementation that is about six times slower than the in-core algorithm for the evaluations described so far in this section. Once the data set exceeds the limits of main memory the performance of the in-core implementation degrades rapidly whereas the out-of-core implementation sees a much more graceful degradation. An interesting observation here is that computation chunking has an even larger role to play in our out-of-core realization (by a factor anywhere from 5 to 10) since chunk-wise processing localizes the computation to run in memory. More sophisticated out-of-core algorithms can be designed by leveraging hash-based data placement techniques (described later in Section 5.2.4), in a manner similar to the work on out-of-core frequent pattern mining [40].

Characterization study for CMP architectures

We now show that our optimizations are suitable for multicores by collecting several hardware performance counters using PAPI toolkit [1]. To this purpose, we run a TB-45K experiment on a system with 1.4GHz Itanium 2 processor and 32GB memory. The system under consideration has following on-chip caches – 16KB L1-data; 16KB L1-instruction; 256KB L2; and 3MB L3.

**Analysis of cache performance:** We demonstrate the effect of all our optimizations, measured in terms of number of cache misses, in Figure 5.9a by taking NOEM in Algorithm 11 as the baseline. Tree matching optimizations improve the cache performance by more than 19 times – while LF shrinks R-matrices, DOM and SIMUL reduce the number of data accesses, thereby improving L2 and L3 misses. Added to that, CHUNK localizes the computation to higher level caches, and improves the L1 misses of NOEM by a factor of 1,442. A step-by-step effect of various optimizations on run time is shown in Figure 5.10d. Overall, simultaneous matching and especially computation chunking help in achieving very good cache performance.

**Analysis of bandwidth pressure:** Since all the cores of a CMP system share a single memory bus, memory bandwidth becomes a key factor to application performance. We devise a novel and simple method to approximate the memory bandwidth by observing the amount of traffic on the front side bus (i.e., off-chip). We first divide the execution time (X-axis in Figure 5.9b-d) into small one msec slices – a coarse-grained analysis. Then the amount of off-chip traffic during each slice (Y-axis) is approximated to be the product of L3 line size and the number of L3 misses in that slice (recorded by PAPI).
Figures 5.9b-d show the variations in off-chip traffic for TreeMiner, Trips, and MCT, respectively. iMB3-T is not considered here due to its poor run time and memory performance. Initial spikes in these figures denote cold L3 misses incurred while bootstrapping (e.g., reading the data set). Frequent accesses to large memory-bound scope-lists result in very high off-chip traffic for TreeMiner. Each cluster of points in Figure 5.9c denotes the traffic seen while mining a single subtree. The spikes followed by sudden dips indicate the non-uniform nature of computation in Trips. In contrast, the well-structured computation of MCT results in more uniform and small sized memory requests. On an average, accesses made by MCT are well below 200KB per msec whereas the accesses made by TreeMiner and Trips are sized more than 1100KB and 600KB per msec, respectively. This difference is even more while mining the patterns with large number of matches – compare small spikes around 6000 msec in Figure 5.9d with the large ones around 8000 msec in Figure 5.9c. From this coarse-grained study it appears that each core in TreeMiner, and to a lesser extent in Trips, aggressively attempts to access main memory (due to embedding lists). For instance, on a dual quad-core system from Section 5.1.2, we observed a sustained cumulative bandwidth of 1.5GB per sec. With 1100KB per msec accesses (i.e., 1GB per sec per core) by TreeMiner, the bandwidth is likely to saturate it is executed on multiple cores. Overall, our optimizations reduce the off-chip traffic and its variability, making them viable for CMPs.

Analysis of working set size: We empirically examined the working sets maintained by different algorithms using Cachegrind [3]. We monitored the change in L1 miss rate by varying the L1 size from 2KB to 256KB (L2 size and its associativity is fixed). We found that L1 miss rate of MCT reduced sharply between 8KB and 16KB and stayed constant for L1 size > 16KB. This suggests that the working set size is between 8KB and 16KB. As shown in Table 5.1, other algorithms maintain relatively larger working sets. This is an encouraging result with respect to CMPs as the amount of cache available for each core is likely to be small [117].

Parallel Performance
We evaluated our parallel algorithms on a dual quad-core E5345 Xeon processor system with 6GB RAM, 8MB shared L2, and 1333 MHz front side bus speed. – see Figure 5.10a. Our adaptive load balancing strategies achieve near-linear speedups up to 7.85-folds on CS and 7.43-folds on TB, when all 8 cores are used. We test the scalability of our techniques on a 16-node SGI Altix 350 SMP system with 16 1.4GHz
Figure 5.9: Characterization of Memory-conscious Optimizations

Figure 5.10: (a, b, c) Parallel performance; (d) Effect of optimizations
Itanium 2 processors and 32GB memory. As shown in Figure 5.10b, the speedup continues to increase with the number of processors, giving a 15.5-fold speedup with all 16 processors. Load balance achieved by individual strategies for $TB$-$45K$ is demonstrated in Figure 5.10c.

An important observation from Figure 5.10b is that the need for fine-grained strategies increases as one increases the number of processors. For $CS$-600, the performance of hybrid strategy (HyP) reaches its plateau at 12 processors ("CS-600 Hybrid" in Figure 5.10b) due to a 6-node pattern that has up to 33 million matches in a single database tree, whose mining took about 45sec. Amdahl’s law suggests that HyP can never perform better than 45sec since it is limited by the job of mining a single tree. Thereafter the efficiency can only be improved by employing more fine-grained strategies such as the one in Section 5.1.2. Similarly for $TB$-$35K$, the speedup from HyP saturates at 16 processors.

The average number of context switches taken over 10 runs of $TB$-$35K$ is shown in Table 5.2. For a given granularity $g$, $\sum_{S} N(g, S)$ is denoted as $N_g$ in the table. When $|C|=1$, there are no context switches as the work is not shared at any level. As $|C|$ increases, we see more and more context switches at fine-grain level reflecting the fact that our strategies adaptively also automatically exploit the parallelism at all levels of granularity. It is worth noting that these numbers are much lower than their theoretical upper bounds from Section 5.1.2: $N_t=48 \ll |FS|=451$; $N_d=11 \ll 451$; and $N_c=19 \ll 451*|C|-1$, where $|C|$ is number of cores. Similar results on $CS$-600 can be found in Table 5.3.

Note that the performance numbers in Table 5.2 are directly obtained from our service. We designed an interesting performance monitoring tool by leveraging the capability of our service to produce such useful numbers and our light-weight mechanism to approximate achieved memory bandwidth (see Section 5.1.2). Such a tool not only is capable of providing real time feedback to applications but is also useful to understand the performance characteristics of many applications on CMPs.

| Cores ($|C|$) | 1 | 2 | 4 | 8 | 16 |
|---------------|---|---|---|---|----|
| $N_t$         | 0 | 4 | 7 | 26| 48 |
| $N_d$         | 0 | 2 | 2 | 10| 11 |
| $N_c$         | 0 | 0 | 0 | 9 | 19 |

Table 5.2: $TB$-$35K$, $|FS|=451$

| Cores ($|C|$) | 1 | 2 | 4 | 8 | 16 |
|---------------|---|---|---|---|----|
| $N_t$         | 1 | 1 | 3 | 4 | 3  |
| $N_d$         | 2 | 3 | 6 | 5 | 5  |
| $N_c$         | 0 | 0 | 0 | 0 | 7  |

Table 5.3: $CS$-600, $|FS|=252$
Table 5.4: Parallelization of TreeMiner

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeMiner</td>
<td>EqP</td>
<td>1.00</td>
<td>1.61</td>
<td>1.94</td>
<td>1.95</td>
<td>2.01</td>
</tr>
<tr>
<td></td>
<td>AdP</td>
<td>1.00</td>
<td>1.77</td>
<td>2.23</td>
<td>2.25</td>
<td>2.30</td>
</tr>
</tbody>
</table>

The results in Figure 5.10 are obtained using a global job pool. However, our service can handle distributed or hierarchical job pools. Further, we expect the contention overhead due to global job pools to be very small as the locking on CMPs is likely to be cheap [72].

