Declarative Querying For Biological Sequences

by

Sandeep Tata

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:

Associate Professor Jignesh M. Patel, Chair
Professor Hosagrahar V. Jagadish
Professor David States
Assistant Professor Martin J. Strauss
Assistant Professor Peter J. Woolf

To the people who taught me to strive, smile, share, and love.

ACKNOWLEDGEMENTS

I have received a lot of help from different people during the course of the last five years at Michigan. Without their help, this dissertation would have been quite impossible. First, I would like to thank my advisor Jignesh. He helped me strike the magical balance between learning, research, work, and play. He was supportive of even my wildest ideas and allowed me the freedom to explore and learn. Graduate school wouldn't have been so much fun without his support and guidance.

I had the good fortune of working with Rich Hankins early in the program. Rich introduced me to refactoring and several other ideas that quickly allowed me to be more productive. I learned much from Rich and am very grateful to him for the encouragement and mentoring he provided.

My colleagues and friends in the database lab made it easy to look forward to going to in work every day. I am thankful to Jason Chen, Magesh Jayapandian, You Jung Kim, Michael Morse, Yuanyuan Tian, Adriane Chapman, Yun Yao Li, Bin Liu, Arnab Nandi, Stelios Paparizos, and Cong Yu.

Finally, I am deeply grateful to my friends and family for making even the difficult times feel like fun!

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

ABSTRACT

Life science research labs today manage increasing volumes of sequence data. Much of the data management and querying today is accomplished procedurally using Perl, Python, or Java programs that integrate data from different sources and query tools. The dangers of this procedural approach are well known to the database community– a) severe limitations on the ability to rapidly express queries and b) inefficient query plans due to the lack of sophisticated optimization tools. This situation is likely to get worse with advances in high-throughput technologies that make it easier to quickly produce vast amounts of sequence data. The need for a declarative and efficient system to manage and query biological sequence data is urgent. To address this need, we designed the Periscope/SQ system. Periscope/SQ extends current relational systems to enable sophisticated queries on sequence data and can optimize and execute these queries efficiently.

This thesis describes the problems that need to be solved to make it possible to build the Periscope/SQ system. First, we describe the algebraic framework which forms the backbone of Periscope/SQ. Second, we describe algorithms to construct large scale suffix tree indexes for efficiently answering sequence queries. Third, we describe techniques for selectivity estimation and optimization in the context of queries over biological sequences. Next, we demonstrate how some of the techniques developed for Periscope/SQ can be applied to produce a powerful mining algorithm that we call FLAME. Finally, we describe GeneFinder, a biological application built on top of Periscope/SQ. GeneFinder is currently being used to predict the targets of transcription factors.

Today, genomic and proteomic sequences are the most abundantly available source of high-quality biological data. By making it possible to declaratively and efficiently query vast amount of sequence data, Periscope/SQ opens the door to vast improvements in the pace of bioinformatics research.

CHAPTER I

Introduction

The life sciences community today faces the same problem that the business world faced over 25 years ago. They are generating increasingly large volumes of data that they want to manage and query in sophisticated ways. However, existing querying techniques employ procedural methods, with life sciences laboratories around the world using custom Perl, Python, or JAVA programs for posing and evaluating complex queries. The perils of using a procedural querying paradigm are well known to a database audience, namely a) severely limiting the ability of the scientist to rapidly express complex queries, and b) often resulting in very inefficient query plans as sophisticated query optimization and evaluation methods are not employed. However, existing database products do not have adequate support for sophisticated querying on biological data sets. This is unfortunate as new discoveries in modern life sciences are strongly driven by analysis of biological datasets. Not surprisingly, there is a growing and urgent need for a system that can support complex declarative and efficient querying on biological datasets.

There are several large databases worldwide that store protein and DNA sequence information. (DNA can be abstractly thought of as a sequence over an alphabet of size four: $\{$A,C,G,T$\}$. Proteins can be represented as sequences over the amino acid alphabet, which is of size twenty. Proteins also have a secondary structure which refers to the local geomet-

ric folding. This too is represented as a sequence over the secondary structure alphabet of size three: alpha helix, beta sheet, and loops.) Some of these databases are growing very fast. For instance, GenBank, a repository for genetic information has been doubling every 16 months [63] – a rate faster than Moore's law! Protein databases, such as PDB [65] and PIR [10, 169] have also grown rapidly in the last few years. The growing sizes of the databases exacerbates the current deficiency in querying methods.

In this thesis, we describe database methods that are required to support declarative and efficient analysis of sequences in an object-relational database system. We have developed these techniques as part of the Periscope/SQ system, which is part of a larger project called Periscope that aims to develop database methods for declarative and efficient querying of all biological data such as graphs, structures, expression data, etc. Periscope/SQ is the sub-system that deals with sequence data. In this thesis, we will focus on Periscope/SQ.

To address the need for easy and efficient querying mechanisms for sequences, we first propose an algebra [145] to express such queries over sequences. This algebra (called PiQA) provides a rich set of operators that permit sophisticated querying on both the primary and secondary structures of protein, and on DNA sequences. In addition, procedures used by existing tools like BLAST can also be expressed in PiQA. PiQA is also the basis for PiQL, an extension to SQL that allows us to declaratively express complex queries over sequence data. The expressive power of PiQA allows us to easily express queries that would be extremely awkward and difficult to express in a plain relational database. The details of PiQA are described in Chapter II.

A data structure that is extremely versatile and useful for evaluating a wide variety of queries on sequence datasets is the suffix tree. The suffix tree is especially useful for finding exact and approximate string matches, and to find repeating patterns. However, methods for constructing suffix trees are often very time-consuming, especially for suffix

trees that are large and do not fit in the available main memory. Even when the suffix tree fits in memory, it turns out that the processor cache behavior of theoretically optimal suffix tree construction methods is poor, resulting in poor performance.

In Chapter III, we explore suffix tree construction algorithms over a wide spectrum of data sources and sizes. We show that on modern processors, a cache-efficient algorithm with $O(n^2)$ worst-case complexity outperforms popular linear time algorithms like Ukkonen and McCreight, even for in-memory construction. For larger datasets, the disk I/O requirement quickly becomes the bottleneck in each algorithm's performance. To address this problem, we describe two approaches. First, we present a buffer management strategy for the $O(n^2)$ algorithm. The resulting new algorithm, which we call **TDD**, scales to sizes much larger than have been previously described in literature. This approach far outperforms the best known disk-based construction methods. Second, we present a new disk-based suffix tree construction algorithm that is based on a sort-merge paradigm, and show that for constructing very large suffix trees with very little resources, this algorithm is more efficient than TDD. The TDD algorithm enables Periscope/SQ to use the suffix tree index to efficiently query large sequence datasets.

Chapter IV discusses estimation techniques, operators, and optimization algorithms used in Periscope/SQ. We describe PiQL, the extension of SQL that can express PiQA queries. We introduce new physical operators and support for suffix tree indexes in the database. The suffix trees add the option of a very efficient access path for many sequence queries. We describe a novel approach to estimating the selectivity of string predicates using a Symmetric Markovian Summary. We also describe a simple, yet highly effective algorithm to optimize sequence queries. We demonstrate that Periscope/SQ is efficient for different kinds of queries and using a real world application in eye genetics, we show that we can achieve speedup of *two orders of magnitude* over existing procedural methods.

Complex sequence analysis goes beyond just querying sequence databases. Mining sequences for interesting patterns is an extremely important and difficult problem. Existing database sequence mining algorithms mostly focus on mining for subsequences. However, for many emerging applications, the subsequence model is inadequate for detecting interesting patterns. Domains that involve medical time series data, financial time series data, biological sequences, etc. often require other more complex models. For instance, mining DNA sequences to identify regulatory regions requires finding *frequent approximate substrings*. The approximate substring model better accommodates the notion of a noisy pattern, and is therefore better suited than the subsequence model for many new applications.

To facilitate mining of different datasets, we present a powerful new model for approximate pattern mining. In Chapter V, we show that this model can be used to capture the notion of an approximate match for a variety of different applications. We present a novel, suffix tree based pattern mining algorithm called FLAME (**FL**exible and **A**ccurate **M**otif D**E**tector). Through an extensive empirical evaluation on both real and synthetic datasets from different domains, we demonstrate that FLAME is a fast, accurate, and scalable method for discovering hidden patterns in large sequence databases.

Periscope/SQ provides the infrastructure to develop sophisticated sequence processing applications. As a demonstration, we built an application called GeneFinder on top of Periscope/SQ. GeneFinder tackles the difficult problem of predicting target genes for transcription factors where the binding signature of the factor is known. GeneFinder takes advantage of Periscope/SQ by combining sophisticated sequence predicates with several relational queries and is able to make high quality predictions. GeneFinder demonstrates the ease with which a declarative framework can be used to rapidly develop an application, which is also significantly faster in executing queries than existing methods. In

Chapter VI, we describe the problem in detail and show how the techniques developed in Periscope/SQ are crucial to GeneFinder.

The contributions in this thesis are summarized in Chapter VII along with our conclusions.

CHAPTER II

Algebra

2.1 Introduction

Recent years have seen an enormous explosion in the sizes and uses of biological data. Several nucleotide and protein sequence data sets are growing at an exponential rate, doubling roughly every 16 months [146]. In addition, the nature of the searches against these databases is also changing, and scientists today would like to ask more complex queries against these data sets. Database management tools have an important role to play in querying such biological datasets [37, 52]. This work focuses on one such aspect, namely the querying of protein data sets based on different structural attributes that describe each protein.

Proteins have the following four levels of structural organizations: primary, secondary, tertiary and quaternary structures. In this study, we focus on querying the primary and secondary structures. The primary structure is simply a linear sequence of amino acids residues that forms the protein. The secondary structure describes how the linear sequence of amino acids residues orients itself, or folds, in three-dimensional space. There are three basic types of folds: alpha-helices, beta-pleated sheets, and turns or loops. Knowledge of a proteins secondary structure has been shown to provide important insights into its evolutionary relationships, and hence its function.

Typically, biologists are interested in finding similarities between a sequenced protein and others in the database so that they can understand the function of the sequenced protein. For instance, given a protein, they may want to determine if similar proteins exist in other species, and may also want to determine the function of the protein. Or they might be interested in knowing if there are other proteins that have a different primary structure, but have a similar secondary structure. The secondary structure of the protein is crucial to understanding the function that the protein performs [13,23,124], and hence it is important to be able to understand it in relation to the primary structure.

2.1.1 The Problem

Today when scientists investigate a protein, they usually search databases of known proteins based on the primary sequence. The search is typically carried out using tools such as BLAST [66,126]. Such search tools essentially finds homologous matches. These search tools return approximate answers, and often a scientist may have to post-process these results, or run the search iteratively (as in PSI-BLAST). In addition, the scientist may query multiple data sets producing a large number of approximate matches that may feed into the next stage of their analysis. With protein queries, in many cases the next step after matching on the primary sequences may be to examine the protein of interest with the secondary structures of other known proteins in the database. The matching on the secondary structure is important as the functionality of proteins is strongly influenced by its actual folding pattern, and even proteins that are not close homologs may exhibit similar behavior if their folding patterns are similar.

As an example, a biologist might have just sequenced the hemoglobin protein in monkeys and may be interested in hemoglobin and other proteins in other species that are similar to this protein. Such comparisons are also useful in tracking evolutionary changes in the structure of the protein [78]. In certain other instances, when a biologist is trying to

find a protein that matches a certain structural fingerprint i.e. a certain spatial arrangement, they might have a secondary structure in mind and want to find proteins in the database that have a similar structure.

In many cases, these steps of querying on the primary and secondary structures may be repeated many times, and for many different databases. Often the iteration between these steps is driven by a manually coded program, which may need to be modified every time the underlying query changes. In addition, this entire process may needs to be carried out for each distinct experiment that is undertaken in a lab. A declarative query tool that permits querying on both the primary and secondary structures can not only reduce the time spent in posing such queries, but can also allow the biologist to pose more complex queries than are currently used today.

As an example, using currently existing tools one cannot express the following query in a straightforward way: *Match the given primary sequence of length 120, but ignore mismatches in the segment from positions 44 to 78 if it is on a loop in the secondary structure*.

In the next several sections we shall describe an algebra that can be used to query protein data sets based on both the primary and secondary structures. The algebra supports approximate matching, and also includes operators that allow extensions of two or more approximate matches to calculate a longer match. We believe that the algebra is expressive enough to express a large class of interesting queries on both the primary and secondary structures of proteins.

The motivations for developing such algebra are fairly obvious to a database audience. The algebra is a first step in providing a declarative query language-based interface to the user, rather than the more cumbersome procedural paradigm that is currently being used for queries across both primary and secondary structures. In addition, the algebra can also

be exploited by a query optimizer to produce efficient query plans.

The key contribution of this work is PiQA, a Protein Query Algebra that enables us to express queries on both the primary and secondary structures of proteins. To the best of our knowledge, PiQA is the first algebra that allows querying on both these structures. Using PiQA we also show how existing queries on only the primary structure can be expressed in this algebra. In addition, we also illustrate the use of the algebra in query optimization.

Though PiQA is basically designed to express queries on protein data, it can easily be applied to querying genetic data. We demonstrate the flexibility of PiQA by describing an application of PiQA in genetic research related to eye-disorders.

2.2 Related Work

Surprisingly, there is little previous work on developing an algebraic framework for querying biological data sets. Recently, Hammer and Schneider [80] proposed a long-term approach towards developing an algebra that abstracts several biological processes. Seshadri et al. [110, 111] describe techniques for querying sequence databases. However, these techniques primarily focus on aggregate-based analysis of sequences, and are not directly applicable for querying biological sequences, which often require pattern matching and approximate matching operators. There has been a lot of work in string matching, including proposals for a declarative language based on alignment calculus for strings [121]. However, these techniques can only be applied to primary sequence matching without approximations. Linguistic approaches have been used in [125] to predict gene structure from DNA sequences. However, such approaches do not generalize for other kinds of pattern based querying over sequences.

The algebraic constructs that we present in this chapter employ many of the constructs that have been developed for nested relational algebras [56, 99, 128, 133]. However, we

have been able to express the queries that we target using only a limited form of nesting, namely PNF relations [99], with only one level of nesting. Consequently, the optimizations, too, are simpler than those developed for more general forms of nesting [94,96,179].

A number of tools have been developed for searching on nucleotide sequences and primary protein sequences. The most frequently used tool in this category is the BLAST [126, 127] family of search programs. BLAST works in three steps: in the first step it finds all K-mers (strings from the alphabet of length K) that score above a certain threshold with some part of the query string. In the next step, it searches the database to find hits. In the final step, BLAST extends the hits according to certain heuristics and returns a list of high-scoring segment pairs. This score is a measure of similarity.

Searching based on the secondary structure of proteins has recently been examined by Hammel and Patel [95]. The authors define an intuitive query language that can be used to express queries on secondary structure and also developed techniques for evaluating and optimizing these queries.

2.3 The PiQA Algebra

The algebra that we describe is a multi-sorted algebra. The operators can be composed to specify complex queries involving both the primary structure and the secondary structure. We have formulated the algebra as an extension to relational algebra so that we still have the advantage of modeling data as relations. More precisely, the relations in our model are in the Partitioned Normal Form (PNF) [99]. (PNF relations restrict the class of general nested relations to guarantee the desirable property that a nest operation is the inverse of an unnest operation.)

We shall first describe all the types in the algebra, and then describe each of the operators, the types of their operands, and the type of the result. In the interest of space,

we do not describe the basic relational algebraic constructs [49], and extensions of these constructs to accommodate PNF relations [99].

2.3.1 Types

The basic types in the algebra are:

- Basic Scalar Types Integers, Characters etc.
- Hits and Matches
- Sequences
- Tuples
- Relations (sets)

Hit: A hit is basically a triple *(p,l,s)*. When specified together with some sequence, the hit *(p,l,s)* means that there is a *hit* at position *p* of length *l* with a score of *s* on the given sequence. For instance, suppose that *A* = (2,3,3) is a hit on the sequence *SEQ* = "TGGTTTAGGAGGTA". This hit refers to the "GGT" substring, which could have matched some query for a score of 3. This hit is shown in the original database sequence as "T**GGT**TTAGGAGGTA", with the hit portion highlighted in bold-face.

Match: A match is simply a set of hits. For example, consider the sequence *SEQ* = "TGGTTTAGGAGGTA", and a query to find "GGT" followed by a "GGA" within 10 symbols. A match for this query using an exact matching paradigm is *X*= {sid, (2,3,3), (8,3,2)}. "sid" is simply a sequence identifier that allows us to determine which sequence this match refers to. In this example, the match describes two hits in the data sequences as shown in bold-face in "T**GGT**TTA**GGA**GGTA".

Consider another example: **(CG2B, ((22, 7, 6), (44, 12, 9)))** is a match which could have been the result of some operation, and it means that the sequence referred to by CG2B

matched at position 22 and at position 44 with lengths of the matches being 7 and 12, and the scores being 6 and 9 respectively.

For ease of presentation, in some of the examples below, we represent the matches in an alternative form. In this alternative representation, the match is represented as a 4-tuple where the first component is an identifier. The remaining components of a match are sequences. The second component is a sequence of integers which refer to positions in a string, the third component is a sequence of integers which refer to the lengths of each of the matches whose positions are referred to by the previous sequence, and the fourth sequence in a match comprises the integers that represents the scores. In this alternative representation, the previous example would be expressed as: **(CG2B, (22, 44), (7, 12), (6, 9)**).

Several operators that we describe operate on sets of matches. We can view a set of matches as a nested relation with the first identifying component of the match serving as a key which functionally determines the other attributes in the relation. With this interpretation, we observe that these sets are in Partition Normal Form [99] .

Regular Expressions and Matches: A regular expression can be used to represent a match criterion. As in [95], a regular expression is expressed as a sequence of segment predicates, each of which must be matched to satisfy the entire expression. Each segment predicate is described by the type and the length of the segment. The type of the segment is drawn from the alphabet of the underlying sequence, and depends on the sequence being queried. For the protein secondary structures, the allowed segment types are h, e, and l, for the alpha-helices, beta-sheets, and loops, respectively. In addition we also add a fourth option, ?, which stands for a gap segment and allows scientists to represent regions of unimportance in a query. The length of the segment is specified using an upper bound and a lower bound, each of which could be 0. In addition the upper bound could be specified

RegExp	$\rightarrow$ {Segments}	
Segments	$\rightarrow$ Segment*	
Segment	$\rightarrow$ <type lb ub>	
type	$\rightarrow$ e|h|l|?	(for protein secondary structures)
type	$\rightarrow$ A|R|N|...|?	(for protein primary structures)
type	$\rightarrow$ A|C|G|T|?	(for nucleotide sequences)
lb	$\rightarrow$ any integer ≥ 0	
ub	$\rightarrow$ any integer $\geq 0 \mid \infty$	
Segment Constraint: lb $\leq$ ub		

Figure 2.1: Regular Expression Syntax

as ∞. Segment predicates over other structures are similar, except that the type used is set to the symbols in the underlying alphabet, with the addition of the "?" symbol. Formally, a regular expression is defined using the rules shown in Figure 2.1.

As an example, consider the following expression on a protein secondary structure: <e 3 5><? 0 ∞><l 7 7>. This regular expression matches all proteins that contain a beta-sheet of length 3 to 5 followed at some point by a loop of length 7.

Sets and Sequences: Sets and Sequences are well known types. Sequences have the standard position (or index) operator which allows access to an arbitrary element in the sequences. For example, the i^{th} position in a sequence S is simply accessed as S(i).

In this algebra, we only permit a sequence of the basic scalar types. That is, we may have a sequence of integers, characters, etc. But we do not have sequences on complex types such as relations. We do not define operations on sequences directly, but on the matches that have sequences as a part. Therefore, not having sequences of more complex types does not detract from the power of expressing queries that PiQA targets.

Since a string is merely a sequence of characters (over a relevant alphabet), we will use the terms string and character sequence interchangeably.

2.3.2 Operators

Match operator (*)

$$* : Set < strings > \times (str \cup regexp) \rightarrow Set < matches >$$

The match operator searches the set of strings (the left operand) to find substrings that approximately match the right operand, which could be a string or a regular expression specifying a set of strings. The result of this operation is a set of matches, each consisting of the identifier of the corresponding string, the match-positions and their lengths and scores. Symbolically, a match expression is of the form: $T * (strorregexp)$, where T is a set of strings. A common use of the match operator is to search on the primary structures using a string str, or searching on the secondary structures using a regular expression $regexp$.

The match operator is defined under *some* matching criterion, for instance PAM-30, PAM-70 or a BLOSUM62 matrix can be used to determine a match score between two primary protein structures. One may also choose to use an exact matching criterion or some other measure of approximate matching gapped, un-gapped, etc. for secondary structures (and even for primary structures). We will not deal in depth with specific matching criteria in this chapter. Though certain kinds of optimization may be possible if we know the matching criterion and scoring function used, in the interest of generality, our formulation will not be tied to any choice of matching criteria, except when explicitly specified.
Example:

Consider a protein table, **P**, with the following attributes: **id**- a unique identifier, **p**- a string representing the protein primary structure, and **s**- a string representing the protein secondary structure. In this example, we shall use the short-hand P.p to denote the set of

id	p	s
1	GQISDSIEEKRGFF	HLLLLLLLLLHEE
2	EEKKGFEEKRAVW	LLEEEEEHHHHHL
3	QDGGSEEKSTKEEK	HHHHLLLEEEELLL

str = EEK , regexp = <l 3 5>
P.p * str = {(1, (8), (3), (3)), (2, (1, 7), (3, 3), (2, 1)), (3, (6, 12), (3, 3), (1, 0)) }
P.s * regexp = {(1, (2, 3, 4, 5, 6, 7, 8), (5, 5, 5, 5, 5, 4, 3), (1, 1, 1, 1, 1, 0, 1)), (3, (5, 12), (3, 3), (2, 2))}

Table 2.1: Example Matches

ID	POSITION	LENGTH	SCORE
1	8	3	3
2	1	3	2
2	7	3	1
3	6	3	1
3	12	3	0

Table 2.2: Unnested Relation

primary sequences, and P.s to denote the set of secondary sequences.

The scores in the example above have been arbitrarily assigned. However, we can also choose to explicitly specify the scoring criteria. If we for instance wished to specify that the matches be scored using the BLOSUM62 matrix, we would say P.p $*_{BLOSUM62}$ str . This is a simple way of expressing the idea used in BLAST for similarity searching. The type of matching operation used has an effect on the semantics of other operators that we will describe in subsequent sections.

Nest (ν) and Unnenst (μ) Operators

$$* : Set < matches > \rightarrow Set < matches >$$

Unnest is a simple operator that flattens out a relation holding matches. For instance, Unnest({(1, (8), (3), (3)), (2, (1, 7), (3, 3), (2, 1)), (3, (6, 12), (3, 3), (1, 0))}), would result in the relation shown in Table 2.2.

The Nest operation is merely the reverse. It collapses each of the tuples (matches) into single, more complex matches. Note that the ID does not serve as a key in this relation.