Parallel speedups of TreeMiner algorithm using our task-level methods are shown in Table 5.4. The rationale for these results is as follows. Inherent dependency structure in scope-lists makes it difficult to apply more fine-grained strategies to TreeMiner. Since these lists are maintained on per-pattern basis, data partitioning methods like HyP, which construct the lists in parallel incur significant synchronization overhead. Further, excessive use of dynamic data structures in TreeMiner serializes the heap accesses, affecting the parallel efficiency – as \(|C|\) is changed from 1 to 8, the system time (from the “time” command) increased by more than 4 times. Techniques like memory pooling are ineffective here as these data structures grow arbitrarily in size. These results re-emphasize the following mantra for good parallel efficiency: reduce the memory footprints; reduce the use of dynamic data structures; and reorganize the computation so that more fine-grained strategies can be applied.

5.1.3 Parallel Tree Indexing and Querying

We will now consider the use of multicore systems to design parallel algorithms for XML database indexing. We present several methods to parallelize the our LCS-TRIM framework presented in Chapter 2. The techniques however have a much general applicability. As described in Section 2.2, our tree indexing method consists of three major steps – (i) data transformation; (ii) index construction; and (iii) query processing. The final step has two important sub-parts, viz. list intersection and subtree matching. In this section, we discuss how multicore systems can be leveraged to perform each of these steps.

First, the data transformation step maps each tree in the database to its sequential representation using Algorithm 1. Performing this step on a shared memory system involves a simple data-parallel approach where each thread (or processor) runs the
transformation algorithm on a different tree. The set of all trees can be treated as a task pool where each task corresponds to a single tree. All threads extract tasks one by one, and process them. Since the transformation of different trees is totally independent of each other, the synchronization overhead incurred in the overall process is very minimal. Only the accesses to common task pool needs to be synchronized.

Parallel Index Construction

In index construction phase, an inverted index is built on the set of labels. It is a union of \((l, L)\) pairs, where \(l\) refers to a tree node label in the database, and \(L\) refers to the list of trees in which \(l\) presents (see Section 2.2.3). The list \(L\) is maintained in sorted order for computational efficiency while performing the query processing. Given a database \(D\) of trees, and a shared memory system with \(|P|\) processors, Algorithm 14 presents a method in which the index is constructed, in parallel.

The block of code in parallel for loops from Lines 2 and 12 are executed by all threads simultaneously. For example in Line 2, the database \(D\) is treated as a task pool and all trees from \(D\) are processed by all threads in \(P\). \(thd\) in Line 3 refers to a data structure that is local to each thread. It can be accessed only by the thread that owns it, unless it is explicitly made global by the owner. Several parallel thread libraries provide such a feature. For example, when using POSIX Pthreads API, one can declare thread-specific data via pthread_key variables [140]. Such data structures are allocated on private memory block that is local to each thread.

The thread-specific data structure \(thd\) holds an inverted index that is maintained locally by all threads. All the tree node labels in a given tree \(T\) are added to this local inverted index structure in Lines 4 – 7. Once all trees in the database are processed, all local index structures are made global in Line 9 – each thread provides read permissions to all other threads so that they can access its local inverted index. As each tree is processed, the set of all labels scanned so far are maintained in \(LabelSet\) – Line 6. This step must be performed in a mutually exclusive manner as \(LabelSet\) is common to all threads.

The global index structure is then built in Lines 12 – 16, by processing node labels from \(LabelSet\). For each label \(l\), all local indices are probed in Line 13 to obtain lists corresponding to \(l\). \(index(p).list(l)\) corresponds to a list from index built on processor \(p\). The combined list \(L_u\) contains the set of all trees in which the current label \(l\) is present. This list is sorted, and subsequently added to the global index in Line 15.
The overhead incurred due to synchronization in Algorithm 14 is very small. Only the processing in Lines 6 and 15 corresponds to critical section because they access global data structures, and they requires some form of coordination. As the amount of work performed in critical section is very small, the overhead is minimal. We can thus expect the parallel efficiency of index construction step to be high.

Algorithm 14 Parallel Index Construction

| Input: Database $D = \{T_1, T_2, \cdots, T_n\}$, Set of threads $P$ |
| 1: $LabelSet \leftarrow \phi$ |
| 2: for each tree $T$ in $D$ do (in parallel) |
| 3: $thd$ $\leftarrow$ thread specific data \{local to the thread that creates it\} |
| 4: for each node $m$ in $T$ do |
| 5: add $m.label$ to $thd.index$ \{$thd.index$ is an inverted index\} |
| 6: add $m.label$ to $LabelSet$, if it is not already present |
| 7: end for |
| 8: end for |
| 9: Mark $thd.index$ of every thread as global |
| 10: Say $index(p)$ refers to the index that is constructed on thread $p$ |
| 11: $InvertedIndex \leftarrow \phi$ |
| 12: for each label $l$ in $LabelSet$ do (in parallel) |
| 13: $L_u \leftarrow \bigcup_{p=1}^{\mid P \mid} index(p).list(l)$ |
| 14: sort $L_u$ |
| 15: add $(l,L_u)$ to $InvertedIndex$ \{requires synchronization\} |
| 16: end for |
| 17: return $InvertedIndex$ |

Parallel Query Processing

Given a query $q$ with twig nodes $(q_1, \cdots, q_{|q|})$, query processing in $LCS-TRIM$ involves the following three main steps:

1. For each query node label $q_i$, probe the inverted index to fetch its list $L(q_i)$

2. Intersect all fetched lists $L(q_i)$, $1 \leq i \leq |q|$ to obtain a set of trees in which all query nodes present. Let this set be denoted by $CL$

3. Finally, perform subtree matching on each tree present in $CL$ using Algorithm 4.
The first step is very simple as it only involves several accesses to the index structure built in Algorithm 14. Most of the processing time is spent in second and third steps. We now discuss how one can parallelize these two steps.

Parallel List Intersection:

First, consider the second step that involves intersecting multiple lists fetched from the index structure. More formally, we need to construct a set $CL = \cap_{i=1}^{a} L(q_i)$ such that the resulting set consists of candidate trees containing all query nodes. As mentioned in Section 2.2.3, tree identifiers in all inverted index lists are maintained in sorted order. These lists are often stored in compressed format to reduce the space occupied by the index structure. Tatikonda et al have proposed an approach to intersect sorted lists stored in compressed format [173]. This work has been done in the context of web search query processing. Similar method can be employed here to construct $CL$, in parallel. For completeness, we briefly describe their method here.

![Figure 5.11: Skip Lists](image)

The main idea behind their approach is to generate fine-grain tasks to perform the list intersection concurrently on multiple cores. However, the compressed lists in the inverted index limit the ability to perform random accesses on the lists. A treeID at a position $i$ can be accessed only after decompressing the portion of list prior to $i$. To facilitate effective work partitioning while retaining the benefits of compression, the lists are stored in the form of skip lists. A skip list is a sorted list of tree identifiers (also referred to as items) denoted using a linked list that connects sparse subsequences of items [153]. Each link is a pointer $i \rightarrow j$ between two non-consecutive items $i$ and $j$ in the list. The term skip is used interchangeably to indicate
a link or the end points of a link. The number of items skipped between two skips \(i\) and \(j\) is referred to as skip block size or skipsize.

Two example skip lists are shown in Figure 5.11, where \(a_i\)'s and \(b_j\)'s are the skips in List1 and List2, respectively. Space overhead is reduced by storing all tree identifiers between a pair of skips in compressed format, and the skips themselves are maintained without any compression. For computational convenience, the highest tree identifier in the list is stored as a separate skip pointing to Null. More formally, the skip list for a query node label \(a\) is denoted as \(L(a)=(S_a, C_a)\) where \(S_a\) is the sequence of skips in the skip list and \(C_a\) is the sequence of tree ids in compressed format.

![Diagram](image)

**Figure 5.12: Parallel List Intersection Framework** [173]

The parallel framework in which given skip lists are intersected is shown in Figure 5.12. It follows a producer-consumer model where the producer thread makes a pass over skip lists to create small and independent tasks. These tasks are processed subsequently by all threads to produce the list \(CL\). Each task is formed by the sequence of skips obtained from all lists in the query. The total number of tasks generated for a given query is equal to the number of skip blocks present in the shortest posting list.

Assume that a given twig query \(q\) has two node labels \(a\) and \(b\), whose lists from the inverted index are \(L(a)=(S_a, C_a)\) and \(L(b)=(S_b, C_b)\) with \(m = |S_a|, n = |S_b|,\) and \(m < n\). For each skip pointer in \(L(a)\), a task is created with one or more skips from \(L(b)\). The set of all independent tasks generated from \(q\) would be \(\{t_1, t_2, ..., t_m\}\), where a task \(t_i = [(s_i, s_{i+1}), (s_j, s_k)]\), \(s_i, s_{i+1} \in S_a\) for \(1 \leq i \leq m\) and \(s_j, s_k \in S_b\) for \(1 \leq j \leq k \leq n\). Note that, \(s_{i+1}\) is undefined when \(i = m\). Furthermore, for a given \(s_i\)
and $s_{i+1}$ in $L(a)$, the skips from $L(b)$ are chosen such that $s_i \geq s_j$ and $s_{i+1} \leq s_k$. For the example skip lists in Figure 5.11, a total of three tasks are produced (shown in the figure) by considering the sequence of skips $(a_1 = 20, a_2 = 35, a_3 = 90, a_4 = 175)$ from List1 and $(b_1 = 10, b_2 = 50, b_3 = 90, b_4 = 145, b_5 = 205)$ from List2.

The above described method can easily be extended to queries with more than two query node labels. It is important to note that each task encapsulates the complete information, and hence, it can be processed independently on any processor. Once the tasks are created by the producer process, they are extracted from the task pool one by one and are processed by all threads in the system. The processing of each involves decompression and intersection of resulting decompressed lists [173].

Parallel Subtree Matching:

The set $CL$ computed in the list intersection phase consists of candidate trees that may contain matches for given query twig $q$. The final answer for $q$ is computed by executing the subtree matching algorithm (Algorithm 4) on each candidate tree $T \in CL$. This step is very similar to Lines A-C of MCT Algorithm 12, where newtidlist is constructed by enumerating the matches of a subtree newpat in trees pointed by tidlist. While the given query $q$ corresponds to newpat in Algorithm 12, the set $CL$ is equivalent to tidlist. The goal is to find all matching trees i.e., constructing newtidlist.
Recall that the adaptive strategies discussed in Section 5.1.2 are aimed at parallelizing MCT Algorithm 12. Similar strategies can be employed here to process the set $CL$ in parallel. More specifically, the data-parallel and chunk-parallel modes are directly applicable. While the data-parallel mode operates on different trees of $CL$ in parallel, the chunk-parallel mode processes a single tree $T \in CL$ in parallel to obtain the matches of $q$ within $T$.

We evaluated the data-parallel mode on the following query from Cslogs [207] data set:

```
//1155[//5996][//5996][//5996][//5996][//5996][//5996]
```

The experiment is run on a SMP system with 8 processors and 32 GB of shared memory. The considered query has a large number of matches (474, 716, 009 spread over 608 trees) in the data set. Each processor in the SMP system picks up a candidate tree from the list returned by the index and finds the twig matches in that tree. As shown in Figure 5.13, the data-parallel mode can provide up to 6.3-fold speedup when using all 8 processors. The distribution of 474 million matches over 608 trees is not uniform. As a result, the data-parallel mode suffers from load imbalance and delivers only sub-linear performance. The chunk-parallel mode, a more fine-grained parallelization mechanism, can be used to further improve the efficiency (see Figure 5.10b).

5.1.4 Scheduling Service for Multicores

A key outcome of our efforts in adaptive parallelization is a scheduling service that is targeted at multicore chips and SMP systems. We believe that such services will be ubiquitous as systems grow more complex and are essential to realize performance commensurate with technology advances. For simplicity, we limit our discussion to the basic interface shown in Algorithm 15. Functions I1 and I2 are basic start and clean-up routines. Jobs in our system are implemented using job descriptors (see above). Once the service is started, I3 specifies the list and the order among different granularities which the application wants to exploit. It also creates different job pools and other data structures used for scheduling. $gOrder$ determines the order in which the job pools are accessed. For each granularity, I4 defines an application handle that is invoked to execute the the job of that granularity. I5 (optionally) registers a synchronization callback handle that is used for jobs whose return flag is set to true. I6 is responsible for scheduling and completing all jobs by performing
context switches, if needed (similar to Algorithm 13). I7 and I8 are invoked for the creation and execution of jobs. I9 is a check point function used to evaluate whether or not to switch between different granularities. This function is implemented by the application itself. A typical implementation of this function must evaluate two main conditions – (i) check thread pool to determine if there exist any ideal threads waiting for work; and (ii) if the first condition evaluates to true then check if there is a need to change the task granularity. This is done by assessing the complexity of current work. Such process in the context of our parallel tree mining algorithm is shown in Figure 5.14. It can be see from the figure that the overhead incurred in repeated invocations of I9 is negligible as its implementation involves evaluation of simple conditions.

![Diagram of Different Parallelization Modes](image)

**Figure 5.14: Implementation of Different Parallelization Modes**

The way we invoke different routines from the interface is shown in Algorithm 13. Different granularities are set up by invoking I3, I4, and I5 in line 2. Lines 11 and 13 call the `sync` from I5 since the jobs at `data` and `chunk` level require coordination. Entire scheduling of jobs i.e., lines 3-15 make up the implementation of I6.

We have specifically employed this service for the task of tree mining but we expect it to be useful for a range of pattern mining tasks (from itemsets to graphs) as well as more broadly for other data-intensive applications. For example, one can easily
Algorithm 15 Prototype interface for scheduling service

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void startService ()</td>
</tr>
<tr>
<td>2</td>
<td>void stopService ()</td>
</tr>
<tr>
<td>3</td>
<td>int register ( int *granularities, int size, int *gOrder )</td>
</tr>
<tr>
<td>4</td>
<td>int bind ( int gran, void (*callback) (void *) )</td>
</tr>
<tr>
<td>5</td>
<td>int finalize ( int gran, void (*sync) (void *) )</td>
</tr>
<tr>
<td>6</td>
<td>void schedule ()</td>
</tr>
<tr>
<td>7</td>
<td>int createJob ( int gran, void *inputs )</td>
</tr>
<tr>
<td>8</td>
<td>int executeJob ( job *j )</td>
</tr>
<tr>
<td>9</td>
<td>bool evaluateForSpawning ( job *j ) {implemented by the application}</td>
</tr>
</tbody>
</table>

parallelize famous algorithms like FP-Growth [92] and gSpan [200] using our service (see Section 5.3). The current implementation is limited to CMPs and SMPs but we are in the process of extending this service for cluster systems comprising of multicore nodes. This service can also be implemented on top of Intel’s TBB and other related libraries for portability, as opposed to current pthreads implementation. As shown in Section 5.1.2, the service is capable of producing some useful performance statistics. We leverage this feature in designing a performance monitoring tool that provides real-time feedback to applications. It can be used in a variety of applications running on CMP architectures.