(ID, Position) can serve as a composite key. These operators make it easier to define certain operations like intersection.

Union Operator ($\cup$)

$$\cup : Set < matches > \times Set < matches > \rightarrow Set < matches >$$

The set union operator generates a set that consists of all the matches of the two sets it operates on. If match with a common protein exist in the two sets, their match-positions are combined, and their length functions are updated. Symbolically, the operation is represented as:

$T = R \cup S$, where R and S are two sets of matchings, and T is their union.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4))}

S = {(2, (5), (4), (3)), (5, (1, 8), (5, 5), (4, 5))}

T = R $\cup$ S = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4, 5), (4, 4, 4), (3, 4)), (5, (1, 8), (5, 5), (4, 5))}

Intersection Operator ($\cap$)

$$\cap : Set < matches > \times Set < matches > \rightarrow Set < matches >$$

The intersection of two sets of matches consists only of matches with proteins common to both sets. Within each match, only match-positions common to both sets are included. Symbolically, the operation is represented as:

$T = R \cap S$, where R and S are two sets of matches, and T is their exact intersection.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4)), (3, (7, 13, 22), (7, 7, 7), (5, 6, 6))}

S = {(2, (1, 5), (4, 6), (3, 5)), (3, (13), (7), (6)), (5, (1, 8), (6, 6), (5, 6))}

T = R ∩ S = {(2, (1), (4), (3)), (3, (13), (7))}

To disambiguate the definition, we observe that T = Nest (Unnest(R) ∩ Unnest(S)).

Difference Operator $(-)$

$$- : Set < matches > \times Set < matches > \rightarrow Set < matches >$$

The difference of two sets of matches consists of matches that are present in the first set and not the second. The operation is clearly not commutative. If matches with a common protein exist in both sets, only match-positions in the first set that do not occur in the second are included in the result set. In such cases, the length function of the match may be updated if its cardinality changes. Symbolically, the operation is represented as:

T = R $-$ S, where R and S are two sets of matches, and T is their difference.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4)), (3, (7, 13, 22), (7, 7, 7), (5, 6, 6))}

S = {(2, (1, 5), (4, 6), (3, 5)), (3, (13), (7), (6)), (5, (1, 8), (6, 6), (5, 6))}

T = R $-$ S = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (4), (4), (4)), (3, (7, 22), (7, 7), (5, 6))}

Match Extension Operators $(|, ||)$

$|| : Set < matches > \times Set < matches > \rightarrow Set < matches >$, and

$| : matchXmatch \rightarrow match)$

The Match Extension operator, operates on two matches, and returns a match that is the list of all matches that can be formed by concatenating a match from Match-1 with a match from Match-2. That is, the result of the operator is the list of matches in its left operand that could be extended in length using the right operand. Symbolically, let

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots, a_k), f, s_1)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots, b_L), g, s_2)$ be another match.

$m_1|m_2$ is defined only when $pid_1 = pid_2$, and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_n), h, s3)$, where $c_i = a_j$, $h(i) = f(i) + g(i)$, and for some p, $a_j + f(j) = b_p$.

Clearly, the operator is not commutative. If R and S are sets, then the operation of match extension can be written as: $T = R||S = \{m|m = m_1|m_2, m_1 \in R, m_2 \in S\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = R || S = {(1, (1, 22), (14, 13), (13, 11))}

The general form of the extension operator concatenates two matches that are at most at a distance k and re-computes the score of the new match. The match extension operator described above is obtained by putting $k = 0$ in the generalized form. One can mathematically describe the operator as follows:

$m1 = (pid_1, (a_1, a_2, a_3, \ldots, a_k), f, s_1)$ be a match, and

$m2 = (pid_2, (b_1, b_2, b_3, \ldots, b_L), g, s_2)$ be another match.

$m1|_k m_2$ is defined only when $pid_1 = pid_2$,

and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_n), h, s_3)$,

where $c_i = a_j$, and for some $p, b_p - (a_j + f(j)) \leq k$, and $h(i) = f(i) + g(i) + k$

If R and S are sets, then the operation of match extension can be written as T = R $||_k$ S = $\{m | m = m_1|_k m_2, m_1 \in$ R, $m_2 \in$S$\}$.

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (7, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = R $||_2$ S = {(1, (1, 22), (16, 13), (12, 11))}

Overlap Operator (Φ, ϕ)

$\Phi : Set \times Set \rightarrow Set$, and

$\phi : match \times match \rightarrow match)$

The overlap operator is in a certain sense a generalization of the exact intersection operator, i.e., a weaker definition of intersection over a set of Matches. It returns a match position in Match-1 (the first match input), if the corresponding string contains (is a superset of) the string that corresponds to a match position in Match-2. The purpose of this operator, in the algebra, is to express queries for proteins containing a fragment that, in its entirety, has a certain primary (or secondary) structure while only a part of it has a certain secondary (or primary) structure. Let:

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots, a_k), f, s)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots, b_l), g, t)$ be another match.

$m_1 \phi m_2$ is defined only when $pid_1 = pid_2$ and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_n), h)$,

where $c_i = a_j$, and for some $b_l, a_j \leq b_l$ and $a_j + f(j) \geq b_l + g(l)$

Symbolically, if R and S are sets, then the result of this operation, T can be written as:

$T = R\Phi S = \{m | m_1 \in R, m_2 \in S, m = m_1 \phi m_2\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = R Φ S = {(1, (8), (10), (8))}

Non-overlap Operators (Ψ, ψ)

$\Psi : Set < matches > \times Set < matches > \rightarrow Set < matches >$, and

$\psi : match \times match \rightarrow match)$

The non overlap operator is in some sense a generalization of the difference operator. Over matches, the operation produces a match from its left operand if a match element from the right operand does not overlap with it completely. Over sets of matches, the operator basically does the non overlap checking with every pair of matches. The purpose of this operator, in the algebra, is to express queries for proteins containing a fragment that, in its entirety, has a certain primary (or secondary) structure while no part of it has a certain secondary (or primary) structure. Let:

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots a_k), f)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots, b_l), g)$ be another match.

$m_1 \psi m2$ is defined only when $pid_1 = pid_2$ and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_n), h)$, where $c_i = a_j$, and for no $b_l, a_j \leq b_l$ and $a_j + f(j) \geq b_l + g(l)$

If R and S are sets, then the result of this operation, T can be written as: $T = R\Psi S =$ $\{m | m_1 \in R, m_2 \in S, m = m_1 \psi m_2\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = RΨS = {(1, (1, 22), (4, 6), (4, 5)), (2, (3, 7), (3, 4), (3, 3))}

2.4 Expressive Power of PiQA

2.4.1 Sample Queries

In this section, we demonstrate the expressive power of PiQA using several different examples.

1. Find all proteins that contain the primary structure sequence "QISDSIE" with the secondary structure of "DSI" being <H 3 3> or <L 3 3>.

   (P.p * "QIS") || ((P.p * "DSI") Φ ((P.s * <H 3 3>) ∪ (P.s * <L 3 3>))) || (P.p * "E")

2. Find all proteins that contain the secondary structure <E 1 5><L 2 2><E 3 9> and a primary structure sequence "SSDGTQ" nowhere within it.

   (P.s * <E 1 5><L 2 2><E 3 9>) Ψ (P.p * "SSDGTQ")

3. Find all proteins that contain the primary structure sequence "SPPNKD" with the condition that the secondary structure for "PP" is not <E 2 2>.

   (P.p * "S") || ((P.p * "PP") - (P.s * <E 2 2>)) || (P.p * "NKD")

4. Find all proteins that have the secondary structure <H 3 6> or <E 4 5> and the primary structure "NKN" contained in it.

   ((P.s * <H 3 6>) ∪ (P.s * <E 4 5>)) Φ (P.p * "NKN")

5. Match "AAANBPPPPSDF" with the database, but ignore mismatch in the segment NBPPP if it is on a loop.

(P.p * “AAA”) || ((P.p * “NBPPP”) ∪ (P.s * <L 5 5>)) || (P.p * “PSDF”)

6. 6. Match a protein with secondary structure <L 20 40><E 10 30> that has the fragment “AAPQS” in the loop segment.

   ((P.s * <L 20 40>) Φ (P.p * “AAPQS”)) || (P.s * <E 10 30>)

2.4.2 Expressing BLAST

As mentioned in Section 2.2, BLAST is a family of similarity searching tools. One of the tools called blastp is used for similarity searching amongst protein datasets. The BLAST idea was trivially expressed in Section 2.3. We can express the BLAST algorithm at a finer detail. We will express a blastp query for “QAANVP” in PiQA as a demonstration.

To express this query, we need the following notation: Δ_k is used to denote the set of all possible protein strings of length k.

If we are considering proteins from only the basic 20 amino acids, then the size of Δ_k would be 20^k. The first step of BLAST, in which we prune out all the *k*-mers below a certain threshold, can be expressed as follows:

(1) A = {QAA, AAN, ANV, NVP} $*_{BLOSUM62,Threshold=T}\Delta_3$

The second step of BLAST, in which the database is searched to find hits that match with any of the *k*-mers from the previous step is:

(2) B = A $*_{Exact}$ P.p

The third step of BLAST involves extending the hits to form HSPs. The first version of BLAST extends the hits residue by residue on both sides. In the second version, a two-hit method is used which extends hits only when two of the hits are within a certain distance. There are several variations on the heuristic for this step. We express this step as:

(3) C = R $||_{Maxdist}$ R $||_{Maxdist}$ R $||_{Maxdist}$ R ...

We could incorporate a transitive closure for this operator in the algebra so that all possible ways of extending the hits are captured in the algebra. BLAST stops extending the hits when the score of the extended hit drops below a certain value of the maximum it had reached since the start of the process of extending. A simple filtering operation can be defined to select out only those matches with a minimum score from C. When writing this programmatically, a while loop structure can be used to stop the hit-extension process precisely when it is desired. This is much like the use of *while* in SQL even though it is not part of the relational algebra.

2.5 Query Evaluation

A typical query of the protein dataset consists of one or more search predicates connected by operators defined in the algebra. The first step in the evaluation of such a query is to generate all possible query plans that can be used to evaluate the query correctly. These plans differ essentially in the order in which the algebraic operations are performed. Costs are computed for each query plan and the cheapest plan is selected for evaluation.

2.5.1 Cost Model

The cost of a query plan is essentially the sum-total of costs of all the operations performed in it. We observe the asymptotic complexities of the cost functions of the various algebraic operators to be as follows.

1. Cost of a Match Operation (*) The cost of a match operation of the form ($P * ps$) depends on the exact algorithm used to find the matches. We expect that the parameters involved would be:

   (a) $|P|$, the size of the protein database P in bytes,

   (b) S_{ps}, the selectivity of the search sequence ps, and

(c) Other parameters used by the specific algorithm used to perform the matching.

2. Cost of Set (Union, Intersection or Difference) Operations ($\cup$, $\cap$, $-$)

   The cost of an exact union operation $(A \cup B)$, an exact intersection operation $(A \cap B)$ or and exact difference operation $(A - B)$, where A and B are two sets of matches, is a function of the sizes of the two sets, $|A|$ and $|B|$. If the two sets are sorted, the cost of the operation is of the order $O(max(|A|, |B|))$.

3. Cost of an Overlap or Non Overlap Operation Operations (Φ,Ψ)

   The cost of an approximate intersection operation $(A \Phi B)$ or an approximate difference operation $(A \Psi B)$, where A and B are two sets of matchings, is a function of the lengths of the two sets, $|A|$ and $|B|$. The cost of the operation is of the order $O(|A| \times |B|)$.

4. Cost of a Match Extension Operation ($||$)

   The cost of a match extension operation $(A||B)$ where A and B are two sets of matchings, is also of the order $O(max(|A|, |B|))$.

2.5.2 Generation of Query Plans

A query of the protein dataset can have multiple plans that direct its evaluation. The one selected is that which minimizes the overall cost incurred. Each query plan generated is a tree in which the resulting set of proteins of one search predicate becomes the base dataset for all the other search predicates in the query. We illustrate the generation of query plans with an example:

EQ: *Match the primary structure sequence AAANBPPPPSDF with the database, but ignore a mismatch in the segment NBPPP if it is on a loop.*

In our algebra, this query is expressed as:

(P.p * AAA) || ((P.p * NBPPP) ∪ (P.s * <L 5 5>)) || (P.p * PSDF)

Since the match extension operator (||) is associative and distributive over the union operator (∪), we can evaluate the predicates in the following different orders:

1. (P.p * AAA) || ((P.p * NBPPP) ∪ (P.s * <L 5 5>)) || (P.p * PSDF)
2. ((P.p * AAA) || (P.p * NBPPP) || (P.p * PSDF)) ∪ ((P.p * AAA) || (P.s * <L 5 5>) || (P.p * PSDF))
3. ((P.p * AAA) || (P.p * NBPPP)) ∪ ((P.p * AAA) || (P.s * <L 5 5>)) || (P.p * PSDF)
4. ((P.p * AAA) || ((P.p * NBPPP) || (P.p * PSDF)) ∪ ((P.s * <L 5 5>) || (P.p * PSDF))

Observe that in executing each of the plans listed above, we would need to perform three match operations. Also notice that all the strings that are in the result of the expression are likely to be in the result of each of the match operations. We may be able to optimize this query by picking one of the three match operations, and using the list of all proteins that it outputs to constitute the set of strings over which the other matches are done, instead of performing the matches over the entire dataset P. This set of strings is likely to be much smaller than the full dataset P, which may lead to a more efficient query plan. However, the cost we incur is that we may miss out good matches that might have been found by the other match operators, but would not have matched with the first operator. We can quantify this tradeoff and let the optimizer decide how many of the match operations should be done from the base dataset and how many from the output set of strings of other match operators.

Consider that we have three match sub-queries as in the above example. We define the selectivity of a sub-query as the ratio of the number of strings in the output to the number of strings in the base dataset. We define the importance of a sub-query as a measure of its importance relative to the entire query. If there is only one match operation, then its

importance is unity. The importance is a measure of how much of an impact a match on the sub-query fragment will contribute to the overall match score. A simple metric for this is the ratio of the length of the sub-query to the length of the full query. To illustrate this point, consider the following example: (P.p * PNB) || (P.p * AAATTTAAA)

Let p_1 and p_2 denote the two sub-queries "PNB" and "AAATTTAAA". For the purpose of this example, let the entire protein database be the following set of proteins, each row the form (id, primary structure):

(1, *AAATTTAAA*AAU*PNB*PSTTT)

(2, PSSSQRRTTSTRRAAAUWVV)

(3, UIPPSTTTGGG*AAATTTAAA*R)

(4, QQQPPLSSTTRRWRNNNBBB)

(5, *AAATTTAAA*VVVWUIPLAAR)

(6, WQQQRRWSSTWWWCCFFFA)

(7, PPPPPNNNNNNPP*PNB*STRRQ)

(8, TTT*AAATTTAAA*RASTTTWW)

(9, VVVWWWAAABBBSSSQQQT)

(10, PPPQQSSSRRR*AAATTTAAA*)

Here we see that the selectivity of sub-query p_1 is 0.2 while that of p_2 is 0.5. We want to use more selective (one with least selectivity) sub-queries so that the size of the new dataset is smaller. If we just tried to use p_1, and use its outputs then we would miss out on matches like strings 3, 5, 8, and 10. We also need to consider the importance of the sub-query. Clearly, p_1 is less important than p_2. The optimizer can use some preferences from the user about how much optimization at what cost to quality should be done by specifying selectivity and importance levels for which the output of one match operation

may be used for the input of another.

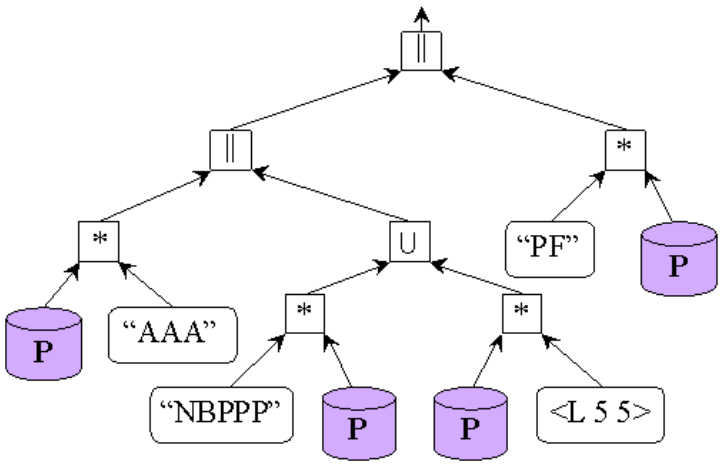

Figure 2.2: Query Plan for EQ

We present one possible query plan that could be generated for query **EQ** in Figure 2.2 We introduce a filtering operation π, which extracts the IDs from a set of matches obtained from the result of a match operation. This operator is used to construct an alternative and likely more optimal query plan for this same query is shown in Figure 2.3. The fragment "AAA" is first matched, and the results of that query are further probed with the remaining match operators leading to a potentially large savings in computational effort.

2.6 Conclusions

In this chapter we have presented PiQA, an algebra for expressing queries on both the primary and secondary structures of proteins. The algebra provides a rich set of operators that permits approximate matching, combination of two or more matches, and various

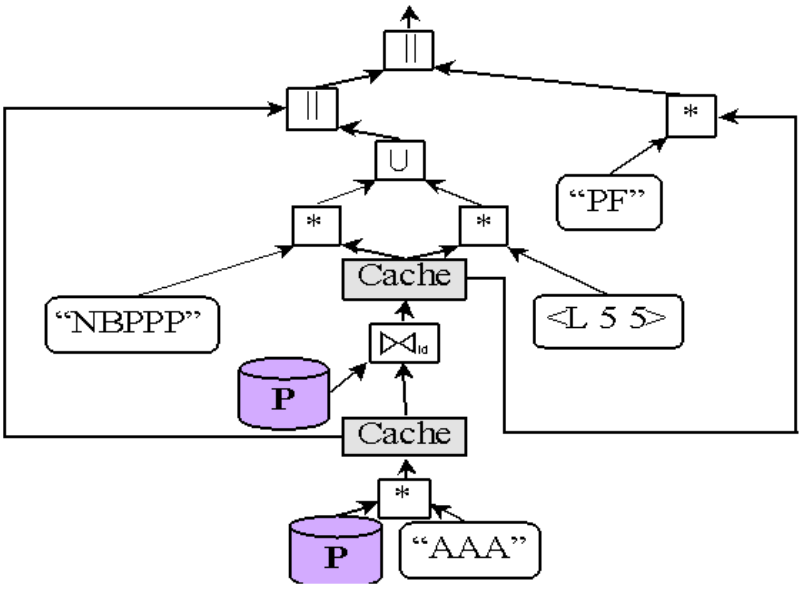

Figure 2.3: More Efficient Query Plan for EQ

set operations on the matches. The algebra provides a unified approach to querying on both primary and secondary structures of proteins, and also be used to optimize complex queries.

PiQA is the first step in developing a declarative querying interface on all protein structures. In the following chapters, we describe actual system that we built using the basic algebraic framework provided by PiQA.

We note that although this chapter concentrates on the protein structures, the algebra can also be applied to querying on nucleotide sequence data sets, which are similar in nature (from the querying perspective) to the protein primary sequences. In fact we demonstrate several examples using DNA sequences in the following chapters.

CHAPTER III

Suffix Tree Construction

3.1 Introduction

Querying large string datasets is becoming increasingly important in a number of life-science and text applications. Life science researchers are often interested in explorative querying of large biological sequence databases, such as genomes and large sets of protein sequences. Many of these biological datasets are growing at exponential rates — for example, the sizes of the sequence datasets in GenBank have been doubling every sixteen months [146]. Consequently, methods for *efficiently* querying large string datasets are critical to the success of these emerging database applications.

A suffix tree is a versatile data structure that can help execute such queries efficiently. In fact, suffix trees are useful for evaluating a wide variety of queries on string databases [64]. For instance, the exact substring matching problem can be solved in time proportional to the length of the query, once the suffix tree is built on the database string. Suffix trees can also be used to solve approximate string matching problems efficiently. Some bioinformatics applications such as MUMmer [40,41,90], REPuter [92], and OASIS [105] exploit suffix trees to efficiently evaluate queries on biological sequence datasets. However, suffix trees are not widely used because of their high cost of construction. As we show in this chapter, building a suffix tree on moderately sized datasets, such as a single chromosome

of the human genome, takes over 1.5 hours with the best known existing disk-based construction technique [73]. In contrast, the techniques that we develop in this chapter reduce the construction time by a factor of five on inputs of the same size.

Even though suffix trees are currently not in widespread use, there is a rich history of algorithms for constructing suffix trees. A large focus of previous research has been on linear-time suffix tree construction algorithms [103, 156, 166]. These algorithms are well suited for small input strings where the tree can be constructed entirely in main memory. The growing size of input datasets, however, requires that we construct suffix trees efficiently on disk. The algorithms proposed in [103, 156, 166] cannot be used for disk-based construction as they have poor locality of reference. This poor locality causes a large amount of random disk I/O once the data structures no longer fit in main memory. If we naively use these main-memory algorithms for on-disk suffix tree construction, the process may take well over a day for a single human chromosome.

The large and rapidly growing size of many string datasets underscores the need for fast *disk-based* suffix tree construction algorithms. Theoretical methods for optimal external memory suffix tree construction have also been developed [54], however, the practical behavior of these algorithms has not been explored. A number of recent research investigations have also examined practical suffix tree construction techniques for large data sets [16, 73]. However, these approaches do not scale well for large datasets (such as an entire eukaryotic genome).

In this chapter, we present new approaches for *efficiently* constructing large suffix trees on disk. We use a philosophy similar to the one in [73]. We forgo the use of suffix links in return for a much better memory reference pattern, which translates to better scalability and performance for constructing large suffix trees.

The main contributions in this chapter are as follows:

1. We introduce the "Top Down Disk-based" (TDD) approach which can be used to efficiently build suffix trees for a wide range of sizes and input types. This technique includes a suffix tree construction algorithm called PWOTD, and a sophisticated buffer management strategy.

2. We compare the performance of TDD with Ukkonen [156], McCreight [103], and a suffix array based technique: Deep-Shallow [101] for the in-memory case, where all the data structures needed for building the suffix trees are memory resident (i.e. the datasets are "small"). Interestingly, we show that even though Ukkonen and McCreight have a better worst-case *theoretical* cost on a random access machine, TDD and Deep-Shallow perform better on modern cached processors because they incur fewer cache misses.

3. We systematically explore the space of data sizes and types, and highlight the advantages and disadvantages of TDD with respect to other construction algorithms.

4. We experimentally demonstrate that TDD scales gracefully with increasing input size. With extensive experimental evaluation, we show that TDD outperforms existing disk-based construction methods. Using the TDD process, we are able to construct a suffix tree on the *entire human genome in thirty hours* on a single processor machine! To the best of our knowledge, suffix tree construction on an input string of this size (approximately three billion symbols) has yet to be reported in literature.

5. We describe a new algorithm called ST-Merge that is based on a partition and merge strategy. We experimentally show that ST-Merge algorithm is more efficient than TDD when the input string size is significantly larger than the available memory. However, for most current biological sequence datasets on modern machines with large memory configuration, TDD is the algorithm of choice.

The remainder of this chapter is organized as follows: Section 3.2 discusses related work. The TDD technique is described in Section 3.3, and we analyze the behavior of this algorithm in Section 3.4. The ST-Merge algorithm is presented in Section 3.5. Section 3.6 describes the experimental results, and Section 3.7 presents our conclusions.

3.2 Related Work

Linear time algorithms for constructing suffix trees have been described by Weiner [166], McCreight [103], and Ukkonen [156]. (For a discussion on the relationship among these algorithms, see [59].) Ukkonen's is a popular algorithm because it is easier to implement than the other algorithms. It is an $O(n)$, in-memory construction algorithm based on the clever observation that constructing the suffix tree can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of *suffix links*, which provide a mechanism for quickly traversing across subtrees, the suffix tree can be expanded by simply adding the $i + 1^{st}$ character to the leaves of the suffix tree built on the previous i characters. The algorithm thus relies on suffix links to traverse through all of the subtrees in the main tree, expanding the outer edges for each input character. McCreight's algorithm is a space-economical linear time suffix tree construction algorithm. This algorithm starts from an empty tree and inserts suffixes into the partial tree from the longest to the shortest suffix. Like Ukknonen's algorithm, McCreight's algorithm also utilizes suffix links to traverse from one part of the tree to another. Both are linear time algorithms, but they have poor locality of reference. This leads to poor performance on cached architectures and on disk.

Variants of suffix trees have been considered for disk-based construction [71]. Recently, Bedathur and Haritsa developed a buffering strategy, called TOP-Q, which improves the performance of Ukkonen's algorithm (which uses suffix links) when constructing on-disk

suffix trees [16]. A different approach was suggested by Hunt et al. [73] where the authors drop the use of suffix links and use an $O(n^2)$ algorithm with a better locality of memory reference. In one pass over the string, they index all suffixes with the same prefix by inserting them into an on-disk subtree managed by PJama [12], a Java-based object store. Construction of each independent subtree requires a full pass over the string. The main drawback of Hunt's algorithm is that the tree traversal incurs a large number of random accesses during the construction process. A partition and clustering based approach is described by Schürmann and Stoye in [131] which is an improvement over Hunt et al. This approach uses clustering to better organize disk accesses. A partitioning-based approach was suggested by Clifford and Sergot in [33] to build distributed and paged suffix trees. However, this is an in-memory technique. Cheung et al. [31] have recently proposed an algorithm called *DynaCluster*. This algorithm employs a dynamic clustering technique to reduce the random accesses that are incurred during the tree traversal. Every cluster contains tree nodes that are frequently referenced by each other. In this chapter, we compare our suffix tree construction methods with TOP-Q [16], Hunt's [73] method and DynaCluster [31], and show that in practice our methods for constructing suffix trees are more efficient.

A top-down suffix tree construction approach has been suggested in [8]. In [60], Giegerich, Kurtz, and Stoye explore the benefits of using a lazy implementation of suffix trees. In this approach, the authors argue that one can avoid paying the full construction cost by constructing the subtree only when it is accessed for the first time. This approach is useful either when a small number of queries are posed or only short queries are posed against a string dataset. When executing a large number of (longer) queries, most of the tree must be materialized, and in this case, this approach will perform poorly.

Previous research has also produced theoretical results on understanding the average

sizes of suffix trees [18, 142], and theoretical complexity of using sorting to build suffix trees. In [53], Farach describes a linear time algorithm by constructing odd and even suffix trees, and merging them. In [54], the authors show that this algorithm has the same I/O complexity as sorting on the DAM model described by Vitter and Shriver [160]. However, they do not differentiate between random and sequential I/O. In contrast, our approach makes careful choices in order to reduce random I/O, and incurs mostly sequential I/O.

Suffix arrays are closely related to suffix trees, and can be used as an alternative to suffix trees for many string matching tasks [1,30,34,108]. A suffix tree can also be constructed by first building a suffix array. With the help of an additional LCP (Longest Common Prefix) array, a suffix array can be converted into a suffix tree in $O(n)$ time. Theoretical linear time suffix array construction algorithms have been proposed in [84,87,88]. There has also been considerable interest in practical suffix array construction algorithms. The Deep-Shallow algorithm proposed in [101] is a space efficient internal memory suffix array construction algorithm. Although its worst case cost is $\Theta(n^2 \log n)$, it is arguably the fastest in-memory method in practice. In [84, 85, 100], algorithms for constructing LCP arrays in linear time are proposed.

The long interest of the algorithmic community in optimal external memory suffix array construction algorithms has led to the external DC3 algorithm recently proposed by Dementiev et al. [42]. This external construction method is based on the Skew algorithm [84]. The Skew algorithm is a theoretically optimal suffix array construction algorithm, and uses a merge-based approach. This method recursively reduces the suffix array construction using a two thirds to one thirds split of the suffix array. Each recursive call first sorts the larger array, and the smaller array is sorted using the ordering information in the larger array. The arrays are merged to produce the final array. The external DC3 algorithm extends the in-memory Skew algorithm with the help of the STXXL library [140]. The STXXL

library is a C++ template library that enables containers and algorithms to process large amounts of data that don't fit in main memory. It also improves performance by supporting multiple disks and overlapping I/O with CPU computation (see [140] for details). The external DC3 algorithm [42] is theoretically optimal and superior to the previous external suffix array construction methods in practice. We draw some comparisons between our methods and the external DC3 algorithm in Section 3.6.5, and show that in practice TDD is faster than the external DC3 algorithm.

TDD uses a simple partitioning strategy. However, a more sophisticated partitioning method was recently proposed by Carvalho et al. [25], which can complement our existing partitioning method.

3.3 The TDD Technique

Most suffix tree construction algorithms do not scale due to the prohibitive disk I/O requirements. The high per-character space overhead of a suffix tree quickly causes the data structures to outgrow main memory, and the poor locality of reference makes efficient buffer management difficult.

We now present a new disk-based construction technique called the "Top-Down Disk-based" technique, hereafter referred to simply as TDD. TDD scales much more gracefully than existing techniques by reducing the main-memory requirements through strategic buffering of the largest data structures. The TDD technique consists of a suffix tree construction algorithm, called PWOTD, and the related buffer management strategy described in the following sections.

3.3.1 PWOTD Algorithm

The first component of the TDD technique is our suffix tree construction algorithm, called PWOTD (Partition and Write Only Top Down). This algorithm is based on the

wotdeager algorithm suggested by Giegerich *et al*. [60]. We improve on this algorithm by using a partitioning phase which allows one to immediately build larger, independent subtrees in memory. (A similar partitioning strategy was proposed in [131].) Before we explain the details of our algorithm, we briefly discuss the representation of the suffix tree.

The suffix tree is represented by a linear array, just as in *wotdeager*. This is a compact representation using 8.5 bytes per indexed symbol in the average case with 4 byte integers. Figure 3.1 illustrates a suffix tree on the string ATTAGTACA$ and the tree's corresponding array representation in memory. Shaded entries in the array represent leaf nodes, with all other entries representing non-leaf nodes. An *R* in the lower right-hand corner of an entry denotes a rightmost child. Note that leaf nodes are represented using a single integer, while non-leaf nodes use two integers. (The two entries of a non-leaf node are separated by a dashed line in the figure.) The first entry in a non-leaf node is an index into the input string; the character at that index is the starting character of the incoming edge's label. The length of the label can be deduced by examining the children of the current node. The second entry in a non-leaf node points to the first child. For example, in Figure 3.1, the non-leaf node represented by the entries indexed by 0 and 1 in the tree array has four leaf children located at entries 12, 13, 14 and 15, respectively. The parent's suffix starts at index 0 in the string, whereas the children's suffixes begins with the indexes 1, 7, 4 and 9, respectively. Therefore, we know the length of the parent's edge label is $min\{1, 7, 4, 9\} - 0 = 1$. Note that the leaf nodes do not have a second entry. The leaf node requires only the starting index of the label; the end of the label is the string's terminating character. See [60] for a more detailed explanation.

The PWOTD algorithm consists of two phases. In the first phase, we partition the suffixes of the input string into $|A|^{prefixlen}$ partitions, where $|A|$ is the alphabet size of the string and *prefixlen* is the depth of the partitioning. The partitioning step is executed

as follows. The input string is scanned from left to right. At each index position i, the *prefixlen* subsequent characters are used to determine one of the $|A|^{prefixlen}$ partitions. This index i is then written to the calculated partition's buffer. At the end of the scan, each partition will contain the suffix pointers for suffixes that all have the same prefix of size *prefixlen*. Note that the number of partitions ($|A|^{prefixlen}$) is much smaller than the length of the string.

To further illustrate the partition step, consider the following example. Partitioning the string ATTAGTACA$ using a *prefixlen* of 1 would create four partitions of suffixes, one for each symbol in the alphabet. (We ignore the final partition consisting of just the string terminator symbol $.) The suffix partition for the character A would be {0,3,6,8}, representing the suffixes {ATTAGTACA$, AGTACA$, ACA$, A$}. The suffix partition for the character T would be {1,2,5} representing the suffixes {TTAGTACA$, TAGTACA$, TACA$}. In phase two, we use the *wotdeager* algorithm to build the suffix tree on each partition using a top down construction.

The pseudo-code for the PWOTD algorithm is shown in Figure 3.2. While the partitioning in phase one of PWOTD is simple enough, the algorithm for *wotdeager* in phase two warrants further discussion. We now illustrate the *wotdeager* algorithm using an example.

Example Illustrating the wotdeager Algorithm

The PWOTD algorithm requires four data structures for constructing suffix trees: an input string array, a suffix array, a temporary array, and the suffix tree. For the discussion that follows, we name each of these structures *String*, *Suffixes*, *Temp*, and *Tree*, respectively.

The Suffixes array is first populated with suffixes from a partition after discarding the first *prefixlen* characters. Using the same example string as before, ATTAGTACA$ with *prefixlen*=1, consider the construction of the Suffixes array for the T-partition. The suffixes

in this partition are at positions 1, 2, and 5. Since all these suffixes share the same prefix, T, we add one to each offset to produce the new Suffix array {2,3,6}. The next step involves sorting this array of suffixes based on the first character. The first characters of each suffix are {T, A, A}. The sorting is done in linear time using an algorithm called *count-sort* (for a constant alphabet size). In a single pass, for each character in the alphabet, we count the number of occurrences of that character as the first character of each suffix, and copy the suffix pointers into the Temp array. We see that the count for A is 2 and the count for T is 1; the counts for G, C, and $ are 0. We can use these counts to determine the character group boundaries: group A will start at position 0 with two entries, and group T will start at position 2 with one entry. We make a single pass through the Temp array and produce the Suffixes array sorted on the first character. The Suffixes array is now {3, 6, 2}. The A-group has two members and is therefore a branching node. These two suffixes completely determine the subtree below this node. Space is reserved in the Tree to write this non-leaf node once it is expanded, then the node is pushed onto the stack. Since the T-group has only one member, it is a leaf node and will be immediately written to the Tree. Since no other children need to be processed, no additional entries are added to the stack, and this node will be popped off first.

Once the node is popped off the stack, we find the longest common prefix (LCP) of all the nodes in the group {3, 6}. We examine position 4 (G) and position 7 (C) to determine that the LCP is 1. Each suffix pointer is incremented by the LCP, and the result is processed as before. The computation proceeds until all nodes have been expanded and the stack is empty. Figure 3.1 shows the complete suffix tree and its array representation.

Discussion of the PWOTD Algorithm

Observe that phase 2 of PWOTD operates on subsets of the suffixes of the string. In *wotdeager*, for a string of n symbols, the size of the Suffixes array and the Temp array needed to be $4 \times n$ bytes (assuming 4 byte integers are used as pointers). By partitioning in phase 1, the amount of memory needed by the suffix arrays in each run is just $(4 \times n)/(|A|^{prefixlen})$ on average. (Some partitions might be smaller and some larger than this figure due to skew in real world data. Sophisticated partitioning techniques can be used to balance the partition sizes [25].) The important point is that partitioning decreases the main-memory requirements for suffix tree construction, allowing independent subtrees to be built entirely in main memory. Suppose we are partitioning a 100 million symbol string over an alphabet of size 4. Using a $prefixlen = 2$ will decrease the space requirement of the Suffixes and Temp arrays from 400 MB to approximately 25 MB each, and the Tree array from 1200 MB to 75 MB. Unfortunately, this savings is not entirely free. The cost of the partitioning phase is $O(n \times prefixlen)$, which increases linearly with *prefixlen*. For small input strings where we have sufficient main memory for all the structures, we can skip the partitioning phase entirely. It is not necessary to continue partitioning once the Suffixes and Temp arrays fit into memory. For even very large datasets, such as the human genome, partitioning with *prefixlen* more than 7 is not beneficial.

3.3.2 Buffer Management

Since suffix trees are an order of magnitude larger in size than the input data strings, suffix tree construction algorithms require large amounts of memory, and may exceed the amount of main memory that is available. For such large datasets, efficient disk-based construction methods are needed that can scale well for large input sizes. One strength of TDD is that its data structures transition gracefully to disk as necessary, and individ-

ual buffer management polices for each structure are used. As a result, TDD can scale gracefully to handle large input sizes.

Recall that the PWOTD algorithm requires four data structures for constructing suffix trees: *String*, *Suffixes*, *Temp*, and *Tree*. Figure 3.3 shows each of these structures as separate, in-memory buffer caches. By appropriately allocating memory and by using the right buffer replacement policy for each structure, the TDD approach is able to build suffix trees on extremely large inputs. The buffer management policies are summarized in Figure 3.3 and are discussed in detail below.

The largest data structure is the Tree buffer. This array stores the suffix tree during its intermediate stages as well as the final computed result. The Tree data structure is typically 8-12 times the size of the input string. The reference pattern to Tree consists mainly of sequential writes when the children of a node are being recorded. Occasionally, pages are revisited when an unexpanded node is popped off the stack. This access pattern displays very good temporal and spatial locality. Clearly, the majority of this structure can be placed on disk and managed efficiently with a simple LRU (*L*east *R*ecently *U*sed) replacement policy.

The next largest data structures are the Suffixes and the Temp arrays. The Suffixes array is accessed as follows: first a sequential scan is used to copy the values into the Temp array. The count phase of the count sort is piggybacked on this sequential scan. The sort operation following the scan causes writes back into the Suffixes array. However, there is some locality in the pattern of writes in the Suffixes array, since the writes start at each character-group boundary and proceed sequentially to the right. Based on the (limited) locality of reference, one expects LRU to perform reasonably well. The Temp array is referenced in two sequential scans: the first to copy all of the suffixes in the Suffixes array, and the second to copy all of them back into the Suffixes array in sorted order. For this

reference pattern, replacing the most recently used page (MRU) works best.

The String array has the smallest main-memory requirement of all the data structures, but the worst locality of access. The String array is referenced when performing the count-sort and to find the longest common prefix in each sorted group. During the count-sort all of the portions of the string referenced by the suffix pointers are accessed. Though these positions could be anywhere in the string, they are always accessed in left to right order. In the function to find the longest common prefix of a group, a similar pattern of reference is observed. In the case of this find-LCP function, each iteration will access the characters in the string, one symbol to the right of those previously referenced. In the case of the count-sort operation, the next set of suffixes to be sorted will be a subset of the current set. This is a fairly complex reference pattern, and there is some locality of reference, so we expect LRU and RANDOM to do well. Based on evidence in Section 3.6.4, we see that both are reasonable choices.

3.3.3 Buffer Size Determination

To obtain the maximum benefit from buffer management policy, it is important to divide the available memory amongst the data structures appropriately. A careful apportioning of the available memory between these data structures can affect the overall execution time dramatically. In the rest of this section, we describe a technique to divide the available memory among the buffers.

If we know the access pattern for each of the data structures, we can devise an algorithm to partition the memory to minimize the overall number of buffer cache misses. Note that we need only an access pattern on a string representative of each class, such as DNA sequences, protein sequences, etc. In fact, we have found experimentally that these access patterns are similar across a wide-range of datasets (we discuss these results in detail in Section 3.6.4.) An illustrative graph of the buffer cache miss pattern for each data structure

is shown in Figure 3.4. In this figure, the X-axis represents the number of pages allocated to the buffer as a percentage of the total size of the data structure. The Y-axis shows the number of cache misses. This figure is representative of biological sequences, and it is based on data derived from actual experiments in Section 3.6.4.

As we will see at the end of section 3.3.3, our buffer allocation strategy needs to estimate only the relative magnitudes of the slopes of each curve and the position of the "knee" towards the start of the curve. The full curve as shown in Figure 3.4 is not needed for the algorithm. However, it is useful to facilitate the following discussion.

TDD Heuristic for Allocating Buffers

We know from Figure 3.4 that the cache miss behavior for each buffer is approximately linear once the memory is allocated beyond a minimum point. Once we identify these points, we can allocate the minimum buffer size necessary for each structure. The remaining memory is then allocated in order of decreasing slopes of the buffer miss curves.

We know from arguments in Section 3.3.2 that references to the String have poor locality. One can infer that the String data structure is likely to require the most buffer space. We also know that the references to the Tree array have very good locality, so the buffer space it needs is likely to be a very small fraction of its full size. Between Suffixes and Temp, we know that the Temp array has more locality than the Suffixes array, and will therefore require less memory. Both Suffixes and Temp require a smaller fraction of their pages to be resident in the buffer cache when compared to the String. We exploit this behavior to design a heuristic for memory allotment.

We suggest setting the minimum number of pages allocated to the Temp and Suffixes arrays to $|A|$. During the sort phase, we know that the Suffixes array will be accessed at $|A|$ different positions which correspond to the character group boundaries. The incremental

benefit of adding a page will be very high until $|A|$ pages, and then one can expect to see a change in the slope at this point. By allocating at least $|A|$ pages, we avoid the penalty of operating in the initial high miss-rate region. The TDD heuristic chooses to allocate a minimum of $|A|$ pages to Suffixes and Temp first.

We suggest allocating two pages to the Tree array. Two pages allow a parent node, possibly written to a previous page and then pushed onto the stack for later processing, to be accessed without replacing the current active page. This saves a large amount of I/O over choosing a buffer size of only one page.

The remaining pages are allocated to the String array up to its maximum required amount. If any pages are left over, they are allocated to Suffixes up to its maximum requirement. The remaining pages (if any) are allocated to Temp, and finally to Tree.

The reasoning behind this heuristic is borne out by the graphs in Figure 3.4. The String, which has the least locality of reference, has the highest slope and the largest magnitude. Suffixes and Temp have a lower magnitude and a more gradual slope, indicating that the improvement with each additional page allocated is smaller. Finally, the Tree, which has excellent locality of reference, is nearly zero. All curves have a knee which we estimate by choosing minimum allocations.

An Example Allocation

The following example demonstrates how to allocate the main memory to the buffer caches. Assume that your system has 100 buffer pages available for use and that you are building a suffix tree on a small string that requires 6 pages. Further assume that the alphabet size is 4 and that 4 byte integers are used. Assuming that no partitioning is done, the Suffixes array will need 24 pages (one integer for each character in the String), the Temp array will need 24 pages, and the Tree will need at most 72 pages. First we allocate

4 pages each to Suffixes and Temp. We allocate 2 pages to Tree. We are now left with 90 pages. Of these, we allocate 6 pages to the String, thereby fitting it entirely in memory. From the remaining 84 pages, Suffixes and Temp are allocated 20 and fit into memory, and the final 44 pages are all given to Tree. This allocation is shown pictorially in the first row of Figure 3.5.

Similarly, the second row in Figure 3.5 is an allocation for a medium sized input of 50 pages. The heuristic allocates 2 pages to the Tree, 4 to the Temp array, 44 to Suffixes, and 50 to the String. The third allocation corresponds to a large string of 120 pages. Here, Suffixes, Temp, and Tree are allocated their minimums of 4, 4, and 2 respectively, and the rest of the memory (90 pages) is given to String. Note that the entire string does not fit in memory now, and portions will be swapped into memory from disk when they are needed.

Observe from Figure 3.5 that when the input is small and all the structures fit into memory, most of the space is occupied by the largest data structure: the Tree. As the input size increases, the Tree is pushed out to disk. For very large strings that do not fit into memory, everything but the String is pushed out to disk, and the String is given nearly all of the memory. By first pushing the structures with better locality of reference onto disk, TDD is able to scale gracefully to very large input sizes.

Note that our heuristic does not need the actual utility curves to calculate the allotments. It estimates the "knee" of each curve using the algorithm, and assumes that the curve is linear for the rest of the region.

3.4 Analysis

In this section, we analyze the advantages and the disadvantages of using the TDD technique for various types and sizes of string data. We also describe how the design choices we have made in TDD overcome the performance bottlenecks present in other

proposed techniques.

3.4.1 I/O Benefits

Unlike the approach of [16] where the authors use the in-memory $O(n)$ algorithm (Ukkonen) as the basis for their disk-based algorithm, we use the theoretically less efficient $O(n^2)$ *wotdeager* algorithm [60]. A major difference between the two algorithms is that Ukkonen's algorithm sequentially accesses the string data and then updates the suffix tree through random traversals, while our TDD approach accesses the input string randomly and then writes the tree sequentially. For disk-based construction algorithms, random access is the performance bottleneck as on each access an entire page will potentially have to be read from disk; therefore, efficient caching of the randomly accessed disk pages is critical.

On first appearance, it may seem that we are simply trading some random disk I/O for other random disk I/O, but the input string is the smallest structure in the construction algorithm, while the suffix tree is the largest structure. TDD can place the suffix tree in very small buffer cache as the writes are almost entirely sequential, which leaves the remaining memory free to buffer the randomly accessed, but much smaller, input string. Therefore, our algorithm requires a much smaller buffer cache to contain the randomly accessed data. Conversely, for the same amount of buffer cache, we can cache much more of the randomly accessed pages, allowing us to construct suffix trees on much larger input strings.

3.4.2 Main-Memory Analysis

When we build suffix trees on *small* strings (i.e. when the string and all the data structures fit in memory), no disk I/O is incurred. For the case of in-memory construction, one would expect that a linear time algorithm such as Ukkonen or McCreight would per-

form better than the TDD approach, which has a worst case cost of $O(n^2)$. However, one must consider more than just the theoretical cost to understand the execution time of the algorithms.

Traditionally, in designing disk-based algorithms, all accesses to main memory are considered equally good, as the disk I/O is the performance bottleneck. However, for programs that incur little disk I/O, the performance bottleneck shifts to the main-memory hierarchy. Modern processors typically employ one or more data caches for improving access time to memory when there is a lot of spatial and/or temporal locality in the access patterns. The processor cache is analogous to a database's buffer cache, the primary difference being that the user does not have control over the replacement policy. Reading data from the processor's data cache is an order of magnitude faster than reading data from the main memory. Furthermore, as the speed of the processor increases, so does the main-memory latency (in terms of number of cycles). As a result, the latency of random memory accesses will only grow with future processors.