5.2 Algorithms for Distributed Cluster Systems

5.2.1 The Framework

The goal is to design parallel algorithms to mine and manage tree-structured data on a distributed cluster system with multiple compute nodes. A distributed pattern mining algorithm, for example, asks us to compute the set of all frequent subtrees from a database of trees that is distributed across different compute nodes in a shared-nothing system. As depicted in Figure 5.15, the overall process in most distributed algorithms can be logically divided into following three phases:

1. **Partitioning**: The given database is partitioned and distributed onto all compute nodes in the cluster.

2. **Distributed Processing**: A sequential algorithm is run on individual partitions to obtain partial sets of result.
3. **Joining**: Partial result sets from each partitions are joined to form the complete answer to a given task.

One or more phases can either be combined with each other or even be non-existent depending on the specifics of each algorithm. In the context of both mining and indexing tree databases, the computation performed in second and third phases is relatively simple. For instance, the second phase in a distributed frequent subtree mining involves running a sequential mining algorithm (e.g., MCT Algorithm 12) to produce a partial set of frequent patterns from each partition. These sets are combined and false positive patterns are pruned in the third phase by making one or more scans on the database partitions. Similarly, the second phase in a distributed indexing approach consists of querying individual partitions using a sequential method (e.g., LCS-TRIM 4). The set of matching trees from each partition are then combined in the join phase and presented to the user.

![Figure 5.15: Framework for Distributed Algorithms](image)

The main challenge in developing efficient distributed algorithms for tree-structured data is in the way first phase is designed, where the data set is partitioned across different compute nodes. One may employ a number of different algorithms for this purpose. As shown later in this section, the exact choice of partitioning method greatly affects the overall performance, especially during the join phase. Before we discuss these details, we first present an framework in which different distributed pattern mining methods can be compared analytically.
Analytical Framework for Frequent Subtree Mining

We first introduce some notation. The goal is to mine a database $D$ of trees on a distributed cluster with $|P|$ compute nodes. Let $S^a$ denote the actual set of all frequent patterns obtained from $D$ by running a sequential algorithm (e.g., MCT Algorithm 12). Let $D_i$ be the portion of database that is assigned to a compute node $P_i$, $1 \leq i \leq |P|$. The set of frequent patterns obtained from each partition is denoted by $S_i$. Let the union set of patterns from all partitions is represented by $S^u$.

$$S^u = \bigcup_{i=1}^{P} S_i$$

A frequent pattern that is local to one or more $S_i$’s may not be frequent globally. Therefore, the sets $S^a$ and $S^u$ may not be identical. There may exist false positives in such a way that $f \in S^u$ and $f \notin S^a$. However, due to the way frequency is computed, there can not exist any false negatives. A true frequent pattern must be frequent in at least one partition [6]. In other words, if $f \in S^a$ then $f \in S^u$, as there exists at least one partition such that $f \in S_i$.

From its definition, a subtree $f$ is considered frequent if its support is greater than a given threshold. Let $C^a_f$ be the total number of database trees in which $f$ occurs as a subtree – i.e., its actual support in $D$. Similarly, $C^i_f$ be the count associated with $f$ in a specific partition $P_i$. Then, $f$’s count in $S^u$ can be computed as follows:

$$C^u_f = \sum_{i=1}^{P} C^i_f$$ \hspace{1cm} (5.2.1)

We now construct two numerical measures that are useful to analytically evaluate the performance of different distributed pattern mining algorithms. They are the following:

1. $N^u$: As mentioned earlier, it is possible for union set $S^u$ to contain false positive patterns. $N^u$ measures the number of such patterns in $S^u$ that must be re-evaluated against the database for their global frequency. It is equal to the number of patterns whose combined frequency is not sufficient to declare them as frequent. The following formula computes this number:

$$N^i = |\{f \in S^i|C^u_f < \text{minsup}\}|, 1 \leq i \leq |P|$$

$$N^u = \sum_{i=1}^{P} N^i = |\{f \in S^u|C^u_f < \text{minsup}\}|$$ \hspace{1cm} (5.2.2)
2. **Cost**: While the above measure focuses only on the frequent patterns, one may also be interested in the exact counts associated with each pattern returned by the distributed algorithm. Even though a pattern’s count in the union set is sufficient to declare it as frequent, the count may not be same as its actual count, i.e., \( C_f^u \neq C_f^o \). All patterns in \( S^u \) with inexact counts must be re-evaluated against one or more partitions to obtain their actual counts. Let \( I_f(i) \) be an Indicator function that indicates whether or not the pattern \( f \) belongs to \( S_i \). It is defined as follows:

\[
I_f(i) = \begin{cases} 
0, & \text{if } C_f^i = 0; \\
1, & \text{otherwise.}
\end{cases}
\]

The number of partitions that must be accessed to determine the exact count of the pattern \( f \) can then be computed as \( Cost_f = |P| - \sum_{i=1}^{N^u} I_f(i) \). In an ideal scenario, the cost associated with a pattern would be zero, when it occurs in every partial set i.e., in all \( S_i \)'s. The overall cost associated with each partition, and the total cost can be computed as follows:

\[
Cost^i = \sum_{f \in S_i} Cost_f \\
Cost^u = \sum_{i=1}^{|P|} Cost^i = \sum_{f \in S^u} Cost_f 
\]

(5.2.3)

The total cost \( Cost^u \) will be zero if and only if both the sets \( S^a \) and \( S^u \) are identical. Smaller the values for \( N^u \) and \( Cost^u \), the better is the performance. Higher values for these two measures result in too many accesses to individual database partitions during the join phase. We will now discuss the specific methods we develop for tree mining and tree indexing.

### 5.2.2 Distributed Frequent Subtree Mining

As mentioned earlier, the main challenge is to design effective data partitioning strategies (the first phase in Figure 5.15). Along with user-defined input parameters for the mining task, the method that is used to partition the database is also critical to the overall performance. A bad partitioning method results in large values for \( N^u \) and \( Cost^u \). A good partitioning strategy, in the case of frequent pattern mining, must spread the database in such way that the distribution of data records in each
partition closely matches with that of the entire data set. Such a method decreases the number of false positive patterns, and as a result, reduces the work performed in the join phase. We describe one such strategy in this section. A baseline strawman method, that we refer to as random partitioning, is to randomly assign each database record to one of the $|P|$ partitions. While doing so, it may also be useful to ensure that all partitions are of similar size, for better load balance.

While the random partitioning is very easy to implement, it does not take the underlying data characteristics into consideration. It is thus hard to reason about the values for $N^i$ and $Cost^i$, and it is difficult to guarantee that the data distribution across different partitions will be same. Furthermore, the variance in resulting performance from multiple runs of the same algorithm may also be very high. We address these issues by developing a partitioning method that aims to ensure that each partition has properties similar to that of the entire data set.

Our data partitioning method makes use of tree hashing method that is developed earlier in Section 3.1. The entire process is summarized as Algorithm 16. It consists of three main steps. In the first step, all database trees are hashed to their respective signatures (or sketches) using Algorithm 7. These signatures are clustered to form strata in Line 5. Such stratification is similar to traditional data clustering, and it is performed by making use of the similarity measure defined in Equation 3.1.3. Individual partitions are then formed in the third step by looping through all strata (Lines 6-13). The data records from each strata are spread across $|P|$ partitions. The partitioning is complete and non-overlapping i.e., every record is assigned exactly to a single partition.