Linear time algorithms such as Ukkonen and McCreight require a large number of random memory accesses due to the linked list traversals through the tree structure. In Ukkonen, a majority of cache misses occur after traversing a suffix link to a new subtree and then examining each child of the new parent. The traversal of the suffix link to the sibling subtree and the subsequent search of the destination node's children require random accesses to memory over a large address space. Because this span of memory is too large to fit in the processor cache, each access has a very high probability of incurring the full main-memory latency. Similarly, McCreight's algorithm also traverses suffix links during construction, and incurs many cache misses. Furthermore, the rescanning and scanning steps used to find the extended locus of the head of the newly added suffix result in more random accesses. Using an array-based representation [91], where the pointers to the

children are stored in an array with an element for each symbol in the alphabet, can reduce the number of cache misses. However, this representation uses a lot of space, potentially leading to higher execution time. In previous work, both McCreight [103] and TOP-Q [16] argue for the linked list based implementation as being a better choice.

Observe that when using the linked list implementation, as the alphabet size grows, the number of children for each non-leaf node will increase accordingly. As more children are examined to find the right position to insert the next character, the number of cache misses also increases. Therefore, Ukkonen's method will incur an increasing number of processor cache misses with an increase in alphabet size. Similarly, with McCreight's algorithm, an increase in alphabet size leads to more cache misses.

For TDD, the alphabet size has the opposite effect. As the branching factor increases, the working set of the Suffixes and Temp arrays quickly decreases, and can fit into the processor cache sooner. The majority of read misses in the TDD algorithm occur when calculating the size of each character group (in Line 8 of Figure 3.2). This is because the beginning character of each suffix must be read, and there is little spatial locality in the reads. While both algorithms must perform random accesses to main memory, incurring very expensive cache misses, there are three properties about the TDD algorithm that make it more suited for in-memory performance: (a) the access pattern is sequential through memory, (b) each random memory access is independent of the other accesses, and (c) the accesses are known a priori. A detailed discussion of these properties can be found in [60]. Because the accesses to the input data string are sequential through the memory address space, hardware-based data prefetchers may be able to identify opportunities for prefetching the cache lines [75]. In addition, techniques for overlapping execution with main-memory latency can easily be incorporated in TDD.

The Deep-Shallow algorithm of [101] is a space efficient in-memory suffix array con-

struction technique. It differentiates the cases of sorting suffixes with a short common prefix from sorting suffixes with a long common prefix. These two cases are called "shallow" sorting and "deep" sorting respectively. The Bentley-Sedgewick multikey quick sort [17] is used as the shallow sorter, and a combination of different algorithms are used in the deep sorter. The memory reference pattern is different in the case of each algorithm, and a thorough analysis of the reference pattern is very complicated. This complex combination of different sorting strategies at different stages of suffix array construction turns out to perform very well in practice.

3.4.3 Effect of Alphabet Size and Data Skew

In this section, we consider the effect of alphabet size and data skew on TDD.

There are two properties of the input string that can affect the execution time of TDD: the size of the alphabet and the skew in the string. The average case running time for constructing a suffix tree on a Random Access Machine for *uniformly random input strings* is $O(n \log_{|A|} n)$, where $|A|$ is the size of the input alphabet and n is the length of the input string. (A uniformly random string can be thought of as a sequence generated by a source that emits each symbol in sequence from the alphabet set with equal probabilities, and the symbol emitted is independent of previous symbols.) The suffix tree has $O(\log_{|A|} n)$ levels [43], and at each level i, the suffixes array is divided into $i^{|A|}$ equal parts ($|A|$ is the branching factor, and the string is uniformly random.) The count-sort and the find-LCP (Line 7 of Figure 3.2) functions are called on each of these levels. The running time of count-sort is linear. To find the longest common prefix for a set of suffixes from a uniformly distributed string, the expected number of suffixes compared before a mismatch is slightly over 1. Therefore, the find-LCP function would return after just one or two comparisons most of the time. In some cases, the actual LCP is more than 1 and a scan of the entire suffixes is required. Therefore, in the case of uniformly random data, the

find-LCP function is expected to run in constant time. At each of the $O(log_{|A|}n)$ levels, the amount of computation performed is $O(n)$. This gives rise to the overall average case running time of $O(n \log_{|A|} n)$. The same average case cost can be shown to hold for random strings generated by picking symbols independently from the alphabet with *fixed* non-uniform probabilities. [9] shows that the height of trees on such strings is $O(\log n)$, and a linear amount of work is done at each level, leading to an average cost of $O(n \log n)$.

The longest common prefix of a set of suffixes is actually the label on the incoming edge for the node that corresponds to this set of suffixes. The average length of all the LCPs computed while building a tree is equal to the average length of the labels on each edge ending in a non-leaf node. This average LCP length is dependent on the distribution of symbols in the data. Real datasets, such as DNA strings, have a skew that is particular to them. By nature, DNA often consists of large repeating sequences; different symbols occur with more or less the same frequency but certain patterns occur more frequently than others. As a result, the average LCP length is higher than that for uniformly distributed data.

Figure 3.6 shows a histogram for the LCP lengths generated while constructing suffix trees on the SwissProt protein database [10] and the first 50 MB of Human DNA from chromosome 1 [57]. Notice that both sequences have a high probability that the LCP length will be greater than 1. Even among biological datasets, the differences can be quite dramatic. From the figure, we observe that the DNA sequence is much more likely to have LCP lengths greater than 1 compared with the protein sequence (70% versus 50%). It is important to note that the LCP histograms for the DNA and protein sequences shown in the figure are not representative of all DNA and protein sequences, but these particular results do highlight the differences one can expect between input datasets.

For data with a lot of repeating sequences, the find-LCP function will not be able to

complete in a constant amount of time. It will have to scan at least the first l characters of all the suffixes in the range, where l is the length of the actual LCP. In this case, the cost of find-LCP becomes $O(l \times r)$ where l is the length of the actual LCP, and r is the number of suffixes in the range that the function is examining. As a result, the PWOTD algorithm will take longer to complete.

TDD performs worse on inputs with many repeats such as DNA. On the other hand, Ukkonen's algorithm exploits these repeats by terminating an insert phase when a similar suffix is already in the tree. With long repeating sequences like DNA, this works in favor of Ukkonen's algorithm. Unfortunately, this advantage is not enough to offset the random reference pattern which still makes it a poor choice for large input strings when using cached architectures.

The size of the input alphabet also has an important effect. Larger input alphabets are an advantage for TDD because the running time is $O(n \log_{|A|} n)$, where $|A|$ is the size of the alphabet. A larger input alphabet size implies a larger branching factor for the suffix tree. This in turn implies that the working size of the Suffixes and Temp arrays shrinks more rapidly - and could fit into the cache entirely at a lower depth. For Ukkonen, a larger branching factor would imply that on an average, more siblings will have to be examined while searching for the right place to insert. This leads to a longer running time for Ukkonen. The same discussion also applies to McCreight's algorithm. There are hash-based and array-based approaches that alleviate this problem [91], but at the cost of consuming much more space for the tree. A larger tree representation naturally implies that for the in-memory case, we are limited to building trees on smaller strings.

Note that the case where Ukkonen's and McCreight's methods will have an advantage over TDD is for short input strings over a small alphabet size with high skew (repeat sequences). TDD is a better choice in all other cases. We experimentally demonstrate

these effects in Section 3.6.

3.5 The ST-Merge Algorithm

The TDD technique works very well so long as the input string fits into available main memory. In Section 3.6, we show that if the input string does not fit completely in memory, accesses to the string will incur a large number of random I/O. Consequently, for input strings that are significantly larger than the available memory the performance of TDD will rapidly degrade. In this section, we present a merge-based suffix tree construction algorithm that is more efficient than TDD when the input data string does not fit in main memory.

The ST-Merge algorithm employs a divide-and-conquer strategy similar to the external sort-merge algorithm. It is outlined in Figure 3.7 and shown in detail in Figure 3.8. While the ST-Merge algorithm can have more than one merge phase (as with sort-merge), here we only present a two-phase algorithm which has a single merge phase. (As with external sort-merge, in practice, this two-phase method is often sufficient with large main memory configurations.) At a high-level, the ST-Merge algorithm works as follows: To construct a suffix tree for a string of size n, the algorithm first partitions the set of n suffixes into k disjoint subsets. Then a suffix tree is built on each of these subsets. Next, the intermediate trees are merged to produce the final suffix tree.

Note that the partitioning step of ST-Merge can be carried out in any arbitrary way– in fact, we could randomly assign a suffix to one of k buckets. However, we choose to partition the suffixes such that a given subset will contain only contiguous suffixes from the string. As we will discuss in detail in Section 3.5.1, using this partition strategy, we have a very high locality of access to the string when constructing the trees on each partition.

In the merging phase, the references to the input string have a more clustered access pattern, which has a better locality of reference than TDD. In addition, the ST-Merge method permits a number of merge strategies. For example, all the trees could be merged in a single merge step, or alternatively trees can be merged incrementally, i.e., trees are merged one after another. However, the first approach is preferable as it reduces the number of intermediate suffix trees that are produced (which may be written to the disk).

For building the suffix trees on the individual partitions, the ST-Merge algorithm simply uses the PWOTD algorithm. The subsequent merge phase is more complicated, and is described in detail below.

There are two main subroutines used in the merge phase: *NodeMerge* and *EdgeMerge*. The merge algorithm starts by merging the root nodes of the trees that are generated by phase 1. This is accomplished by a call to NodeMerge. EdgeMerge is used by NodeMerge when it is trying to merge multiple nodes that have outgoing edges with a common prefix. The NodeMerge and EdgeMerge subroutines are shown in Figures 3.9 and 3.10, respectively.

The NodeMerge algorithm merges the nodes from the source trees and creates a merged node as the ending node of the parent edge in the merged suffix tree. Note that the parent edge of the merged node is NULL only when the roots of the source trees are merged. The NodeMerge algorithm first groups all the outgoing edges from the source nodes according to the first character along each edge, so that edges from each group share the same starting alphabet. If the alphabet set size is $|A|$, there are at most $|A|$ groups of edges. As the edges of each node are already sorted, replacement selection sort or count sort can be used to generate the groups. Next, the algorithm examines each edge group. If the edge group contains only one edge, then it implies that this edge along with the subtree below is a branch of the merged node in the merged suffix tree. In this case, the algorithm simply

copies the entire branch from the source tree to the merged tree. If a group contains more than one edge, the algorithm creates a new outgoing edge of the merged node. This step is carried out by calling EdgeMerge.

Note that NodeMerge will never need to merge a leaf node with an internal node. If such a case arose, it would mean that the suffix represented by the leaf node is a prefix of another suffix. This cannot happen since we add a terminating symbol to the end of the string to prevent this very case!

The EdgeMerge algorithm merges together multiple edges that start with the same symbol. It first finds the longest common prefix (LCP) of the set of edges. Then, it creates a new edge in the result tree and labels it with the LCP. If any of the source edges have labels longer than the LCP, the edges are artificially split by inserting a node after the LCP. All the nodes ending at LCP now can be merged together with a call to NodeMerge, since they are all at the end of edges labeled identically.

A detailed example of ST-Merge is shown in Figures 3.11 to 3.15.

3.5.1 Comparison with TDD

In this section, we present an analysis of the ST-Merge algorithm and discuss its relative advantages and disadvantages.

The main advantage for ST-Merge comes from the way it accesses the disk. In the partition and build phase, the algorithm accesses only a small portion of the string corresponding to that partition (the suffixes at the end of each partition may require accesses that spill across the partition boundary). This ensures that most accesses to the string are in memory if the buffer for the String is at least the size of the partition. This can be much smaller than the whole string, and can therefore save a large amount of I/O. In fact, the first phase of the algorithm typically takes an order of magnitude less time than TDD. In the second phase, the input trees and the output tree are all sequentially accessed. So,

allocating each tree requires only a small buffer. The remaining memory is allocated to the string. Compared to TDD, the accesses to the string in the second phase of ST-Merge have more spatial locality of reference. This is because the accesses to the string (driven by the trees from phase 1) result in a smaller working set.

The decision of how many partitions to use in the first phase can be made using a simple formula. Suppose that M is the total amount of memory available. Let n be the size of the input string. The number of partitions to be used in the first phase is given by $k = \lceil \frac{n \times f}{M} \rceil$, where f (> 1) is an adjustment multiplication factor to account for overhead associated with the memory required for the auxiliary data structures, which are proportional in size to the input string. When the amount of main memory is greater than the string size, partitioning does not provide much benefit, and we simply use TDD.

Now, we examine the worst case cost of the merge algorithm. The first phase is $O(n^2)$ in the worst case. The second phase has two components: the cost of merging the nodes, and the cost of merging the edges. In the worst case, each node in the output tree ($O(n)$ nodes) is a result of merging k nodes from the source trees. This involves sorting at most $|A| \times k$ edges. Any sorting algorithm can be used to group the edges– a count sort can do this in $O(|A| \times k)$ time. Therefore, the cost of merging the nodes is $O(n \times k)$ (assuming a constant sized alphabet). The cost of merging the edges is the sum of the lengths of the edge labels of the source trees. This is because each symbol on an edge is considered at most once. In the worst case, the length of an edge is $O(n)$. This yields a worst case cost of $O(n^2)$. Adding the three components, the worst case cost of ST-Merge is $O(n^2)$.

Next, we derive a loose bound for the average case cost assuming that the string is generated by a Bernoulli source (i.e. the characters are drawn from the alphabet independently with fixed probabilities). The first phase takes $O(n \log \frac{n}{k})$, with k partitions each taking time $O(\frac{n}{k} \log \frac{n}{k})$. The cost of merging the edges is $O(n \log \frac{n}{k})$ on average, since

the number of edges in the source trees totals $O(k \times \frac{n}{k})$, and the average length of the LCP is $O(\log \frac{n}{k})$ [9]. The worst case cost of merging the nodes serves as an upper bound for the average case cost. Adding the three components, the average cost of merging is $O(nk + n \log \frac{n}{k})$. As $k = \Theta(n)$, this is $O(n^2)$. Note that in practice with large main memory configurations, k is usually a small number, since $k = \lceil \frac{n \times f}{M} \rceil$, where M is the size of the memory.

It is important to note that since ST-Merge writes a set of intermediate trees (the trees generated for each partition in the first phase) and merges them together for the final tree, the amount of data it writes is approximately twice the amount written by TDD (assuming that phase 2 requires only a single pass). However, this disadvantage is offset by the fact that the amount of memory required by the string buffer is smaller for ST-Merge and this results in less random I/O. The exact effect of these two factors depends on the ratio of the size of the string to the amount of memory available. In Section 3.6.6, we compare the execution times of TDD and ST-Merge.

3.6 Experimental Evaluation

In this section, we present the results of an extensive experimental evaluation of the different suffix tree construction techniques. First, we compare the performance of TDD with Ukkonen's algorithm [156] and Kurtz's implementation [91] of McCreight's algorithm [103] for constructing in-memory suffix trees. For the in-memory case, we also compare these algorithms with an indirect approach that builds a suffix array first and converts the suffix array to a suffix tree. The suffix array method we choose is the Deep-Shallow algorithm [101], which is a fast, lightweight, in-memory suffix array construction algorithm. Then we compare TDD with Hunt's algorithm [73] for disk-based construction performance. We also evaluate the external DC3 algorithm [42], which is a fast disk-based

suffix array construction technique. Finally, we examine the performance of ST-Merge and TDD when the input string is larger than the available memory.

3.6.1 Experimental Setup and Implementation

Our TDD algorithm uses separate buffer caches for the four main structures: the string, the suffixes array, the temporary working space for the count sort, and the suffix tree. We use fixed-size pages of 8K for reading and writing to disk. Buffer allocation for TDD is done using the method described in Section 3.3.3. If the amount of memory required is less than the size of the buffer cache, then that structure is loaded into the cache, with accesses to the data bypassing the buffer cache logic. TDD was written in C++ and compiled with GNU's g++ compiler version 3.2.2 with full optimizations activated.

For an implementation of Ukkonen's algorithm, we use the version from [177]. It is a textbook implementation based on Gusfield's description [64] and is written in C. The algorithm operates entirely in main memory, and there is no persistence. The suffix tree representation uses 32 bytes per node.

For the McCreight's algorithm we use the implementation that is part of the MUMmer software package [148]. This version of McCreight's algorithm is both space and time efficient, and the tree representation requires 10.1 bytes on average per input character.

The implementation of the Deep-Shallow suffix array construction algorithm is from [39]. Since this algorithm only constructs a suffix array, to build a suffix tree we augmented this method with a method for converting the suffix array to a suffix tree. For the remainder of this section, we refer to this Deep-Shallow implementation for constructing suffix trees as Deep-Shallow*. The conversion from suffix arrays to suffix trees requires the construction of an LCP array. For this implementation, we used the GetHeight algorithm proposed in [85]. We implemented a simple linear algorithm for converting a suffix array to a suffix tree as described in [7].

Our C++ implementation of Hunt's algorithm is from the OASIS sequence search tool [105], which is part of a larger project called Periscope [112]. The OASIS implementation uses a shared buffer cache instead of the persistent Java object store, PJama [12], described in the original proposal [73]. The buffer manager employs the CLOCK replacement policy. The OASIS implementation performed better than the implementation described in [73]. This is not surprising since PJama incurs the overhead of running through the Java Virtual Machine.

To compare TDD with a disk-based suffix array construction method, we used the external DC3 algorithm [42]. For the external DC3 suffix array construction algorithm, we use the code provided in [51]. The external DC3 algorithm from [51] can support multiple disks, but for all the disk-based methods including DC3, we used only one disk.

For the disk-based experiments that follow, unless stated otherwise, all I/O is to raw devices; i.e., there is no buffering of disk blocks by the operating system, and all reads and writes to disk are synchronous (blocking). This provides an unbiased accounting of the performance for disk-based construction as operating system buffering will not (positively) affect the performance. Therefore, our results present the worst case performance for the disk-based construction methods. Using asynchronous writes is expected to improve the performance of our algorithm over the results presented. Each raw device accesses a single partition on one Maxtor Atlas 10K IV drive. The disk drive controller is an LSI 53C1030, Ultra 320 SCSI controller.

All experiments were performed on an Intel Pentium 4 processor with 2.8 GHz clock speed and 2 GB of main memory. This processor includes a two-level cache hierarchy. There are two first level caches, named L1-I and L1-D, that cache instructions and data respectively. There is also a single Level-2 (L2) cache that stores both instructions and data. The L1 data cache is an 8 KB, 4-way set-associative cache with a 64 byte line size.

The L1 instruction cache is a 12 K trace cache, 4-way set associative. The L2 cache is a 512 KB, 8-way, set-associative cache, also with a 128 byte line size. The operating system was Linux, kernel version 2.4.20.

The Pentium 4 processor includes 18 event counters that are available for recording micro-architectural events, such as the number of instructions executed [76]. To access the event counters, the *perfctr* library was used [117]. The events measured include: clock cycles executed, instructions and micro-operations executed, L2 cache accesses and misses, Translation Lookaside Buffer (TLB) misses, and branch mispredictions.

3.6.2 Implications of 64-bit Architectures

The implementation that we use for the evaluation presented in this section, is based on a 32-bit architecture. However, our code can easily be adapted to use 64-bit addressing. In this section, we briefly examine the impact of using 64-bit architectures, which can directly address more than 4GB of physical memory.

We first investigate the memory requirement of the data structures used in our algorithms. There are two types of pointers in the data structures. The first type is a *string pointer*, which points to a position in the input string. The second type of pointer is a *node pointer*, which points to another node in the suffix tree. For the pointer to the string position, a 64-bit integer representation is needed only when the string size is larger than 4G (2^{32}) symbols. For the pointers to nodes, a 64-bit integer representation is needed only if the number of array entries in the suffix tree structure is more than 4G. Note that if the string has less than 4G symbols, and the suffix tree has more than 4G entries, then we can use a 32-bit representation for the string pointer and a 64-bit representation for the node pointer.

A non-leaf node in the suffix tree (the *Tree* structure shown in Figure 3.1) has one string pointer and one node pointer, whereas a leaf node simply has one string pointer. In our

tree representation, in addition to the tree array, we have 2 bits per entry in the tree array to indicate whether the entry is a leaf or a non-leaf, and whether the entry is the right-most sibling (see Figure 3.1 for details). The bit overhead is not affected by the changes to the pointer representation.

With a 32-bit representation for both string and node pointers, the size of a non-leaf node is 8 bytes, and the size of a leaf-node is 4 bytes. Going to a 64-bit representation adds four bytes for each pointer type that is affected.

In addition to the actual suffix tree (the *Tree* structure shown in Figure 3.1), the suffix tree construction algorithm also uses two additional arrays, namely the *Suffixes* and *Temp* arrays. Both of them only contain string pointers. The size of the entries for both these arrays is 4 bytes with a 32-bit representation.

Note that TDD uses a partitioning method to construct the suffix trees (see Section 3.3 for details). This partitioning method constructs disjoint suffix trees based on the first few symbols of the suffixes (the $prefixlen$ variable in Figure 3.2). Since each disjoint suffix tree only contains node pointers that point to nodes within the subtree, even when the *total* number of entries in the system is more than 4G, as long as each subtree has less than 4G entries, the node pointers can continue to use 32-bit representation.

3.6.3 Comparison of the In-Memory Algorithms

To evaluate the performance of the TDD technique for in-memory construction, we compare with the $O(n)$ time algorithms of Ukkonen and McCreight, and the Deep-Shallow* algorithm. We do not evaluate Hunt's algorithm in this section as it was not designed as an in-memory technique.

For this experiment, we used six different datasets: chromosome 2 of Drosophila Melanogaster from GenBank [57], a slice of the SwissProt dataset [10] containing 20 million symbols, and the text dataset from the 1995 collection from project Gutenberg [119].

Data Source	**Description**	**Symbols** (10^6)
dmelano	D.Melanogaster Chr. 2 (DNA)	20
guten95	Gutenberg Project, Year 1995 (English Text)	20
swp20	Slice of SwissProt (Protein)	20
unif4	4-char alphabet, uniform distrib.	20
unif40	40-char alphabet, uniform distrib.	20
unif80	80-char alphabet, uniform distrib.	20

Table 3.1: Main Memory Data Sources

The DNA dataset has an alphabet size of 5 (4 nucleotides, and the character 'N' for unknown positions). The protein dataset has an alphabet size of 23 (for the 20 amino acids, one character for representing unknown, and two characters to represent combinations), and the text dataset uses an alphabet of size 61 (all uppercase characters, numbers, and punctuation marks). We also chose three strings that contain uniformly distributed symbols from an alphabet of size 4, 40, and 80. The datasets used in this experiment are summarized in Table 3.1.