There exist a variety of methods to cluster the set of all signatures into strata (Line 5). Broder et al have proposed a method in the context of clustering web documents [36]. This method outputs a set of tuples for each minhash value in the signatures, and performs an external merge sort on the tuples to compute the number of minhashes that are common across all pairs of documents. A user-defined similarity threshold can then be enforced to obtain clusters of documents.

Stratification of signatures can also be done via modified versions of approximate nearest neighbor algorithms [102]. The set of all signatures $Sigs$ produced in Line 3 can be thought of a matrix where each row corresponds to a tree signature of fixed size, let that size be $K$. Each signature is first divided into $B$ bands of size $R$, i.e., $K = B \times R$. The signatures are then inserted into $B$ hash tables. For each band $b$, the corresponding elements from the signature are hashed into hash table $H_b$. For
Algorithm 16 Stratified partitioning

**Input:** database $D$, number of compute nodes $|P|$, tree signature function $sf$

1: $Sigs \leftarrow \emptyset$
2: for each tree $T$ in $D$ do
3: \hspace{1em} $Sigs(T) =$ compute signature of $T$ using $sf$ \{see Algorithm 7\}
4: end for
5: $Strata \leftarrow stratify (Sigs)$ \{see Algorithm 17\}
6: for $i = 1$ to $|P|$ do
7: \hspace{1em} $partition \leftarrow \emptyset$
8: \hspace{2em} for each strata $s$ in $Strata$ do
9: \hspace{3em} Select $\left\lfloor \frac{|s|}{|P|} \right\rfloor$ records from $s$ \{no record is selected more than once from $s$\}
10: \hspace{3em} Add selected records to $partition$
11: end for
12: Assign $partition$ to compute node $P_i$
13: end for

example, let the portion of signature for band $b$ be $(x_1, x_2, \cdots, x_R)$. It is hashed using the following formula:

$$h((x_1, x_2, \cdots, x_R)) = (a_1 x_1 + \cdots + a_R x_R \mod P) \mod M$$

where $P$ is a large prime, $M$ is the hash table size, and $a_1 \cdots a_R \in \mathbb{Z}_P^*$. Each resulting hash table denotes a specific partitioning arrangement of the data set. A union-find style algorithm [101] can then be employed to combine different arrangements given by $B$ hash tables. In the final result, all signatures that hash at least once to the same bucket are grouped together into one strata.

An alternative method to construct strata from signatures is to use locality sensitive hashing (LSH) [94, 102]. This method constructs a LSH-signature from each tree signature, and outputs similar pairs of trees according to the constructed LSH signatures. The psuedocode for this method is shown as Algorithm 17. A similar method that does not use an explicit two-dimensional matrix is proposed by Haveliwala et al [94].

**Empirical Evaluation**

We evaluate our data partitioning methods by running a parallel tree mining workload using Treebank data set introduced in Section 2.3. The data set is distributed across 10 different compute nodes, and then MCT algorithm is run on each partition.
Algorithm 17 Stratification via LSH (stratify)

1: define a two-dimensional matrix $mat$ with dimensions $nxn$, where $n = |D|$
2: for $l = 1$ to $L$ do
3:    generate $m$ distinct random indices, each from the range $1, \ldots, K$
4:    for $i = 1$ to $n$ do
5:       $LSH_i \leftarrow$ concatenation of minhash values from $Sigs(i)$ pointed by selected $m$ indices
6:       for $j = 1$ to $i$ do
7:          $LSH_j \leftarrow$ concatenation of minhash values from $Sigs(i)$ pointed by selected $m$ indices
8:          if $LSH_i = LSH_j$ then
9:             increment $mat[i][j]$ and $mat[j][i]$
10:        end if
11:    end for
12: end for
13: end for
14: Run a union-find style algorithm [101] to group trees which match on at least $L_u$ signatures, where $L_u < L$ is a user-defined threshold

The comparison between random and our stratified partitioning methods is shown in Figure 5.16. The number of patterns that must be re-evaluated (Equation 5.2.2) and the number of partitions that must be accessed (Equation 5.2.3) in the join phase decrease with increasing support threshold. Since the stratified partitioning ensures that every partition is a valid representation of the data set, it consistently provides better performance over random strategy, for all support values. While the random strategy makes 1865 accesses to partitions during the join phase at $\text{minsup} = 58\%$, our method makes only 1516 accesses. This result demonstrates the fact that the stratified method produces a smaller number of false positive patterns that require a re-evaluation. The difference between two methods however reduces as one increases the minimum support – total number of actual patterns itself is very small when the support is high.

We now break down the overall performance by measuring the cost at the level of individual partitions. Figure 5.17 plots the values for $Cost^i$ for all 10 partitions ($1 \leq i \leq 10$). Note that the range of y-axis is same in both the charts. Evidently, the variance among $Cost^i$ values is high when the data set is distributed using a random strategy. The variance is small in case of a stratified method because the all resulting partitions exhibit similar properties and possess similar data distribution. Most of
Figure 5.16: Performance of data partitioning methods

Figure 5.17: Variance of cost metric across different partitions (\(\text{minsup} = 58\%\))

<table>
<thead>
<tr>
<th>(\text{minsup})</th>
<th>Random</th>
<th>Stratified</th>
</tr>
</thead>
<tbody>
<tr>
<td>86%</td>
<td>1.35</td>
<td>0.87</td>
</tr>
<tr>
<td>77%</td>
<td>7.79</td>
<td>4.25</td>
</tr>
<tr>
<td>67%</td>
<td>20.89</td>
<td>13.88</td>
</tr>
<tr>
<td>58%</td>
<td>70.90</td>
<td>45.40</td>
</tr>
</tbody>
</table>

Table 5.5: Standard deviation among \(\text{Cost}^i\) values for different partitions
the variance in this case is contributed from those strata whose size is less than $|P|$. Such small-sized strata can not be distributed equally across all partitions, thereby introducing some variance. If the number of records in every stratum is greater than $|P|$ then the variance among $Cost^i$ values would be much smaller. Table 5.5 shows the standard deviation among the values of $Cost^i$ at various support levels.

Overall, stratified partitioning produces partitions whose data distribution is similar to that of entire data set. As a result, it reduces the number of false positive patterns that must be re-evaluated during the join phase. The variance in results from different partitions is also reduced, thereby guaranteeing a better load balance and good scalability.

Figure 5.18: Evaluation of stratified sampling that is powered by our tree hashing methods
Sampling Tree-Structured Data

A popular data-driven approach to improve the efficiency of large scale algorithms is to rely on sampling – instead of running the algorithm on the entire data set, it is run only on a selected sample of data. The effectiveness of this approach depends mainly on the quality of the extracted sample. A good sample must be a true representative of the data set with similar characteristics. Generating good samples from structured data is however non-trivial. One must not only sample from the node content, the structure among different nodes should also be taken into account.

The data partitioning method in Algorithm 16 can easily be modified to generate stratified samples from a given data set. Assume that we are interested in extracting a $s\%$ sample from the data set. The tree signatures and subsequently strata are produced in a similar manner to stratified partitioning (Lines 1-5 in Algorithm 16). Once the strata are created, a $s\%$ sample can be constructed by simply extracting $s\%$ records from each strata. Each partition produced from our stratified strategy is in fact represents a $\frac{100}{|P|}\%$ sample from the data set. The number of records extracted from each strata is proportional to its size. As we extract records from every strata, the resulting sample will be a true representative of the data set.