Figure 3.16 shows the execution time breakdown for four algorithms, grouped by the datasets. In order, we present the times for TDD, Ukkonen, McCreight, and DeepShallow*. Note that since this is the in-memory case, TDD reduces to the PWOTD algorithm. In these experiments, all data structures fit into memory. The total execution time is decomposed into the time executing the following microarchitectural events (from bottom to top): instructions executed plus resource related stalls, TLB misses, branch mispredictions, L2 cache hits, and L2 cache misses (or main-memory reads).

From Figure 3.16, we observe that the L2 cache miss component is a large contributor to the execution time for all algorithms. All algorithms show a similar breakdown for the small alphabet sizes of DNA data (unif4 and dmelano). When the alphabet size increases from 4 symbols to 20 symbols for *swp20*, then to 40 symbols for *unif40*, and finally to 80 symbols for *unif80*, the cache miss component of the suffix link based algorithms (Ukko-

Data Source	SA (sec)	LCP (sec)	Conv (sec)	Total (sec)
unif4	9.32	9.34	5.09	24.03
dmelano	10.65	9.69	7.25	27.59
swp20	9.57	9.22	4.86	23.65
unif40	7.87	10.61	3.98	22.46
guten95	9.31	8.1	4.58	21.78
unif80	7.53	9.98	3.67	21.18

Table 3.2: Execution Time Details for Deep-Shallow*: Time spent by the algorithm in the three phases – suffix array construction (SA), LCP array construction (LCP), and suffix array to suffix tree conversion (Conv).

nen and McCreight) increases dramatically, while it remains low for TDD. The reason for this, as discussed in Section 3.4.2, is that these algorithms incur a lot of cache misses while following the suffix link to a new portion of the tree, and in traversing all the children when trying to find the right position to insert the new entry. The suffix array based method, Deep-Shallow*, does not exhibit this increase.

We observe that for each dataset, TDD outperforms the implementation of Ukkonen's algorithm that we use, and the performance difference increases with the alphabet size. This behavior was expected based on discussions in Section 3.4.3. TDD is faster than Ukkonen's method by a factor of 2.5 (*dmelano*) to 16 (*unif80*). TDD also outperforms McCreight's algorithm for *swp20*, *unif40*, *guten95*, and *unif80* by a factor of 2.7, 6.2, 1.5, and 10.9 respectively. On the other two datasets, *unif4* and *dmelano*, the performance is nearly the same. Interestingly, the suffix array based method, Deep-Shallow*, performs roughly as well as TDD. For the Deep-Shallow* algorithm, Table 3.2 shows the actual times spent in each of the three phases of the algorithm.

Collectively, these results demonstrate that despite having a $O(n^2)$ time cost, the TDD technique significantly outperforms the implementations of the linear time algorithms of Ukkonen and McCreight on cached architectures. It does not, however, have any significant advantage over the suffix array based Deep-Shallow* algorithm.

We must caution the reader, however, that this superior performance of TDD is not guaranteed in all cases. There may be inputs with a small alphabet size and a high amount of skew on which Ukkonen or McCreight could out-perform TDD, despite being less cache-efficient.

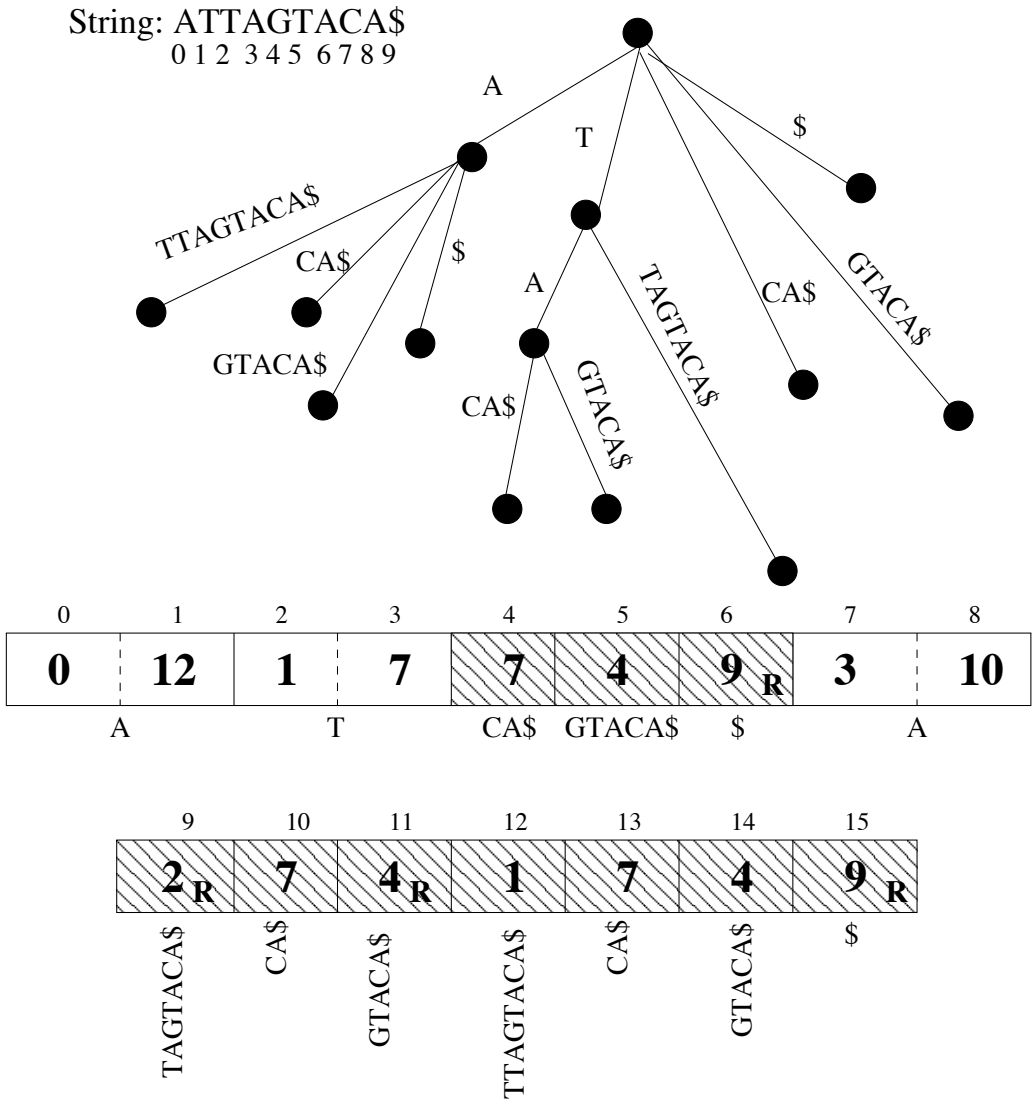

Figure 3.1: Suffix Tree Representation (Leaf nodes are shaded, the rightmost child is denoted with an R)

Algorithm PWOTD(*String*,*prefixlen*)
Phase 1:
Scan the *String* and partition *Suffixes* based
on the first *prefixlen* symbols of each suffix
Phase 2: Do for each partition:
1. START BuildSuffixTree
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on first symbol using *Temp*
4. Output branching and leaf nodes to the *Tree*
5. Push the nodes pointing to an unevaluated range
   onto the *Stack*

While *Stack* is not empty
   6. Pop a node
   7. Find the Longest Common Prefix (LCP) of
      all the suffixes in this range by checking
      the *String*
   8. Sort the range in *Suffixes* on the first
      symbol using *Temp*
   9. Write out branching nodes or leaf nodes to *Tree*
   10. Push the nodes pointing to an unevaluated range
      onto the *Stack*

11. END

Figure 3.2: The TDD Algorithm

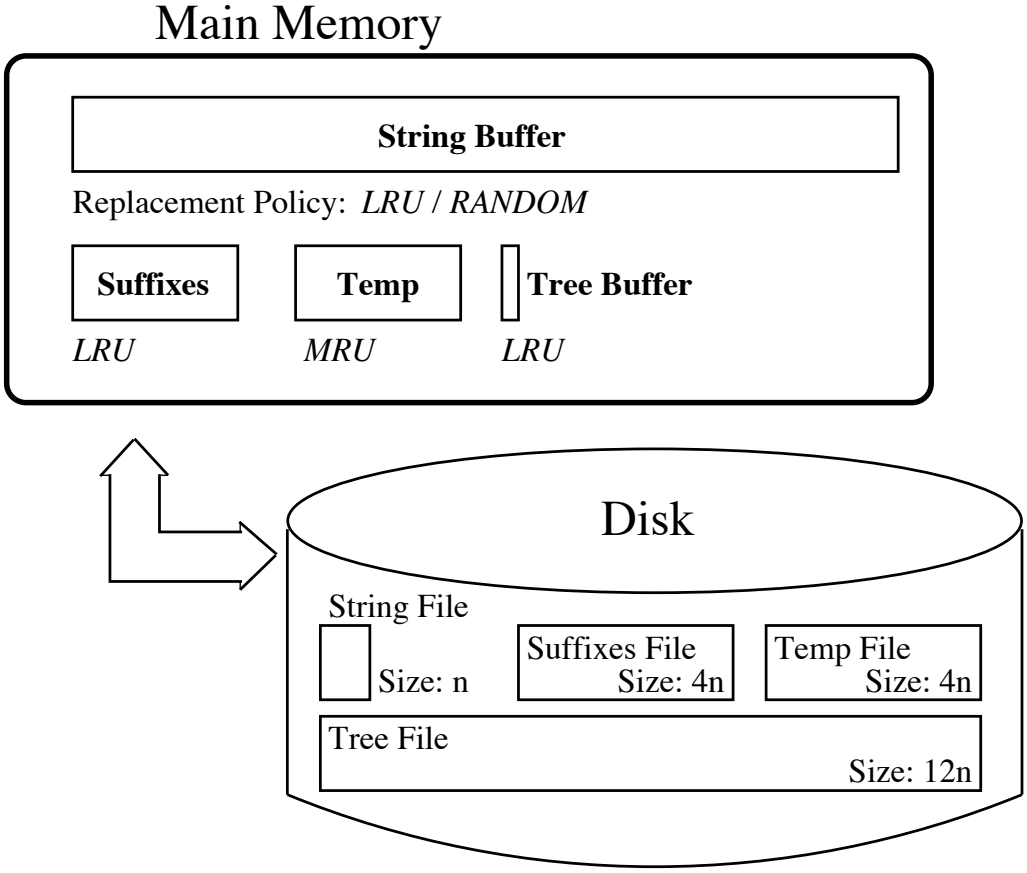

Figure 3.3: Buffer Management Schema

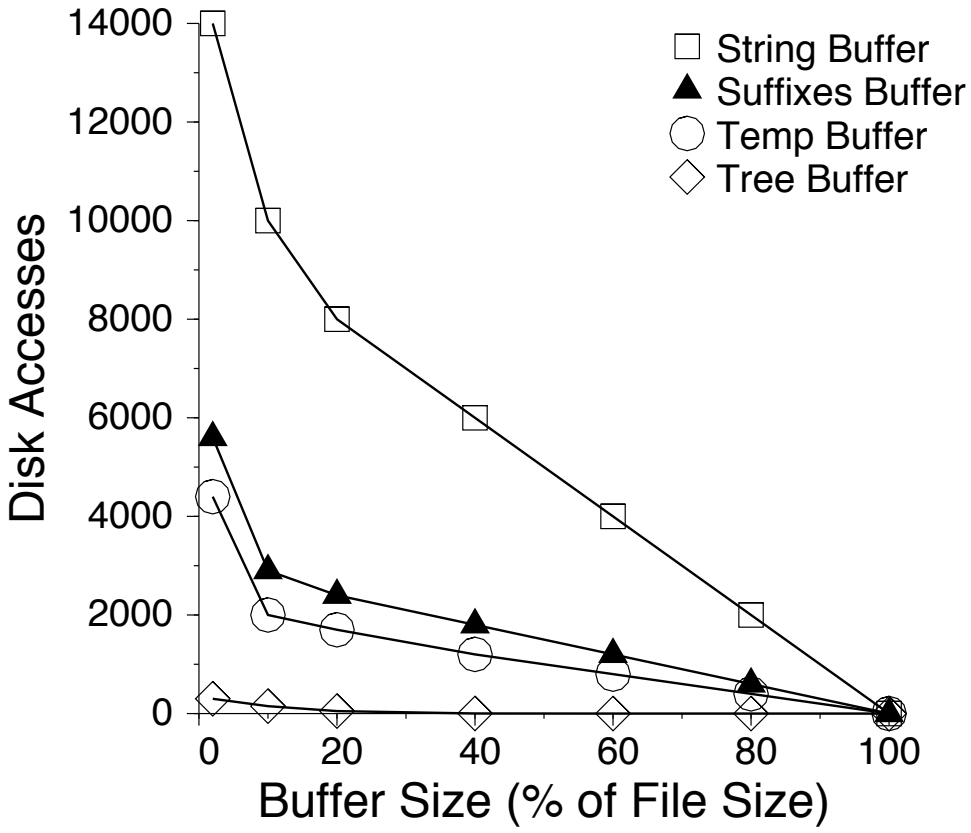

Figure 3.4: Sample Page Miss Curves

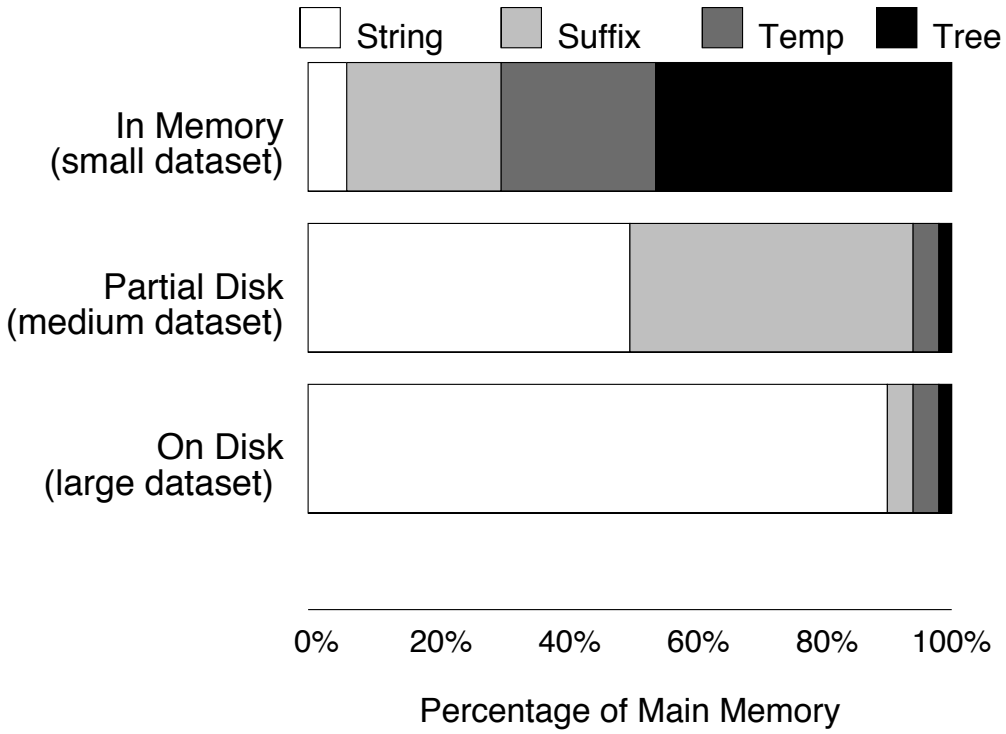

Figure 3.5: Buffer Allocation for Different Data Structures: Note how other data structures are gradually pushed out of memory as the input string size increases.

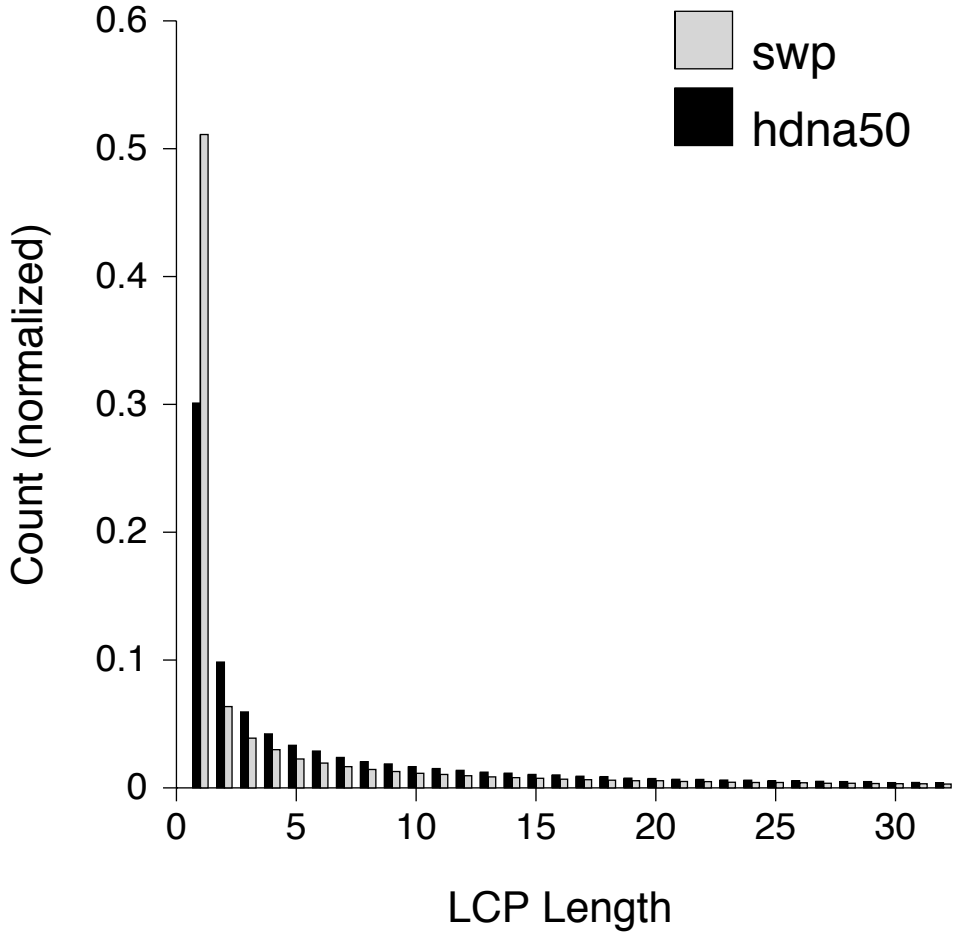

Figure 3.6: LCP Histogram: This figure plots the histogram until an LCP length of 32. For the DNA dataset, 18.8% of the LCPs have a length greater than 32, and for the protein data set 13.8% of the LCPs have a length greater than 32.

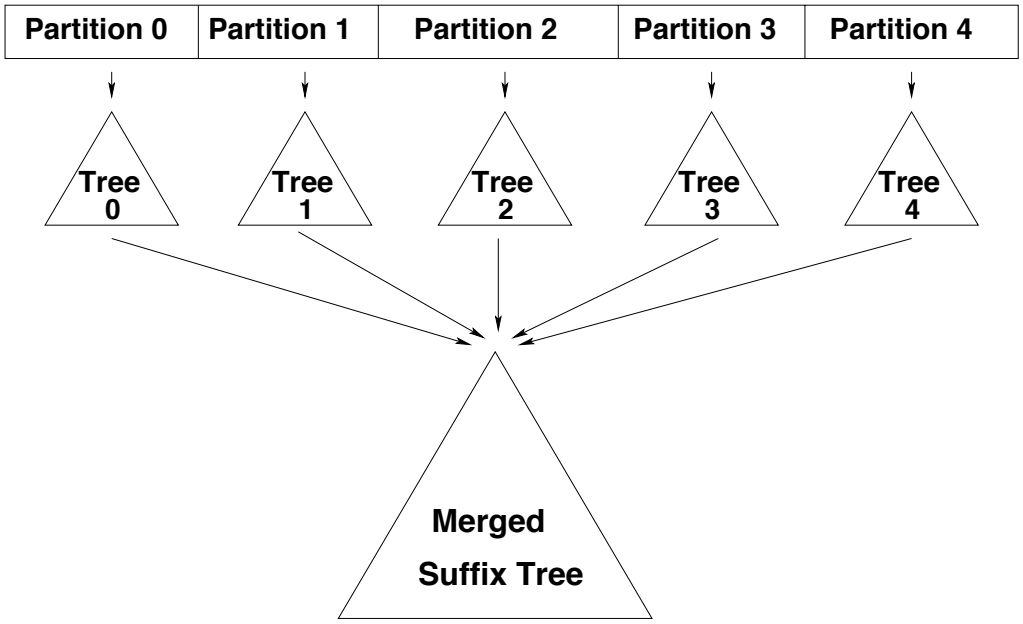

Figure 3.7: The Scheme for ST-Merge

Algorithm ST-Merge
Phase 1
1. Partition the string into k partitions
2. Build trees on each partition using TDD
Phase 2
1. NodeSet={}
2. For each tree to be merged
    3. Add the root of the tree to NodeSet
4. End For
5. NodeMerge(NodeSet, NULL)

Figure 3.8: The ST-Merge Algorithm

```
NodeMerge(NodeSet,ParentEdge)
1. If ParentEdge == NULL
   2. Create a new node N for the merged tree
3. Else
   4. Create a new node N at the end of ParentEdge
5. Group the edges from the nodes in the NodeSet
   by the first character on each edge using a sort.
6. For each edge group
   7. If the group contains one edge
     8. Copy the edge and the subtree below
        from the corresponding source tree to
        node N of the merged tree
   9. Else
     10. EdgeSet={edges in the edge group}
     11. EdgeMerge(EdgeSet, N)
   12. End If
13.End For
```

Figure 3.9: The NodeMerge Subroutine

```
EdgeMerge(EdgeSet, ParentNode)
1. Find the longest common prefix LCP of edges in EdgeSet
2. Create a new edge E, labeled with the LCP,
   as one outgoing edge of ParentNode
3. NodeSet={}
4. For each edge in EdgeSet
   5. If LCP is a proper prefix of the edge
     6. Create a new node to the corresponding tree,
        which breaks the edge at the end of LCP
     7. Add the new node to NodeSet
   8. Else
     9. Add the ending node of the edge to NodeSet
   10. End if
11. End For
12. NodeMerge(NodeSet, E)
```

Figure 3.10: The EdgeMerge Subroutine

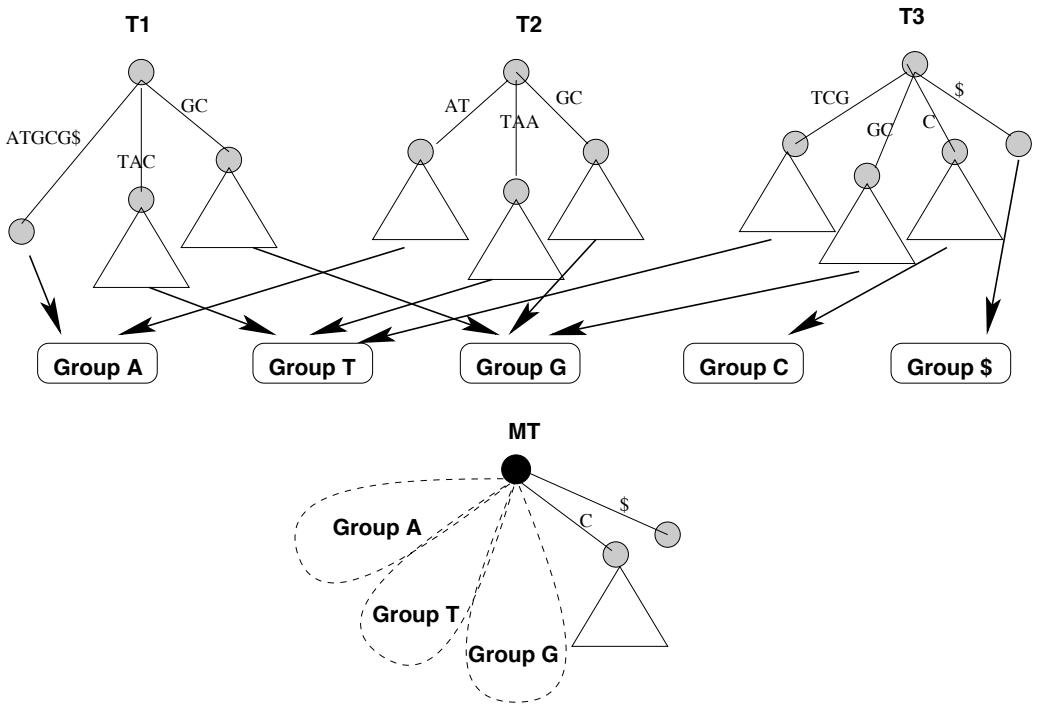

Figure 3.11: Example of Trees Being Merged

T1, T2 and T3 are three source trees to be merged. The final merged tree is MT. The triangle below a node represents the subtree under that node. The algorithm starts by calling NodeMerge on the trees T1–T3, which creates a root node for MT and groups the edges of the source trees according to the first character of each edge. This step produces five groups. Group A, T and G all contain more than one edge, so EdgeMerge is called for each of these groups, whereas group C and $ only have one edge, so the corresponding branches are copied to MT.