We evaluate the effectiveness of this stratified sampling by using it in the context of frequent pattern mining. We compare the set of patterns generated from the sample with the actual set of patterns obtained from the entire data set. We compute the $F$-measure for the sample by evaluating the precision and recall. The $F$-measure is calculated as the fraction $\frac{2 \cdot \text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}}$. We perform experiments on two data sets, Cslogs and Treebank (see Section 2.3).

The first and third charts in Figure 5.18 compare the effectiveness of stratified sampling when compared to random sampling in terms of $F$-measure on Cslogs and Treebank datasets, respectively. The results presented in the figures are averaged over 5 different runs. As expected, stratified sampling using our hashing algorithm, outperforms random sampling from a qualitative perspective across the board for multiple sample sizes. Additionally, since stratified sampling acts upon carefully grouped data records using our proposed hashing schemes, the variance in $F$-measure is very small when compared to that of random sampling – see second and fourth charts in the figure. In other words, the samples obtained from our method are more robust when compared to random samples. The performance does not vary as much across multiple runs of the same algorithm. High accuracy and small variance of our
algorithms make them attractive choices for progressive sampling, where we can use them to quickly find the nature of the learning curve (accuracy vs. sample size) and thereby determine best sample size [145]. The results presented here are representative, we observed similar results for other support levels for both the datasets. For example on Cslogs at 1% support, the variance in F-measure for randomly generated sample was, on average, about 10 times higher than that observed when our stratified sampling is used.

5.2.3 Distributed Tree Indexing and Querying

We now consider the case of distributed tree indexing where we are interested in querying a database that is partitioned across multiple compute nodes in the cluster. As shown in Figure 5.15, once the dataset is partitioned, the query processing involves running a sequential algorithm on each database partition. Note that the index is built on each partition separately using the local data. The set of query matches obtained from all partitions is then combined and returned to the user. The joining phase in this context involves a simple union operator. As in the case of frequent subtree mining in previous section, the main challenge is in designing an effective data partitioning strategy.

The method that is used to partition the database affects different performance aspects of distributed query matching. Recall that the main intuition behind our stratified partitioning in Section 5.2.2 is to make each partition a representative sample of the entire data set. One can employ the same method can be used in the context of distributed querying. For a given query, stratified strategy ensures that each partition produces a similar number query matches. In other words, the work associated with a given query is equally distributed across all partitions. While this method ensures a good load balance, it does not allow any concurrent query processing because each query spends roughly equal amount of time on every compute node.

If the goal is to support concurrency among queries then a data partitioning method must be developed in such a way that each query is answered by only a few compute nodes. Such a behavior can be achieved by keeping all similar-structured trees together on one partition. If a query is selective then it is likely that its matching trees are structurally similar. Since all closely-related trees are kept together, the selective queries are answered by a very few number of partitions. Since different
partitions host data with different characteristics, one can overlap the processing of multiple queries—each query is targeted to a different set of partitions.

In order to keep all structurally similar trees together on a single (or a few) partition, the strata obtained from Algorithm 17 are directly mapped onto partitions. If the number of compute nodes is smaller than total number of strata then multiple strata can be packed into a single partition, and assigned to a compute node. One can also enforce further constraints such as equal-sized partitions.

Assume that the given database that is denoted by $D = \{T_1, \cdots, T_{|D|}\}$ is clustered into different strata. Let $\{S_1, S_2, \cdots, S_{|S|}\}$ be the set of all strata obtained from $D$. Since the stratification is complete and non-overlapping, we have two conditions: $S_1 \cap \cdots \cap S_{|S|} = \emptyset$; and $S_1 \cup \cdots \cup S_{|S|} = D$. Let $Q_i$ be a given query, and the set of all its matching trees in the entire database be $M_i$. Clearly, $M_i \subseteq D$. Let $I(.)$ be an Indicator function that takes a set as its argument. The function evaluates to zero if the argument is an empty set, and to a value 1 otherwise.

A given query may match with trees from multiple strata. We can compute this quantity by finding the number of strata that need to be touched to answer the given query, as described below:

$$N(Q_i) = \sum_{j=1}^{|S|} I(M_i \cap S_j) \quad \quad (5.2.4)$$

It measures the number of strata whose intersection with the matching set $M_i$ is non-empty. We would like this number to be small, in order to guarantee better concurrency.

We now evaluate two different stratification methods—Algorithm 17 and a random strategy. For fair comparison, we ensure that the size distribution of strata obtained from both methods is exactly the same. Table 5.6 compares these two methods by evaluating measure from Equation 5.2.4. The experiment is run on Treebank data set [4] with a query workload of 15 queries. For each query $Q_i$ in the first column, the total number of matching trees ($|M_i|$) in the entire database is shown in the second column. The third column in the Table 5.6 shows measured values of $N(Q_i)$ for strata obtained from Algorithm 17. Similarly, the fourth column shows $N(Q_i)$ values obtained from a random stratification method.

It is clear from Table 5.6 that our hash-signature-based stratification method performs better than a random strategy. As similarly structured trees are placed together in a single strata, $N(Q_i)$ values are smaller when the data is clustered using
Algorithm 17. For example, the query $Q_4$ has a total 830 matching trees, which are distributed across 26 different strata produced from the random strategy. Whereas using our method, all the 830 trees are concentrated in only 5 different strata. In other words, one must touch only these 5 strata while answering the query $Q_4$. The other strata can be used to serve other types of queries, concurrently.

Overall, the method in which the data is partitioned across different cluster nodes is an important factor in driving the performance of distributed querying approach. Good load balance can be achieved by employing the stratified partitioning method from Algorithm 16. In contrast, we can achieve better concurrency if we map strata obtained from Algorithm 17 directly onto cluster nodes.

### 5.2.4 Data Placement Service for Distributed Clusters

As shown in previous sections, effective data placement strategies can enhance the performance of data-intensive applications implemented on high end computing clusters. We now present a distributed data placement framework that is geared towards tree- and graph-structured data sets. Target applications include the placement of a single large graph (e.g. Web graph), a single large tree (e.g. large XML file), a forest of graphs or trees (e.g. XML database) and other specialized graph data types – bi-partite (query-click graphs), directed acyclic graphs etc.
The high-level framework of our placement service is shown in Figure 5.19. The entire service consists of three main steps: data transformation, hash-based data partitioning, and data placement. The general design philosophy in the first step is to first decompose the structured datum into a (multi-)set of pivotal elements which at some level effectively capture the important features of the datum. The exact decomposition method can be tuned based on the requirements of the end application.

In the second step, the resulting pivot (multi-)sets are fed to hash functions to produce fixed size signatures. The hash functions must try to respect the relationships among data items and map similar items to hash values with high similarity. Our framework supports several such hash functions – locality sensitive hashing [102], approximate hashing [40], min-wise hashing [35], or other suitably developed variants. The signatures obtained from the hash function can then be used in intelligently partitioning the data set. As an example, consider Algorithm 16 that converts tree signatures into partitions in such a way that each partition has similar properties to the entire data set.

Once partitions or strata of data entities have been obtained the final step for data placement involves mapping these partitions to physical nodes on a distributed cluster. One must address several systematic issues such as data load balance or skew, affinities among partitions (i.e. if two or more partitions are inter-related should they be placed together) and any application-specific constraints for placement. Our framework supports several methods for placement such as random, round-robin, sorted (e.g., URL based sorting for web applications), and bitonic scheduling [147], to name a few. For example, in the case of distributed indexing application from
Section 5.2.3, the strata obtained from Algorithm 17 can be placed on compute nodes using a round-robin fashion. One can also enforce further constraints such as equal-sized partitions. Such a strategy keeps all similarly structured trees on a single partition, which is useful for the indexing application.