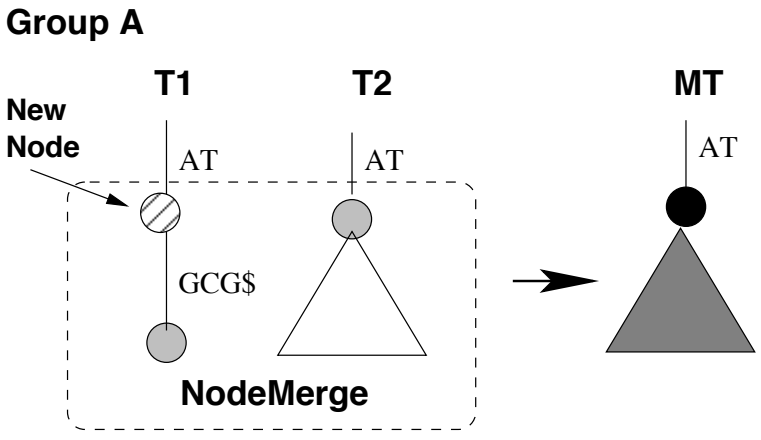

Figure 3.12: EdgeMerge for Group-A

We first create one outgoing edge from MT's root node, and label it with the LCP of the edges in group A. As the edge from T1 is longer than the LCP, we insert a new node in the middle of the long edge of T1 to split it into two edges labeled AT and GCG$ respectively. Then NodeMerge is called on the newly created node in T1 and the node in T2 at the end of the label AT. NodeMerge then produces a node at the end of the edge AT in MT, as well as the sub tree below it.

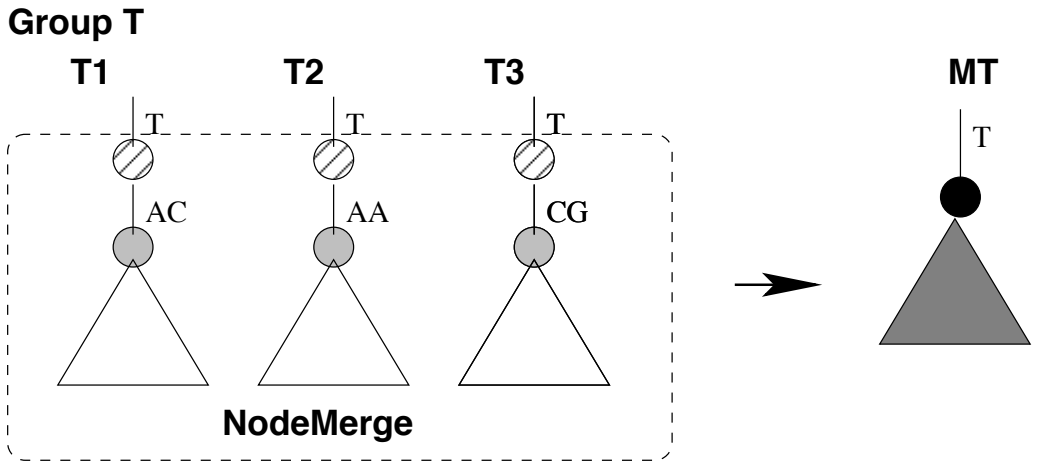

Figure 3.13: EdgeMerge for Group-T

The LCP of the edges in this group is a proper prefix of every edge, so we insert a node at the end of the LCP into every edge. The newly created nodes are then merged by making a call to NodeMerge.

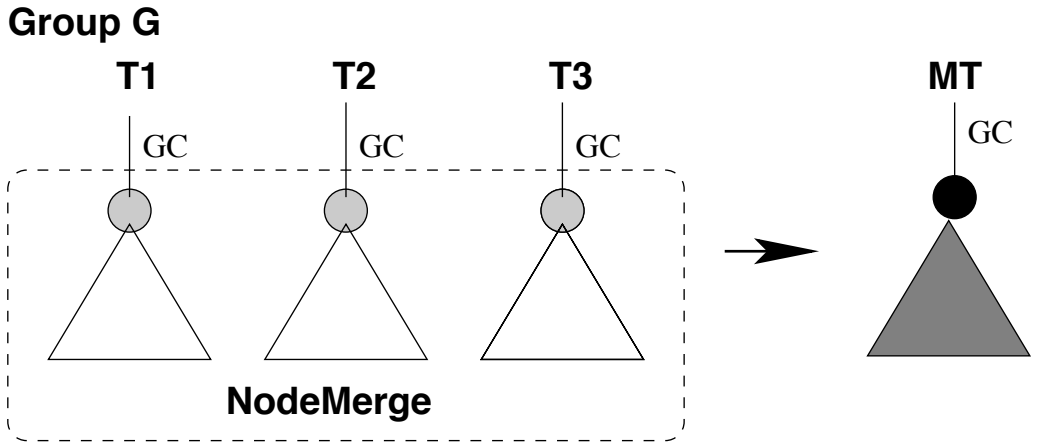

Figure 3.14: EdgeMerge for Group-G

All the edges are the same in this group. Consequently, the corresponding nodes ending at these edges are merged by making a call to NodeMerge.

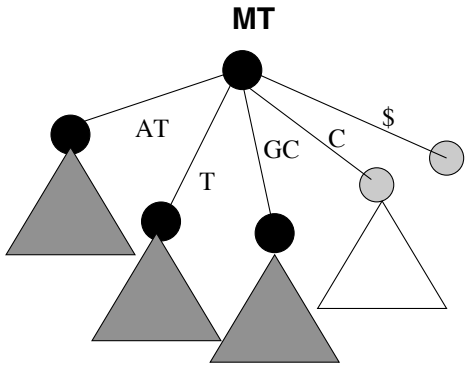

Figure 3.15: The Result of the Merge

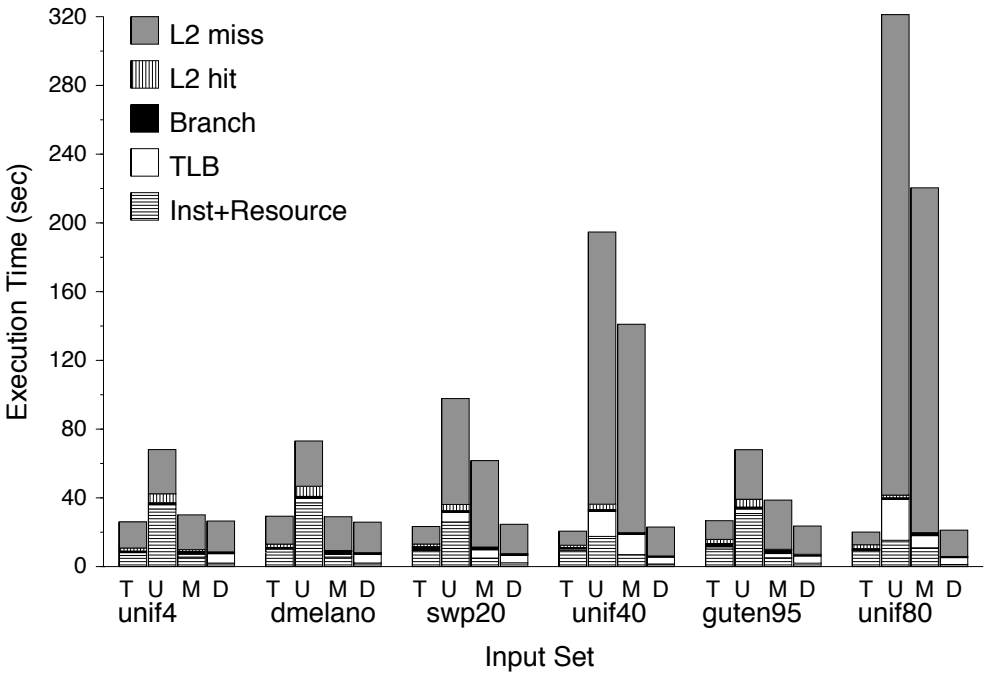

Figure 3.16: In-Memory Execution Time Breakdown for TDD, Ukkonen, McCreight, and Deep-Shallow*

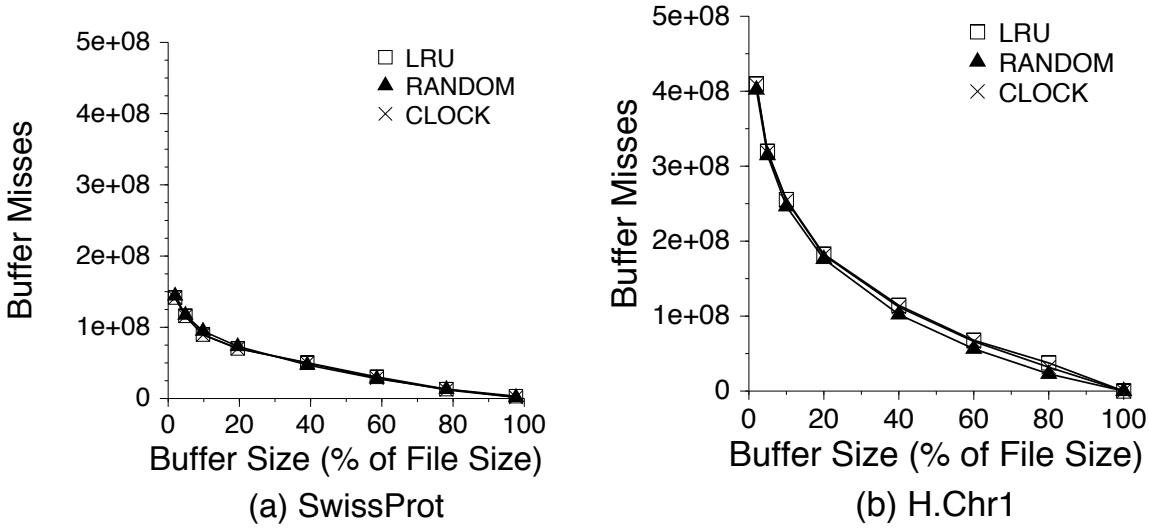

Figure 3.17: String Buffer

3.6.4 Buffer Management with TDD

In this section we evaluate the effectiveness of various buffer management policies on TDD. For each data structure used in the TDD algorithm, we analyze the performance of the LRU, MRU, RANDOM, and CLOCK page replacement polices over a wide range of buffer cache sizes. To facilitate this analysis over the wide range of variables, we employed a buffer cache simulator. The simulator takes as input a trace of the address requests into the buffer cache and the page size. The simulator outputs the disk I/O statistics for the desired replacement policy. For all the results shown here, except for the Temp array, MRU performs the worst by far and is not shown in the figures that we present in this section.

To generate the address request traces, we built suffix trees on the *SwissProt* database [10] and a 50 Mbps slice of the *Human Chromosome*-1 database [57]. A *prefixlen* of 1 was used for partitioning in the first phase. The total size of each of the arrays for these datasets is summarized in Table 3.3.

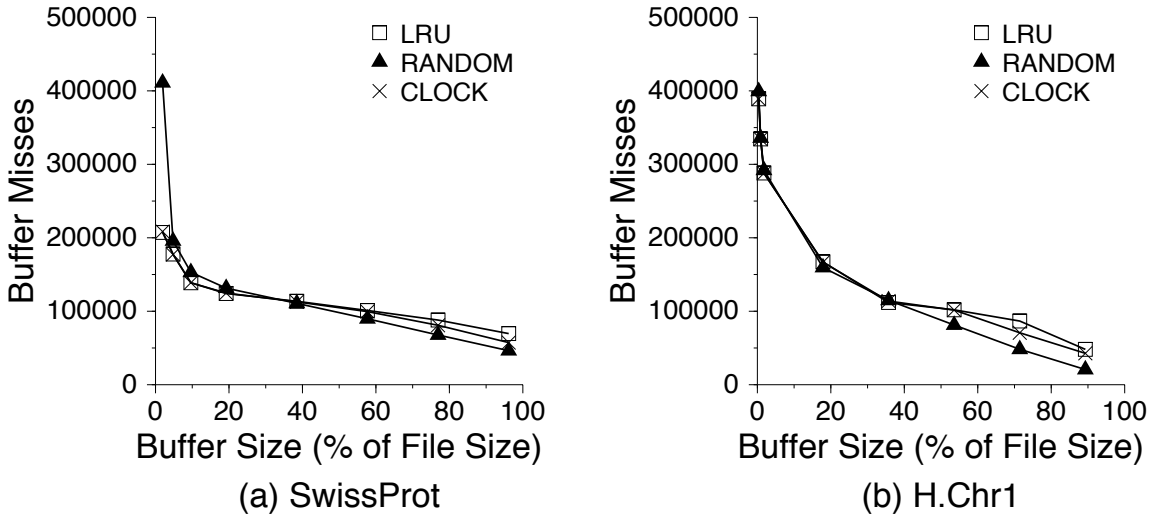

Figure 3.18: Suffix Buffer

Data Structure	SwissProt (size in pages)	Human DNA (size in pages)
String	6,250 (50 MB)	6,250 (50 MB)
Suffixes	1,250 (10 MB)	6,250 (10 MB)
Temp	1,250 (10 MB)	6,250 (50 MB)
Tree	4,100 (32.8 MB)	16,200 (129.6 MB)

Table 3.3: The On-Disk Sizes of each Data Structure

Page Size

In order to determine the page size to use for the buffers, we conducted several experiments. We observed that larger page sizes produced fewer page misses when the alphabet size was large (protein datasets, for instance). Smaller page sizes seemed to have a slight advantage in the case of input sets with smaller alphabets (like DNA sequences). We observed that a page size of 8192 bytes performed well for a wide range of alphabet sizes. In the interest of space, we omit the details of our page-size study. For all the experiments described in this section we use a page size of 8 KB.

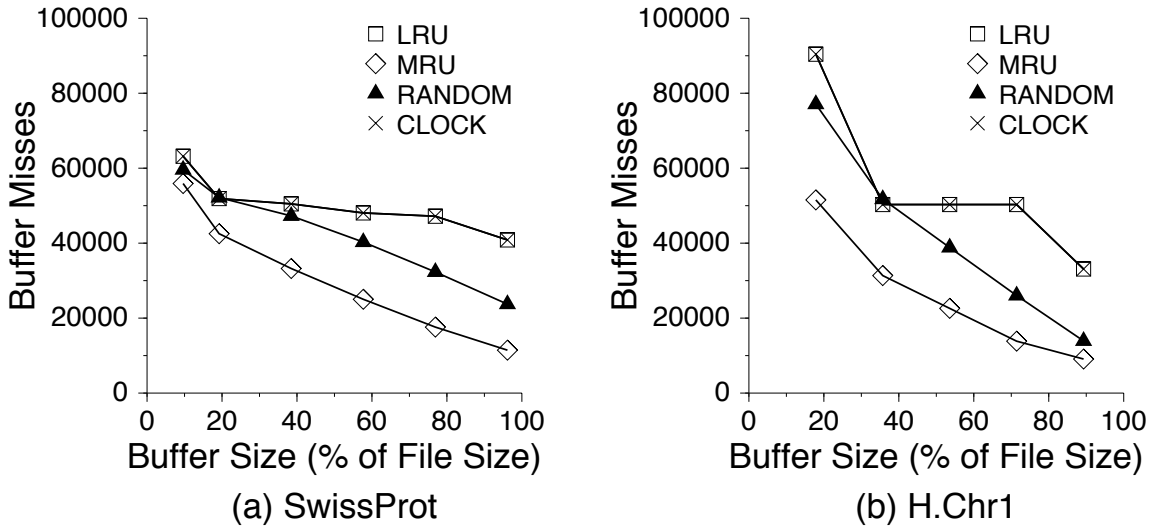

Figure 3.19: Temp Buffer

Buffer Replacement Policy

The results showing the effect of the various buffer replacement policies for the four data structures are presented in Figures 3.17 to 3.20. In these figures, the x-axis is the buffer size (shown as a percentage of the original input string size), and the y-axis is the number of buffer misses that are incurred by various replacement policies.

From Figure 3.17, we observe that for the String buffer LRU, RANDOM, and CLOCK all perform similarly. Of all the arrays, when the buffer size is a fixed fraction of the total size of the structure, the String incurs the largest number of page misses. This is not surprising since this structure is accessed the most and in a random fashion. RANDOM and LRU are both good choices for the String buffer.

In the case of the Suffixes buffer (shown in Figure 3.18), all three policies perform similarly for small buffer sizes. In the case of the Temp buffer, the reference pattern consists of one linear scan from left to right to copy the suffixes from the Suffixes array, and then another scan from left to right to copy the suffixes back into the Suffixes array in the sorted order. Clearly, MRU is the best policy in this case as shown by the results in

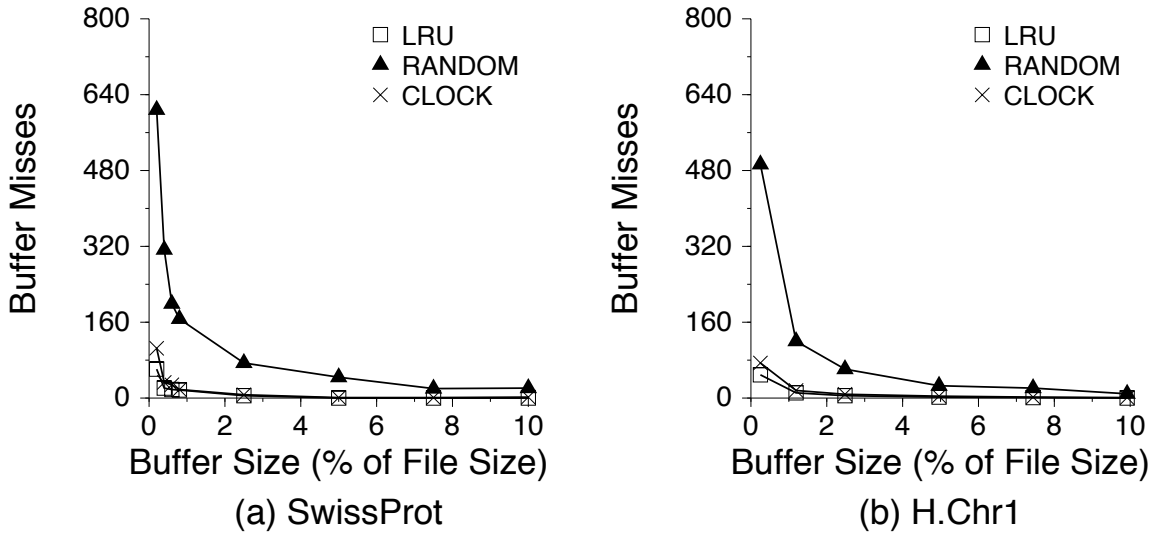

Figure 3.20: Tree Buffer

Figure 3.19. It is interesting to observe that the space required by the Temp buffer is much smaller than the space required by the Suffixes buffer to keep the number of misses down to the same level, though the array sizes are the same.

For the Tree buffer (see Figure 3.20), with very small buffer sizes, LRU and CLOCK outperform RANDOM. However, this advantage is lost for even moderate buffer sizes. The most important observation to be made here is that despite being the largest data structure, it requires the smallest amount of buffer space, and takes a relatively insignificant number of misses for any policy. Therefore for the Tree buffer, we can choose to implement the cheapest policy - the RANDOM replacement policy.

3.6.5 Comparison of Disk-based Algorithms

In this section we first compare the performance of our technique with the technique proposed by Hunt et al. [73], which is currently considered the best practical disk-based suffix tree construction approach. We also compare the performance of TDD with the DC3 suffix array construction method [51]. Note that the DC3 method only constructs a suffix array and not the suffix tree. However, these results provide a lower bound on the cost of

Data Source	**Description**	**Symbols** (10^6)
swp	Entire UniProt/SwissProt (Protein)	53
H.Chr1-50	50 Mbps slice of Human Chromosome-1 (DNA)	50
guten03	2003 Directory of Gutenberg Project (English Text)	58
trembl	TrEMBL (Protein)	338
H.Chr1	Entire Human Chromosome-1 (DNA)	227
guten	Entire Gutenberg Collection (English Text)	407
HG	Entire Human Genome (DNA)	3,000

Table 3.4: On-Disk Data Sources

Data Source	**Symbols** (10^6)	**Hunt (min)**	**TDD (min)**	**Speed -up**	**DC3** (min)
swp	53	13.95	2.78	5.0	12.60
H.Chr1-50	50	11.47	2.02	5.7	12.67
guten03	58	22.5	6.03	3.7	13.78
trembl	338	236.7	32.00	7.4	102.78
H.Chr1	227	97.50	17.83	5.5	74.57
guten	407	463.3	46.67	9.9	120.53
HG	3,000	—	30hrs	—	—

Table 3.5: On-Disk Performance Comparison

constructing a disk-based suffix tree using a suffix array construction method.