5.3 Concluding Discussion

Memory optimizations: Improving the data locality (spatial or temporal) continues to be important, but in addition, bandwidth must also be considered when designing data-intensive algorithms for emerging CMPs. The traditional trade-off between time and space, and its implications for parallelism need to be examined carefully in this light. All our memory optimizations target the above challenges. Each optimization may not amount to a significant improvement on its own but the specific orchestration applied when combining them yields significant savings – L1 misses reduced by up to 1,442 times, memory footprints reduced by a factor of 366, bandwidth pressure reduced significantly by making uniform small-sized accesses to main memory, and overall run time reduced by a factor of four on sequential execution.

These optimizations have a broader applicability in many domains. We briefly mention some of the important ones. NOEM essentially improves the mining by computing the required matches on-demand instead of storing extra state. Such a technique is evidently useful in mining other types of patterns such as graphs, sequences. For example, in the case of gSpan [200], a popular graph mining technique, one can make use of a fast subgraph matching algorithm to enumerate matches before extensions of a subgraph are found. Techniques similar to NOEM are also useful in searching bioinformatics databases [188] and XML repositories [211].

The tree matching optimizations (LF, DOM, SIMUL), though appear to be specific to Prüfer sequences, have a general purpose utility. They can easily be adapted to algorithms which rely on a depth first encoding. They can also be used to reduce the overhead in other dynamic programming based approaches – mining time series [29]; establishing maximal matchings between glycan structures [18]; code generation techniques [8]; (multiple) sequence alignment [138]; and computing consensus and agreement of phylogenetic trees [169]. This list by no means is an exhaustive one.

Computation chunking captures the general notion of breaking the computation
into smaller pieces so that they can be handled efficiently. Such an approach have a
general purpose utility in database query processing [154] and also in mining other
structures such as graphs, DAGs, induced subtrees, and sequences. In gSpan [200],
instead of finding children for each occurrence of a subgraph separately, one can
group a set of occurrences and find the one-edge growths collectively. They can also
potentially be leveraged for answering reachability queries on directed graphs.

Applications seldom realize peak memory bandwidth numbers quoted in product
specifications [192]. These numbers often assume that the references are uniformly
distributed across the memory system, and avoid conflicts for banks and front side
bus – a rare case, in practice. It is thus very important to look at achieved bandwidth,
especially for CMPs where the memory bus is shared among all cores. The method
from Section 5.1.2, though an approximate one, provides an easy and quick way to
study the memory behavior of algorithms at individual core level. We believe that
this light-weight mechanism to measure the bandwidth is widely applicable to several
other data mining and database applications [154].

**Scheduling service:** With regards to task scheduling, algorithms that can adapt
and mold are essential to achieve performance commensurate with the number of cores
in emerging CMP systems. Coarse-grained strategies are usually not sufficient since
systemic, parametric and data-driven constraints make the workload estimation a
challenging task. In such scenarios the ability of an algorithm to adaptively modulate
between coarse grained and fine grained strategies is critical to parallel efficiency. In
fact how much an algorithm can adapt essentially dictates when the performance
plateau is reached, as we observed in our study. Our adaptive strategy demonstrated
near-perfect parallel efficiency on both a recent CMP and a modern SMP system.

A key outcome here beyond the specific tree mining algorithm is the realization of
a general purpose scheduling service that supports the development of adaptive and
moldable algorithms for database and mining tasks. For instance, one can parallelize
a graph mining algorithm like gSpan [200] by simply defining the task descriptors
and appropriate job spawning conditions. Rest of the details like job scheduling,
synchronization, and thread management are transparently taken care by our service.
This service is easily applicable to many other pattern mining algorithms because
they all employ a pattern-growth approach and traverse the search space in depth
first order. They are also applicable to other data mining tasks like classification
using decision trees [209].

**Distributed data placement:** Effective data placement is extremely important
to perform large scale data analysis on distributed cluster systems. Data placement can have a significant impact in localizing the computation, in minimizing synchronization (communication) costs, in enhancing reliability (via strategic replication policies), and in ensuring a balanced workload or enhancing the available bandwidth from massive storage devices (e.g. disk arrays). Our hash-based data placement framework is shown to be useful for various applications, including frequent subtree mining (see Section 5.2.2), tree indexing and querying (see Section 5.2.3), and Web graph compression [41].
In this dissertation, we investigated issues related to mining and managing a specific class of semi-structured data sets where the data is modeled using hierarchical tree objects. For efficient realizations, we argue that *algorithmic optimizations as well as architecture-conscious solutions that leverage the features of underlying computer systems are essential to obtain efficiency that is commensurate with the data complexity*. We designed novel algorithms and heuristics for the following three problems – *mining* common substructures from a forest of rooted trees; *indexing* tree data repositories; and developing effective *hash functions* for tree-structured data. In addition to devising new algorithms, we also developed architecture-specific optimizations in the context of uniprocessor systems, multicore systems, and cluster systems for efficient data analysis and management.

In terms of novel algorithms for semi-structured data mining and management, we made the following contributions:

- We proposed a *concise injective sequential representation* for trees that embodies the complete information via two *non-redundant and mutually exclusive* sequences. We presented two representations, Prüfer sequences and depth first sequences, which rely on post-order and pre-order tree traversals, respectively. Unlike extant methods, we designed algorithms that seamlessly transform the operations on pointer-based tree-structured data into operations on sequences so that the data locality and the instruction level parallelism are improved.

- We presented two sequence-based algorithms (*TRIPS* and *TIDES*) for mining frequent substructures from a forest of rooted ordered trees. Existing mining algorithms use large pointer-based meta-data structures to reduce the time spent in repeated executions of subtree isomorphism checks. We showed that such strategies are highly memory intensive and exhibit poor data locality.
In contrast, *TRIPS* and *TIDES* leverage our injective sequence representations in realizing a pattern-growth mechanism for enumerating the set of all frequent subtrees. The candidate patterns are generated by traversing the search space via *simple and fast operations on sequences*. The subtree matching problem is recasted into a much simpler problem of *subsequence matching*, wherein appropriate structural constraints are embedded deep into the matching process. We showed that our algorithms exhibit very *good data locality* as most computations are performed on linear array-based data structures.

- We proposed a mechanism called *LCS-TRIM* to index and query XML data repositories. Existing tree indexing techniques rely upon very large disk-based index structures and employ expensive query processing methods. Such techniques provide poor response times, and as a result, they are not scalable. In contrast, *LCS-TRIM* relies on a *sequence-based holistic subtree matching* algorithm to retrieve all query matches. This algorithm is complemented with a *small* and *tunable* inverted index structure. While the concise sequential representations enable a cache-conscious design, the small index structure effectively prunes the search space. Through a detailed empirical analysis, we showed that our framework can improve the query retrieval times by up to three orders of magnitude, while reducing the footprint of the index structure by an order of magnitude. We also presented mechanisms that support the retrieval of approximate matches, and the ones that enable the user to specify constraints on the retrieved output and showed how they can be pushed deep into the retrieval process. We showed that the entire query processing mechanism can be made efficient by developing parallel shared-memory algorithms for *index construction, list intersection, and subtree matching*.

- We presented a high-level framework for designing hash functions that map complex variable-sized semi-structured data entities into fixed size hash values or signatures. To the best of our knowledge, this is the first approach that deals with hashing semi-structured data. Through a rigorous theoretical analysis, we showed that one can design universal hash functions for trees that map distinct trees to distinct hash values. The key highlight of our framework is that it allows the users to define their own hash functions as per the demands of end
application. We performed several application case studies (XML deduplication, frequent subtree mining, and phylogenetic analysis) to demonstrate the pertinence of our hash functions.