For this experiment, we used seven datasets which are described in Table 3.4. The construction times for the three algorithms are shown in Table 3.5.

From Table 3.5, we see that in each case TDD significantly outperforms Hunt's algorithm. On the TrEMBL dataset, TDD is faster by a factor of 7.4. For Human Chromosome-1, TDD is faster by a factor of 5.5. For a large text dataset like the Gutenberg Collection, TDD is nearly ten times faster! For the largest dataset, the human genome, Hunt's algorithm did not complete in a reasonable amount of time. TDD finishes in less than 30 hours. The 3 billion symbols of the human genome can be in memory if we use 4 bits per symbol, which is what was used to obtain the number in Table 3.5. The reason why TDD performs better is that Hunt's algorithm traverses the on-disk tree during construction, while TDD

does not. During construction, a given node in the tree is written at most once in TDD. In addition, the careful management of the buffer sizes and the separate buffer replacement policies help reduce the disk I/O costs for TDD even further.

Next, we compare TDD with the fastest known disk-based suffix array construction algorithm – the disk-based DC3 algorithm [42]. These results are shown in Table 3.5. From Table 3.5, we can see that TDD is more than twice as fast as the external DC3 method in all cases. For HG, DC3 did not complete successfully. When the cost of building the LCP array and converting the suffix array to a suffix tree is added, the cost of this approach will be even higher (the number for the external DC3 algorithm in Table 3.5 only includes the time to build the suffix array). The suffix array construction algorithm works by recursively splitting the set of suffixes into a "two thirds" array (for suffixes starting at positions i such that $i \; mod \; 3 \neq 0$) and a "one thirds" array (for suffixes starting at positions i such that $i \; mod \; 3 = 0$). The larger array is sorted using radix sort, essentially giving lexicographic names to triples of symbols in the suffix. If there are two suffixes that cannot be distinguished by radix sort at this level, then an additional level of recursion is used where the lexicographic name is derived from three times as many symbols, and so on. The smaller array is sorted using the information from the "two thirds" array and then merged to this larger array using a fairly simple merge algorithm. In this algorithm, a large amount of random I/O is incurred during the radix sort. In addition, the amount of random I/O quickly increases as the recursion proceeds to a deeper level. This can happen very frequently with biological sequences where long repeats are common and deeper recursion is required to sort suffixes with longer LCPs.

Comparison of TDD with TOP-Q Recently, Bedathur and Haritsa have proposed the TOP-Q technique for constructing suffix trees [16]. TOP-Q is a new low overhead buffer man-

agement method which can be used with Ukkonen's construction algorithm. The goal of the TOP-Q approach is to invent a buffer management technique that does not require modifying an existing in-memory construction algorithm. In contrast, TDD and Hunt's algorithm [73] take the approach of modifying existing suffix tree construction algorithms to produce a new disk-based suffix tree construction algorithm. Even though the research focus of TOP-Q is different from TDD and Hunt's algorithm, it is natural to ask how the TOP-Q method compares to these other approaches.

To compare TDD with TOP-Q, we obtained a copy of the TOP-Q code from the authors. This version of the code only supports building suffix tree indices on DNA sequences. As per the recommendation in [16], we used a buffer pool of 880 MB for the internal nodes and 800 MB for the leaf nodes (this was the maximum memory allocation possible with the TOP-Q code). On 50 Mbp of Human Chromosome-1, TOP-Q took about 78 minutes. By contrast, under the same conditions, TDD took about 2.1 minutes: faster by a factor of 37. On the entire Human Chromosome-1, TOP-Q took 5800 minutes, while our approach takes around 18 minutes. In this case, TDD is faster by two orders of magnitude!

Comparison of TDD with DynaCluster The DynaCluster algorithm [31] is based upon Hunt's algorithm and tries to group nodes that are frequently referenced by each other into one cluster. The clusters are recursively created in a top-down fashion and a depth-first order. By using a dynamic clustering technique, DynaCluster reduces the random accesses to the suffix tree during construction time. However, just as in TOP-Q, DynaCluster is also inherently disadvantaged because they use clustering to improve what is a highly random reference pattern (on a large structure) to start with.

This is highlighted in the following comparison of I/O costs. In one of their experiments in [31], the authors constructed the suffix tree for Human Chromosome-1 (224 MB) with

a total of 864 MB of available memory. The I/O cost of this experiment is more than 800 seconds on their experimental platform. For computing the I/O costs, the authors used simulated disk numbers. Based on their method and the parameters in their paper (30 MB/s transfer rate, 8 KB pages), 800 seconds translates to 3 million disk reads/writes. For the same dataset and with identical parameters, TDD incurs 0.5 million page accesses, which is around a sixth of that incurred by DynaCluster. This directly translates to a clear advantage for TDD.

3.6.6 Constructing Suffix Trees on Very Large Inputs

In the previous section, we saw that TDD outperformed the other methods. The ST-Merge algorithm has advantages over TDD when the input string size is much larger than the main memory available ($\frac{n}{M} >> 1$). When the input string fits in memory, ST-Merge is the same as the TDD algorithm.

Figure 3.21 shows the execution times of TDD and ST-Merge when the data string is much larger than the available main memory. To keep the running times for this experiment measurable, for this experiment only, we limited the total memory available to the algorithms to 6 MB, and varied the size of the input string from 10 MB to 80 MB. The other experimental conditions are the same as before. We note that our main motivation for using a small amount of main memory for this experiment is primarily to keep this experiment manageable. As can be seen in Figure 3.21, even in this "scaled down" setting the execution time for the algorithms is very large - using a larger dataset with a larger amount of memory would have taken many days or weeks for each run. The "scaled down" setting exposes the behavior of these algorithms, while keeping the run times for the algorithms reasonable.

From Figure 3.21, we observe, that when the input data string is significantly larger (about 3 times or more) than the main memory size, the ST-Merge algorithm starts to

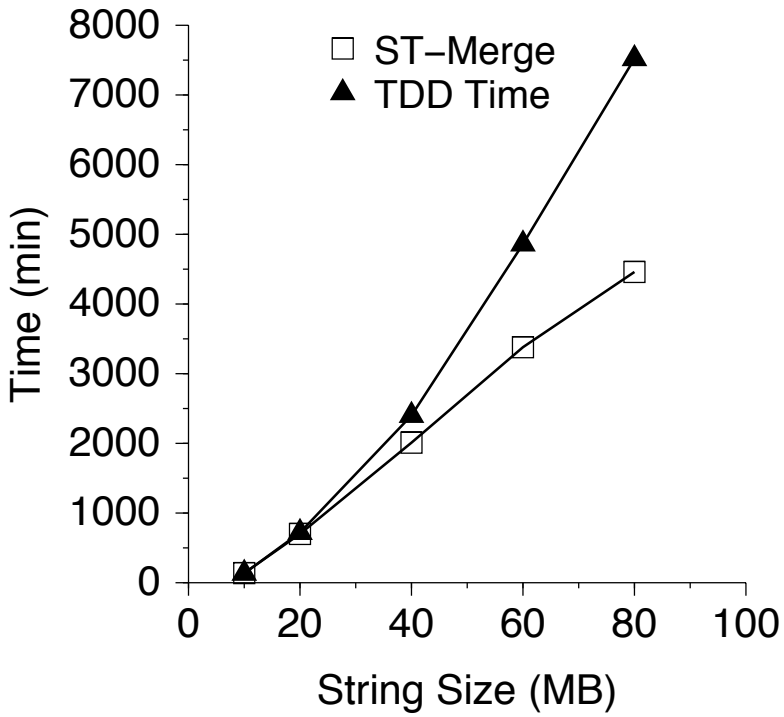

Figure 3.21: Execution Times : TDD and ST-Merge

outperform the TDD algorithm. We also observe that as the ratio $\frac{n}{M}$ increases, the ST-Merge algorithm has a larger advantage over TDD. This is expected because TDD incurs an increasingly larger penalty from the random I/O on the string. Consequently, for very large datasets, in which case the input string is significantly larger than the available main memory, ST-Merge is clearly the algorithm of choice.

3.7 Conclusions

Practical methods for suffix tree construction on large character sequences have been virtually intractable. Existing approaches have excessive memory requirements and poor locality of reference and therefore do not scale well for even moderately sized datasets.

We first compare different algorithms used for constructing suffix trees in-memory. We demonstrate that our method (which is essentially PWOTD for the in-memory case)

has an advantage over Ukkonen's algorithm by a factor of 2.5 to 16. It is also better than McCreight in some cases by up to a factor of 10. We argue that PWOTD wins over Ukkonen and McCreight because of superior cache performance. We also show that PWOTD is competitive with the suffix array based Deep-Shallow* algorithm and takes nearly the same time on various inputs.

To address the problem of disk-based suffix tree construction and unlock the potential of this powerful indexing structure, we have introduced the "Top Down Disk-based" (TDD) technique. The TDD technique includes the suffix tree construction algorithm (PWOTD), and an accompanying buffer management strategy.

Extensive experimental evaluations show that TDD scales gracefully as the dataset size increases. The TDD approach lets us build suffix trees on large frequently used sequence datasets such as UniProt/TrEMBL [10] in a few minutes. The TDD approach outperforms a popular disk-based suffix tree construction method (the Hunt's algorithm) by a factor of 5 to 10. In fact, to demonstrate the strength of TDD, we show that using slightly more main-memory than the input string, a suffix tree can be constructed on the *entire Human Genome in 30 hours on a single processor machine!* These input sizes are significantly larger than the datasets that have been used in previously published approaches.

In this chapter, we also compared TDD with a recently proposed disk-based suffix array construction method [42], and show that TDD also outperforms this method.

Even though TDD far outperforms the existing suffix tree construction algorithms, TDD degrades in performance when the input data string is much larger than the amount of main memory. To address this case, we have also proposed a new merge-based suffix tree algorithm called ST-Merge. The TDD algorithm can be seen as a special case of the ST-Merge algorithm when the number of merge partitions is equal to one. We have implemented ST-Merge, and demonstrated its benefits over TDD.

CHAPTER IV

Selectivity Estimation and Optimization

4.1 Introduction

Life science researchers today are faced with the problem of querying and mining large sequence datasets. There are several large databases worldwide that store protein and DNA sequence information. Some of these databases are growing very fast. For instance, GenBank, a repository for genetic information has been doubling every 16 months [63] – a rate faster than Moore's law! Protein databases, such as PDB [65] and PIR [10, 169] have also grown rapidly in the last few years.

Biologists try to analyze these databases in several complex ways. Similarity search is an important operation that is often used for both protein and genetic databases, although the way in which similarity search is used is different in each case. When querying protein databases, the goal is often to find proteins that are similar to the protein being studied. Studying a similar protein can yield important information about the role of the query protein in the cell. The computational criteria for specifying similarity is approximate, and includes similarity based on the amino acid sequence of the protein, or similarity based on the geometrical structure of the protein, or a combination of these. With genetic databases, scientists perform approximate similarity searches to identify regions of interest such as genes, regulatory markers, repeating units, etc. For any approximate matching query, the

desired output is an ordered list of results.

We note that existing sequence search tools such as BLAST [5, 6] only provide a limited search functionality. With BLAST one can only search for approximate *hits* to a single query sequence. One cannot look for more complex patterns such as one query sequence separated from another query sequence by a certain distance, or a query sequence with some constraints on other non-sequence attributes. Consider the following query: "*Find all genes in the human genome that are expressed in the liver and have a* TTGGACAGGATCCGA *(allowing for 1 or 2 mismatches) followed by* GCCGCG *within 40 symbols in a 4000 symbol stretch upstream of the gene"*. This is an instance of a relatively straightforward, yet important query that can be quite cumbersome to express and evaluate with current methods. One could code a specific query plan for this query in a scripting language. For example, the query plan may first perform a BLAST [5, 6] or Smith-Waterman [136] search to locate all instances of the two query patterns on the human genome. Then, the results of these matches can be combined to find all pairs that are within 40 symbols of each other. Next, a gene database can be consulted to check if this match is in the region upstream of any known gene. Finally, another database search would be required to check if the gene is expressed in the liver. Note that there are several other ways of evaluating this query, which may be more efficient. Moreover, current tools do not permit expressing such queries declaratively, and force the script programmer to pick and encode a query plan. Researchers frequently ask such queries and current procedural methods are quite cumbersome to use and reuse.

In this chapter, we describe a system called Periscope/SQ, which takes on the challenge of building a declarative and efficient query processing tool for biological sequences. The system makes it possible to declaratively pose queries such as the one described above. We also describe techniques to efficiently evaluate such queries, and using a real world

example, demonstrate that Periscope/SQ is faster than current procedural techniques by over two orders of magnitude!

Periscope/SQ is part of a larger research project - called Periscope - which aims to build a declarative and efficient query processing engine for querying on *all* protein and genetic structures [113]. For proteins the structures include not only sequence structure but also various geometrical structures that describe the shape and 3D structure of the protein. The SQ component stands for "Sequence Querying" and is our focus.

The main contributions of in this chapter are as follows:

- We identify the need for an efficient and declarative querying system for biological sequences. We present the design of the Periscope/SQ system that extends SQL to support complex sequence querying operations.
- To optimize complex sequence queries, fast and accurate estimation methods are critical. We make a contribution in this area by presenting a technique for estimating the selectivity of string/sequence pattern matching predicates based on a new structure called the Symmetric Markovian Summary. We show that this new summary structure is less expensive and more accurate than existing methods.
- We introduce novel query processing operators and also present an optimization framework that yields query plans that are significantly faster than simple approaches (which are usually coded by existing procedural querying methods).
- We present a case study of an actual application in eye genetics that is currently using our system, and demonstrate through a simple performance study the advantages of the Periscope/SQ approach.

The remainder of this chapter is structured as follows: Section 4.2 discusses our extensions to SQL. Our query processing technique includes novel string/sequence predicate estimation methods, which are presented in Section 4.3, and query optimization and eval-

uation methods, which are presented in Section 4.4. Section 4.5 contains the results of our experimental evaluation, including an actual application in eye genetics. Section 4.6 describes related work, and Section 4.7 contains our conclusions.

4.2 Extending a Relational DBMS

Biologists often pose queries that involve complex sequence similarity conditions as well as regular relational operations (select, project, join, etc.). Consequently, rather than build a stand-alone tool only for complex querying on sequences, the best way to achieve this goal is to extend an existing object-relational DBMS [138] to include support for the complex sequence processing. For the Periscope project, we have chosen to extend the free open-source object-relational DBMS (ORDBMS) Postgres [149]. Periscope/SQ, and also comment on the new types that are needed for this extension.

4.2.1 Algebra and Query Language

Our query language, which extends the SQL query language, is called called PiQL (pronounced as "pickle"). PiQL incorporates the new data types and algebraic operations that are described in our query algebra PiQA [145] [1].

The purpose of this section is to describe *very briefly* the PiQL extension to SQL and the related algebraic constructs. Readers who are interested in the details of the algebraic properties of these extensions may refer to [145].

Hit and Match Types

Hit: A hit is basically a triple *(p,l,s)*. When specified together with some sequence, the hit *(p,l,s)* means that there is a *hit* at position p of length l with a score of s on the given sequence. For instance, suppose that A = (2,3,3) is a hit on the sequence *SEQ* = "TG-

[1] PiQL stands for Protein Query Language – the full versions of both PiQA and PiQL can be used to query sequences and protein geometrical structures. Since DNA datasets don't have geometrical structures, querying on DNA only requires the subsets of these these methods that allows for querying on biological sequences.

Example PiQL Queries

1.CREATE TABLE prot-matches (pid INT,
p STRING, match MATCH_TYPE)
2. SELECT * FROM MATCH(R,p,"EEK",EXACT,3)
3. SELECT AUGMENT(M1.match, M2.match, 0, 10) FROM
MATCH(prots.p,"VLLSTTSA", MM(PAM30)) M1,
MATCH(prots.p,"REVWAYLL", MM(PAM30)) M2
4. SELECT CONTAINS(AUGMENT(
M1.match, M2.match, 0,10),M3.match) AS resmatch
FROM
MATCH(prots.p,"VLLSTTSA", MM(PAM60)) M1,
MATCH(prots.p,"REVWAYLL", MM(PAM60) M2,
MATCH(prots.s,"LLLLL", EXACT) M3
WHERE score(resmatch) $\geq$ 15

Figure 4.1: Example PiQL Statements

GTTTAGGAGGTA". This hit refers to the "GGT" substring, which could have matched some query for a score of 3. This hit can be shown in the original database sequence as "T**GGT**TTAGGAGGTA", with the hit portion highlighted in bold-face.

Match: A match is simply a set of hits. For example, consider the sequence *SEQ* = "TGGTTTAGGAGGTA", and a query to find "GGT" followed by a "GGA" within 10 symbols. A match for this query using an exact matching paradigm is X= {(2,3,3), (8,3,2)}. This match describes two hits in the data sequences as shown in bold-face in "T**GGT**TTA**GGA**GGTA".

Several operations are defined on the Match type:

- Start(match) is the lowest p value of all the hits in the match.
- End(match) is the highest $p+l$ value of all the hits.
- Length(match) is End(match)-Start(match).
- Flatten(match, f) is the match {(Start(match), Length(match), f(match)}, where f is a score-combination function.

Operations for match type are implemented as user-defined functions on this new data type. Query 1 in Figure 4.1 shows how to create attributes of this type using PiQL.

Match Operator

The Match operator finds approximate matches for a query string. It is implemented as a table function which takes as input a string, an attribute name, a match model (described later), and a cutoff score. The operator returns a relation with the match attribute. As an example of this operator, consider Query 2 shown in Figure 4.1 that finds all instances of the string "EEK" in attribute p of relation R (Table 4.1). The result of the PiQL query returns the relation *R* with an additional match column as shown in Table 4.2. The matching portions are shown in boldface in Table 4.1. These are referred to by position, length, and score in the match column of Table 4.2.

Since local-similarity search is a crucial operation in querying biological sequences, one needs to pay close attention to the match-model. In practice, the commonly used match models include the exact match model, the *k*-mismatch model, and the substitution matrix based models with different gap penalties. An exact match model simply requires that we find exact matches for the query with any substring in the database. A *k*-mismatch model allows for at most *k* differences (mismatches) between the query and any database substring. Finally, the general substitution matrix based models use a substitution matrix that specifies the precise score to be awarded when one symbol in the query is matched with a different symbol in the database. In this model, both insertions and deletions are permitted. A more detailed discussion of various matching models is beyond the scope of this manuscript, and we refer the interested reader to an excellent treatise on this subject [46]. The algorithms that Periscope/SQ uses for these different match models are discussed in Section 4.4.1.

While Periscope/SQ supports the three match models listed above, we focus on the exact match model and the *k*-mismatch model. The substitution matrix model is primarily used for protein sequences, and is not applicable for querying DNA or RNA sequences.

id	p	s
1	GQISDSI**EEK**RGHH	HLLLLLLLLLHEE
2	**EEK**KGF**EEK**RAVW	LLEEEEEHHHHHL
3	QDGGS**EEK**STK**EEK**	HHHHLLLEEEELLL

Table 4.1: Relation R

id	p	s	match
1	GQI...	HLL...	{(8,3,3)}
2	EEK...	LLE...	{(1,3,3),(7,3,3)}
3	QDG...	HHH...	{(6,3,3),(12,3,3)}

Table 4.2: Match Results

The exact and *k*-mismatch models however are often used with both protein and DNA sequences. When we discuss the techniques for query evaluation with the exact and *k*-mismatch models, we will make brief remarks on the extension for arbitrary substitution matrix based model.

Nest and Unnest

These operations can be implemented as table functions that take as input arguments the relation and the list of attributes to nest/unnest, returning the nested/unnested relation. For example, an expression like *Unnest(prot-matches, match)* will unnest the *match* attribute in relation *prot-matches*. Similarly, an expression such as *Nest(prot-matches, pid)* will nest the relation *prot-matches* with the *pid* as the simple key attribute [145].

Match Augmentation Operator

This operator operates on two relations (say *R1* and *R2* - both having a match attribute), and produces a new relation that contains all the non-match attributes, a new match attribute, and a key/id attribute. The match attribute is the union of the match of the left relation and a match on the right relation if the one from the right relation has the same (specified) id-field, and is within a specified distance range after the match of the relation on the left. If the match field in an operand contains several hits, then the operator

computes *flatten(m)* and uses that value for computation. As is obvious, the augmentation operator needs to be given the list of attributes in the two tables that need to be equal before it can compute a tuple in the result relation. As an example, consider Query 3 in Figure 4.1, which will find all matches that are the form "VLLSTTSA" followed by "REVWAYLL" with a gap of 0-10 symbols between them. Each component is found using a match operator, and combined using the augmentation operator.

Contains, Not-Contains

The contains operator selects those matches from its left operand that *contain* some match from the right operand. A match $A(p_1, l_1, s_1)$ is contained in $B(p_2, l_2, s_2)$ if $p_2 \geq p_1$ and $p_2 + l_2 \leq p_1 + l_1$. The syntax is similar to the Match Augmentation operator. The complex query described next (Query 4 in Figure 4.1) demonstrates a use of the contains operator. See [145] for more details.

Complex Query Example: As an example of a complex PiQL query consider the following query:

> *Find all proteins that match the string "VLLSTTSSA" followed by a match of the string "REVWAYLL" such that a hit to the second pattern is within 10 symbols of a hit to the first pattern. The secondary structure of the fragment should contain a loop of length 5. Only report those matches that score over 15 points.*

The PiQL query for this example is shown as Query 4 in Figure 4.1. The three MATCH clauses correspond to the match operators that would be needed to search for each of the specified patterns. The inner AUGMENT function in the SELECT clause finds the patterns that have "VLLSTTSA" followed by the "REVWAYLL". The CONTAINS call makes sure that only those matches that contain a loop of length 5 get selected.

4.3 Estimation, Operators, and Optimization for Query Processing

The introduction of sequence/string matching predicates poses an important problem while trying to optimize PiQL queries. Since an optimizer relies on fast and accurate selectivity estimation methods, poor estimation methods can lead to inefficient query plans (see Section 4.4). We address this issue by first presenting a new technique for estimating the selectivity of exact match predicates that is more accurate than previous methods. Then, we describe extensions of this technique for the k-mismatch and the substitution matrix models.

Our estimation method uses a novel structure called the Symmetric Markovian Summary (SMS). SMS produces more accurate estimates than the two currently known summary structures, namely: Markov tables [2], and pruned suffix trees [79, 89]. A Markov table stores the frequencies of the most common q-grams. (A q-gram is simply a string of length q that occurs in the database.) Pruned suffix trees are derived from count suffix trees. A count suffix tree is a suffix tree [103] where each node contains a count of the number of occurrences of the substring from the root that terminates at that node. To find the number of occurrences of the pattern "computer" using a count suffix tree, we simply traverse the edges of the tree until we locate the node that is at the end of a path labeled "computer', and return the corresponding count value. The pruned count suffix tree uses a pruning rule to store only a small portion of the entire count suffix tree [79]. A simple rule is to store just the top few levels of the tree, or store only those nodes that have a count above a certain value. Observe that a pruned count suffix tree in effect stores the frequencies of the *most commonly occurring patterns* in the database.