With respect to architecture-conscious solutions, we made the following contributions:

- We developed system-specific techniques for efficient data mining and data management on modern multicore systems. For such systems, improving the data locality (spatial or temporal) continues to be important, and in addition, we demonstrated that the limited bandwidth to main memory becomes a performance bottleneck as one increases the number of cores. We showed that a “memory-conscious design” wherein the algorithms trade memory for redundant computations is critical to address these challenges.

Specifically, we designed a dynamic programming based approach for efficient subtree matching. This approach, when coupled with memory-conscious optimizations, improved the run time and memory performance of existing tree pattern mining algorithms by several orders of magnitude while reducing the amount of data transferred over the memory bus. These optimizations have a broader applicability – in mining patterns of other complex data types such as DAGs and graphs; in establishing maximal matchings between glycan structures; in computing consensus and agreement of phylogenetic trees; and in answering reachability queries on graph data.

- We developed scalable strategies for parallelizing pattern mining workloads on multicore systems. These strategies automatically detect the load imbalance in the system and adaptively partitions the work to guarantee good parallel performance. We showed that coarse-grained strategies are not sufficient since systemic, parametric and data-driven constraints make the workload estimation a challenging task. Therefore, algorithms that expose and subsequently exploit fine-grain parallelism are essential to realize performance commensurate with the number of cores in a multicore system. Based on our load balancing strategies, we developed a general purpose task scheduling service that supports the development of adaptive and moldable techniques for parallel database and mining tasks. The service exposes a simple API to applications, and transparently takes care of issues such as job scheduling, synchronization, and thread management.
We designed a hash-based distributed framework for effective data placement on commodity cluster systems. This framework can enhance the performance of data-intensive applications by localizing the computations, minimizing communication overhead, and by improving the load balance. The usability of this framework is demonstrated by developing effective distributed algorithms for both pattern mining as well as database indexing.

6.1 Future Research Directions

Algorithms for Graph-structured Data:

There are several promising directions in which one can extend the work presented in this dissertation. Applications that produce and process more complex semi-structured data types such as directed acyclic graphs, undirected graphs, and bi-partite graphs are ubiquitous. Examples include social networks, software engineering, bioinformatics, communication networks, world wide web, to name a few. Data management and data analytic challenges posed by such complex data are humongous. There exists some work focusing on indexing graph databases [51, 132, 168, 201]. These methods primarily differ in the way they construct the set of features on which the index is built. In order to realize good performance, the features must be as discriminating as possible so that the search space is pruned effectively. Efforts must also be spent in keeping the total size of index in acceptable limits. Simultaneously, heuristics for efficient graph- and subgraph- matching must be employed to select the final set of matches from the candidate set returned by the index.

The data produced in several of these applications is often highly dynamic with constant topological changes. Hosting and providing efficient access to these huge dynamic data sets is a significant challenge. As we highlighted in our dissertation, developing traditional data-driven strategies like sampling and hashing for complex graphical data types is an open problem. One must design methods that not only consider the content such as node labels but also the complex structure that is present in graph data.

Furthermore, most algorithms operating on graphs are computationally very expensive – e.g., the fundamental subgraph isomorphism is NP-complete. Therefore, one must investigate both approximate and heuristic solutions while handling graph data. Similar to the techniques presented in this thesis, adaptive strategies that strike a balance between algorithmic optimizations and architecture-conscious techniques.
Since both the data complexity and the algorithmic computational complexity is high for graph data, leveraging the features of underlying computing systems can be very beneficial.

**Mining Software Engineering Data Repositories:**

Along with generic algorithms and architecture-conscious techniques, domain-specific methods and optimizations are equally important. Applied data mining is an exciting research direction that can open up a plenty of open research problems. For instance, data mining techniques can be effectively leveraged towards improved software performance, software correctness, software reliability, and software debugging. There exists a vast amount of semi-structured data such as revision histories, static and dynamic call graphs, data flow graphs, execution traces, and bug reports. This data contains a wealth of information that can be exploited via data mining tools for designing effective compile time and run time adaptations and optimizations. Few example applications are detailed below:

- Massive call graph data can be analyzed to identify frequently invoked set of procedures, and the specific structure in which they are called from each other. The identified substructures provide insights that can be leveraged either by software developers or by automatic code generation tools to improve the software performance (e.g., by *inlining*).

- Mining rare patterns from such call graphs is useful in detecting anomalous execution or communication patterns in the software. Similar techniques applied on data flow graphs can help us understand the complex linked structure among different objects in the memory (e.g., heap). These patterns also can point us to erroneous scenarios such as bugs and data race conditions [124, 197].

- Interesting patterns extracted from revision history data can help in better understanding of software evolution [127]. Similarly, provenance analysis methods that can trace back the errors through several versions of the same code can be very useful for software development and for software understanding. Such methods must be supported by mechanisms that can store and effectively manage such provenance information.

- Knowledge discovered from mining data flow characteristics in the software can be used for developing effective runtime optimizations and adaptations [81].
The key challenge in designing such applied data mining methods is to identify and subsequently incorporate the domain knowledge into the mining process. Furthermore, determining a set of common abstract operators from diverse application case studies can be used to design general purpose frameworks that are beneficial to a host of algorithms.

**Energy-efficient Algorithms for Heterogeneous Architectures:**

Modern computer systems are becoming extremely complex in the pursuit of higher performance. Heterogeneous architectures hold a promising future. Such heterogeneous computer architectures expose sub-systems with different processing and memory capabilities under the same hood. The scale at which heterogeneous systems operate is expected to increase in the future. For instance, AMD Fusion family Accelerated Processing Units (APUs) that combine CPU and GPU are expected in near future [14].

Such hybrid systems are likely to be beneficial for knowledge discovery algorithms, especially because most data mining workloads often require several different forms of parallelization. For example, in the case of pattern mining, a task-parallel strategy is useful to process multiple patterns simultaneously, and a data-parallel approach can be leveraged to “accelerate” different parts of the computation (e.g., finding the frequency of a pattern). The most important challenge here is in effective work distribution. One must clearly understand the processing capabilities, memory availability, communication needs of each subsystem, and distribute the work onto different sub-systems accordingly. Dynamic strategies as opposed to static distribution schemes are critical in effectively exploiting the heterogeneity. Striking a balance between specific features of these complex architectures and the particular needs of data mining algorithms is the key to efficiency and scalability.

Future heterogeneous architectures are also expected to deliver better “performance per watt” – the rate of computation that can be delivered for every watt of power consumed. With the rapid growth in power and cooling requirements of large scale data analytic systems, heterogeneous architectures are likely to play an important role in the development of energy-efficient data mining algorithms. There are two main challenges:

- Given a set of user-defined budgetary constraints, the main challenge is to perform adaptive system resource provisioning so that the application’s performance can be tuned according to the given constraints. For example, power
consumption can be reduced by switching off or by reducing the frequencies of certain cores in a multicore system, while making sure that the increased response times are within the specified budget constraints.

- Another challenge is to identify and subsequently schedule different portions of computation onto suitable subsystems. Such a scheduling strategy must be driven by the energy and performance portfolios of underlying subsystems.

In summary, the challenges in data analysis in the years ahead will not only be in its unprecedented scale but also in its complexity. Simultaneously, modern computer architectures have ushered in new computing paradigms that when effectively used, can bring about performance improvements of several orders of magnitude. It is thus important to design high performance algorithms that leverage commodity cluster and heterogeneous architectures to build adaptive, scalable solutions that address the challenges of data in both, its complexity and scale.
BIBLIOGRAPHY


[92] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1–12, 2000.


193


Canonical XML. [http://www.w3.org/TR/xml-c14n](http://www.w3.org/TR/xml-c14n).