Notice that in these previously proposed strategies, the summary structures are biased towards recording the patterns that occur frequently. The estimation algorithms then typ-

ically assume a default frequency for patterns that are not found in the summary. For instance, this could be the threshold frequency used in pruning a count suffix tree. If a query is composed mostly of frequently occurring patterns, then this bias towards recording the frequent patterns is not an issue. However, if the query tends to have a higher selectivity (i.e., matches very few tuples,) such a summary can bias the estimation algorithm towards greatly overestimating the result size. As the experimental evaluation in Section 4.3.3 shows, these existing algorithms perform very poorly when it comes to negative queries (where 0 tuples are selected) and queries that are highly selective.

The key strength of SMS is that it captures *both* the frequent and rare patterns. Our estimation algorithm that uses SMS not only produces more accurate estimates for the highly selective predicates (the "weak spot" of previous methods,) but also produces better estimates for predicates with lower selectivities. In the following section, we now describe our estimation algorithm, and the SMS structure.

4.3.1 Estimation Method

Preliminaries

In a traditional database context, the selectivity of a string predicate is the number of rows in which the query string occurs. Alternately, we can define it as the number of occurrences of the query string in the database. Multiple occurrences in each row make these two metrics different. This alternate definition is more useful in bioinformatics where we are interested in finding all occurrences of a query string. This is the definition of selectivity we use in the rest of the chapter. Our technique can also be adapted to return the number of rows, and thereby be used in a traditional database setting for text predicates. This involves calculating q-gram frequencies differently, and in the interest of space we omit this discussion.

Most string datasets (English text or DNA or protein sequences) can be modeled quite

accurately as a sequence emitted by a Markov source. That is, we assume that the source generates the text by emitting each symbol with a probability that depends on the previous symbols emitted. If this dependence is limited to k previous symbols, then we call this a Markovian source with memory k, or simply a k^{th} order Markov source. In [79], the authors show that for most real world data sets, this k is a fairly small number. We refer to this property as the "short-memory" property, to mean that most real world sequences do not have significant long range correlations.

The Estimation Algorithm

Now, suppose that we have a query $q = a_1a_2a_3...a_n$. The number of occurrences of the string q in the database is the probability of finding an occurrence of q times the size of the database. Equivalently, this is *(the probability that the Markov source emits* q*)* $\times$ *(the size of the database)*. If $P(q)$ denotes the probability of the source emitting q, then:

$$\begin{aligned} P(q) &= P(a_1) \times P(a_2/a_1) \times P(a_3/a_1a_2) \times \\ &\quad ... \times P(a_n/a_1...a_{n-1}) \\ &= P(a_1) \times \Pi_{i=2}^{n} P(a_i/a_1...a_{i-1}) \end{aligned}$$

We can exploit the short-memory assumption and use the fact that $P(a/b_1..b_n)$ is the same as $P(a/b_{n-k+1}..b_n)$, where k is the memory of the Markovian source. The expression can now be rewritten as $P(q) = P(a_1) \times \Pi_{i=2}^{n} P(a_i/a_{i-k..i-1})$. If we had a table where we could look up values for $P(a_i/a_{i-k..i-1})$, this probability can be computed easily. The Symmetric Markovian Summary (SMS) provides these values.

The crux of the estimation algorithm is in making the best use of these values, and using reasonable approximations when these values are not found in the summary.

Algorithm StrEst

Estimation Function StrEst(*q, summary*)
1. p = 1.0
2. For i= 1 to $|q|$
   3. $s = q_1 ... q_{i-1}$
   4. If Prob(q_i/s) is stored in the summary, $v = Prob(q_i/s)$
   5. Else, $v = Prob(q_i/s')$,
      where s' is the longest suffix of s such that Prob(q_i/s') is in the summary
   6. $p = p \times v$
7. End For
8. Return $p \times DBsize$

Figure 4.2: Estimation Function StrEst

This algorithm, as shown in Figure 4.2, computes the estimates using the equation described above. While retrieving a probability from the summary, it first looks for $P(a/Y)$. If this value is not found, it searches the summary for $P(a/Z)$, where $Y = bZ$ for some symbol b. It successively searches for shorter suffixes of Y, and if nothing else is found, it returns $P(a)$. This algorithm may make as many as $k|q|$ probes of the summary. The basic intuition behind this approach is that we *expect* $P(a/bZ)$ can be approximated by $P(a/Z)$.

Other Match Models

For the *k*-mismatch model, we use a simple estimation technique. For small values of *k*, we list all possible strings that have at most *k* mismatches with the query string. We compute their selectivity using the exact match model, and add them up. For larger values of *k*, we use a different approach. We compute a *representative selectivity* s_r for the set of strings (W) that have at most *k* differences with the query string. The number of such strings is: $|W| = \sum_{i=1}^{k} C(L,i) \times (A-1)^i$. L is the length of the string and A is the alphabet size. (For an *i*-mismatch string, you choose *i* symbols from the L and replace them with one of $A-1$ symbols for a mismatch.) We then compute the selectivity as $s_r \times |W|$. An

obvious choice for s_r is the exact match selectivity of the query string. A better choice is the average selectivity of the set of strings with l mismatches, where l is a small number like 1 or 2. Such an average will effectively sample a larger subset of W and produce a better estimate (as also supported by the experimental results presented in Section 4.3.4).

For predicates using the general substitution matrix model, a simple estimation method is to use a heuristic that computes the selectivity of an *equivalent k*-mismatch predicate by choosing an appropriate *k*. The value of *k* is determined by examining the substitution matrix, the length of the query (L), and the threshold similarity score (T) of the predicate. We compute the average score for identity (A_i), and the average score for substitution(A_s). Frequent substitutions have a positive score, and rare ones often have a negative score. A near identical match would have a score of approximately $L \times A_i$. Since the required threshold is T, the slack we have is $L \times A_i - T$. This can be uniformly divided over the mismatches - so we compute *k* = $\frac{L \times A_i}{2 \times |A_s|}$. This is a simple and straightforward way of exploiting the matrix. However, this method makes it difficult to account for insertions and deletions. We are currently evaluating the performance of this technique.

Another alternative is to examine the properties of the substitution matrix to expand the query string into a set of closely homologous strings and to use existing estimation methods for each string. For instance, one could construct a set of homologous strings that included insertions and deletions, and then use the method previously described on each string and combine the results.

4.3.2 The Symmetric Markovian Summary

The Symmetric Markovian Summary (SMS) is essentially a lookup table that stores various probabilities of the form $P(a/Y)$, where a is a symbol in A (the alphabet,) and Y is a string of length at most k. If we let D_k denote the set of all probabilities where Y is exactly of length k, then $|D_k| = |A|^{k+1}$. In the simplest case when $k = 0$, this reduces to

storing the unconditional probability for each symbol in the alphabet. Ideally, one would like to have the summary $S = \cup_{i=0}^{k} D_k$ for some sufficiently large k.

The size of such a table grows exponentially with the value of k, making it impractical especially for large alphabets. Therefore, we need to choose a smaller subset of S such that these probabilities provide an accurate estimate. The basic idea behind SMS is to choose only the *most important* probabilities from S. A probability value is less important if we would incur only a small error if we didn't store it and approximated it with a different probability instead (when using algorithm StrEst).

We present two algorithms H1 and H2 that use different notions of the importance of a probability to construct an SMS. These two methods differ in the manner in which they compute the importance of an entry. Before describing these algorithms in detail, we first present the intuition behind defining a good notion of importance.

There are two components to the importance of a probability. A straightforward indicator of importance is the error that might be incurred if the value were *not* in the summary. We call this the δ-value of the probability entry. Suppose that we exclude $P(a/Y)$ from the SMS, and use some $P(a/X)$ (where X is the maximal suffix of Y,) from the summary to approximate it. We compute $\delta = |P(a/Y) - P(a/X)|$. Note that $P(a/Y)$ being more likely than $P(a/X)$ is just as important as it being less likely. It is this symmetric property that leads to a better summary.

An orthogonal but important factor that determines the importance of a probability entry is the likelihood that it will actually be used in some queries. This is basically a workload dependent factor. For instance, even if the probability $P(A/CACAC)$ has a higher δ value than $P(A/AC)$, it might still make better sense to choose $P(A/AC)$ to retain in the summary, simply because it is likely to be used more often than the former. For the workload as a whole, the average error incurred from approximating $P(A/AC)$

Algorithm H1(*String*,*k*,*B*)
OCC = [], STR = [], PROB=[] A = Alphabet U {*null*}
1. Calculate the frequency of each q-gram s for
   q varying from 1 to k as OCC(s).
//Now calculate conditional probability
2. For every a,Y such that |Y| < k
   3. PROB(a/Y) = OCC(Ya)/OCC(Y)
4. End For
5. Create Priority Queue PQ of Size B bytes
6. Fix unconditional probabilities into PQ.
7. For each entry in Prob
   8. priority = $|A|^{-|Y|+1}$ × |Prob(a/Y) - Prob(a/X)|
      where X is the longest suffix of Y present in PQ.
   9. PQ.insert(<a/Y, Prob(a/Y), priority>)
   10. If Size of PQ exceeds B, drop lowest priority element
      and adjust the priorities of affected elements.
11. End For
12. PQ contains the Symmetric Markovian Summary

Figure 4.3: Algorithm H1 to construct SMS

will add up to more than the error from approximating $P(A/CACAC)$ since $P(A/AC)$ is likely to be used more often. The likelihood that a given probability entry will be used for a given workload is the γ-value of the entry. In the absence of any characterization of the queries, one can assume a uniform query distribution and assign a higher γ to shorter strings. We combine these two components to define importance as the product of δ and γ.

Formally speaking, for a given k, and a fixed summary size (B entries), we want to store a subset of values from each of $D_0, D_1, ...D_k$ such that the values we prune away can be approximated well. Mathematically, we want to choose $T \subset U_{i=0}^{k} D_i$ such that $imp = \Sigma_{p \in T}(\gamma \times |p - Approx_T(p)|)$ is maximized. Here $Approx_T(p)$ is the value that will be used to approximate p in T, if p is excluded from T. We want a subset such that the total importance of each of the elements is the maximum over any subset of this size. In other words, we pick the subset that has the most important B elements. This is clearly a hard optimization problem. Constructing an optimal summary with a naive approach will

take an unacceptably long time. We therefore present two heuristic approaches H1 and H2 that perform very well for a wide range of datasets.

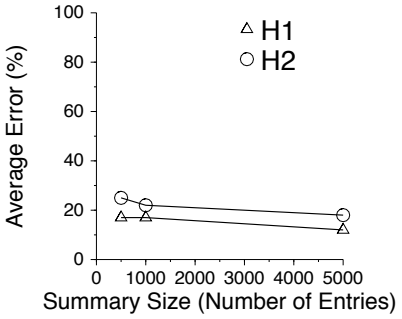

Figure 4.4: Low Selectivity Queries, MGEN: H1 vs. H2

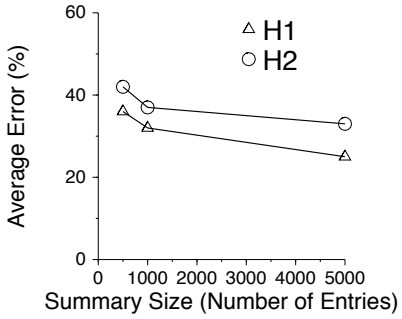

Figure 4.5: Medium Selectivity Queries, MGEN: H1 vs. H2

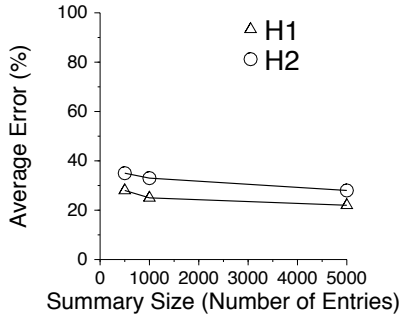

Figure 4.6: High Selectivity Queries, MGEN: H1 vs. H2

Algorithm H1

Algorithm H1 first computes D_0,D_1,...,D_k using a q-gram frequency table. Note that values from D_0 are the unconditional probabilities of occurrence of each of the symbols. We'll always need these for the first symbol of the query string. The algorithm first selects D_0 into the summary structure (maintained as a priority queue). For each of the entries in $D_i(i > 0)$, the algorithm computes $\delta = |P(a/Y) - P(a/X)|$. To find X, the maximal suffix of Y, it scans the priority queue. It then computes $\gamma = |A|^{-|Y|+1}$, and importance = $\delta \times \gamma$ and inserts the entry into the priority queue. If the queue size exceeds the maximum size of the summary, we remove the element with the lowest importance. We then scan the queue and adjust the δ value for those elements that were directly dependent on the entry we just deleted. This heuristic runs in time $O(nBlog(B))$, where B is the summary size, and n is the total number of probability entries being considered.

Algorithm H2

Though H1 is a good heuristic, an important drawback is that it is computationally expensive. H2 uses a simpler algorithm that runs faster than H1, but may yield a slightly less

accurate summary. Instead of scanning the priority queue to find the X that is the maximal suffix, H2 simply uses the unconditional probability instead of the actual $Approx_T(p)$ entry. Everything else remains the same. Note that we don't have to adjust any values now when we delete an element from the priority queue. The main advantage of this algorithm is that it is very simple, and fast. The running time for H2 is $O(nlog(B))$. Experimental evaluations show that H2 is not much worse than H1, but is significantly faster to compute.

Both H1 and H2 store the summary as a list of pairs ("a/Y", $P(a/Y)$) sorted on the first part. A lookup can be performed in $O(log(B))$ time using binary search.

4.3.3 Experimental Evaluation

In this section, we first compare the SMS-based algorithms H1 and H2. We also compare the SMS method with the method of [79], which is currently considered to be the best method for estimating the selectivities of exact match predicates. (Note that the recent work by Surajit *et al*. [26] uses an estimation method that is built upon existing summary structures such as the pruned suffix tree. Their technique uses a learning model to exploit the properties of English text, and is not applicable to biological data. We note that our contribution is orthogonal to [26] as their system can be built on top of SMS.)

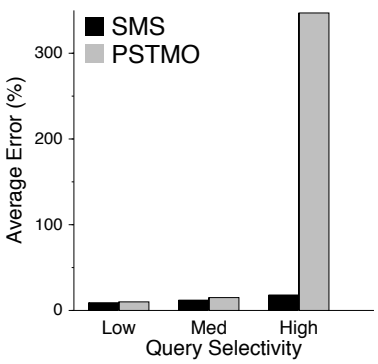

Figure 4.7: MGEN: SMS vs. PSTMO

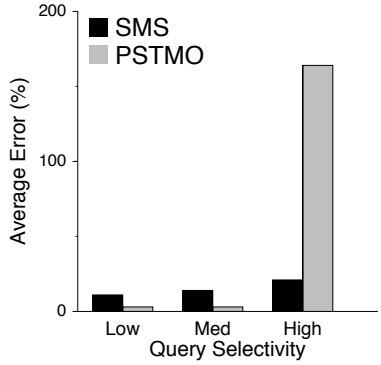

Figure 4.8: SPROT: SMS vs. PSTMO

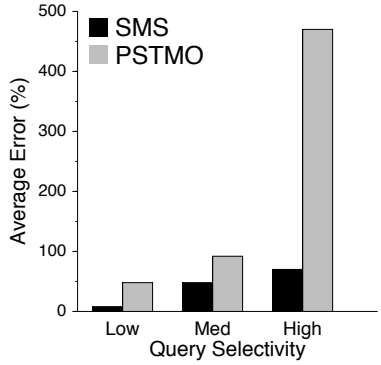

Figure 4.9: GUTEN: SMS vs. PSTMO

Experimental Setup

Data sets: We tested our estimation methods on number of different biological datasets: a nucleotide (DNA) dataset [57] (Chromosome 1 of Mouse, 200 million symbols) and a protein dataset (the SwissProt [10, 158] collection, 53 million symbols). We refer to these datasets as MGEN and SPROT respectively. To demonstrate the applicability of our methods for conventional databases, we tried our methods on a number of English text sources, including DBLP [44], a number of sources from the LDC Corpus [147], and the Gutenberg text repository [119]. The results using these text sources was very similar, and we only present the results using data from the Gutenberg project [119]. We refer to this dataset as GUTEN.

Query Sets: For MGEN, we generated 150 random strings ranging from lengths 6 to 12 so it would span all the selectivities. Similarly, for SPROT, we generated a set of 150 random strings of lengths ranging from 3 to 7. For GUTEN, we randomly picked 150 words of varying lengths from the database itself.

Result Organization: For each algorithm, we classify the queries based on their actual selectivities. Queries that have less than 1% selectivity are classified as high selectivity queries. The ones between 1%-10% were classified medium selectivity, and those that had more than 10% selectivity were classified as low selectivity queries. The metric of accuracy we use is the average absolute relative error calculated as a percentage : $e = 100 \times \frac{|prediction - actual|}{actual}$. We refer to it simply as the *average error*.

Note that since highly selective queries produce only a few results, the error in estimating this class can potentially present a skewed picture. For instance, if the actual number of occurrences was just 1, and we predicted 2 , that's a 100% error! A well established convention to not bias the result presentation for such cases, is to use a correction [26, 79]. While calculating the error, if the $actual$ selectivity is less than $100/|R|$, we divide the

absolute error in selectivity by $100/|R|$ instead of the actual value. $|R|$ is the number of tuples in the relation.

Platform: All experiments in this chapter were carried out on an 2.8 GHz Intel Pentium 4 machine with 2GB of main memory, and running Linux, kernel version 2.4.20.

Comparison of H1 and H2

In our first study, we examine the effect of using an SMS of type H1 versus one of type H2.

We ran the query sets using H1 and H2 on each of the datasets for varying summary sizes. We present the results for low, medium, and high selectivity queries with MGEN in Figures 4.4, 4.5 and 4.6. The results for other datasets are similar and are omitted here. From these figures, we see that as the summary size increases, both H1 and H2 have increased accuracy. However, H1 has a consistent advantage over H2. At larger summary sizes the error from H2 is within 10% of H1.

Note that the cost of using H1 is significantly higher than the cost of H2. For instance, with the MGEN dataset and an SMS with 1000 entries, the time taken to construct H1 is 219 seconds, while H2 takes only 93 seconds. However, H2 incurs only a small loss in accuracy. Therefore, we conclude that *except* for cases where very high accuracy is needed, or if the summary size is very small, we use H1 to construct the summary. In all other cases, we use H2 as it is cheaper to construct, and nearly as accurate as H1.

Comparison with Existing Methods

In this section, we compare our SMS based algorithm with the algorithm proposed in [79]. For this experiment, we used algorithm H2 to construct the summaries. The algorithm in [79] uses a maximum overlap parsing along with a Markovian model for the text. The summary structure they use is a pruned count suffix tree. For ease in presentation,

we refer to the method in [79] as the PSTMO algorithm.

For this experiment, we fixed the summary size to be 5% of the database size (results with 1% and 10% summary sizes are similar, and suppressed in the interest of space). We present the average absolute relative error for each class of query for each dataset in Figures 4.7, 4.8, and 4.9.

For the MGEN dataset (Figure 4.7), SMS has a slight advantage over PSTMO for low and medium selectivity queries. However, for high selectivity queries, PSTMO has a very large error - over 340%, compared to only 18% with SMS! In the case of SPROT (Figure 4.8), we see that PSTMO has a slight advantage for low and medium selectivity queries. This is mostly due to the fact that the query set has many short strings. PSTMO stores the exact counts of these short strings and therefore ends up being very accurate for these queries. However, for longer strings (high selectivity), the error for PSTMO rises sharply to 164%. In contrast, SMS has a low error of 21%. For GUTEN (Figure 4.9), SMS is better in all three cases, and the advantage is very large (70% versus 470%) in the case of highly selective queries. As discussed before in Section 4.3.2 SMS produces more accurate estimates because it is a symmetric digest of the information in the text.

The queries considered in the above study does not consider an important type of query – namely a *negative query*. While searching text databases, users commonly make spelling or typographical errors which result in the string predicate selecting *zero* records. Algorithms like PSTMO tend to provide very poor estimates for these queries. However, our SMS based algorithm works very well for these queries too. We have also experimented with negative queries, and the results are similar to the highly selective queries such as in Figure 4.6.

Execution times: In addition to producing accurate estimates, it is also desirable to have estimation methods that can compute the estimation very fast. We examined the estimation

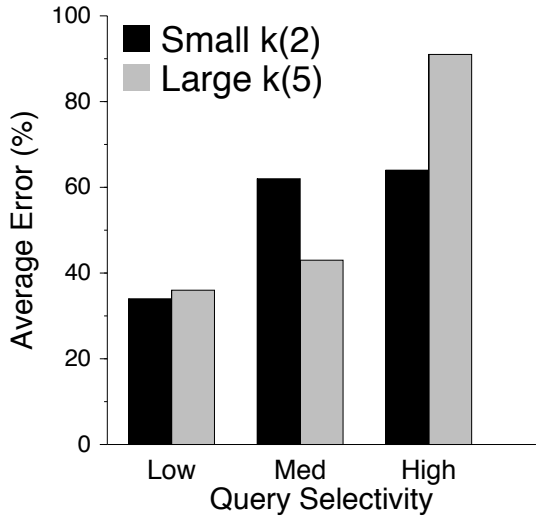

Figure 4.10: K-Mismatch Estimation Error

computation time for each method, and show the average per-query estimation times in Table 4.3. As can be seen from this table, our approach is cheaper than PSTMO. This is because PSTMO needs to repeatedly traverse a suffix tree. Traversing suffix tree nodes is expensive as it involves chasing a number of pointers. *It is noteworthy that the SMS based estimation is both faster and more accurate than PSTMO!*

4.3.4 K-Mismatch Estimation

We examined the efficacy of our approach for estimating predicates using the *k*-mismatch model for different values of *k*. We present the results of the study for the case of a small *k* (2) and a large *k* (5) in Figure 4.10. Observe that the error in estimation in this case is generally higher than the exact model. This is because we use the estimates from the exact model to compute these estimates, and the cumulative error tends to be significantly larger. Inspite of the relatively larger error, the estimates are reasonably accurate for queries of all selectivities.

Summary

In summary, we have presented an algorithm for estimating the selectivity of string/sequence predicates using a novel structure called the Symmetric Markovian Summary (SMS). Our estimation method using SMS is more accurate than existing algorithms, and also takes less time for computing the estimate. Existing methods are particularly poor in estimating the selectivity of highly selective predicates, which is gracefully handled by our approach. As our empirical evaluation shows, in some cases our approach is up to 5 times more accurate than the previous best algorithm.

4.4 Query Evaluation

The introduction of new operators in PiQL presents two significant challenges. First, we need efficient algorithms to execute new operators like match, augment, contains, etc. Second, we need to extend the optimizer to be able to optimize over the new operators. We first discuss algorithms for the crucial match operator. We then briefly describe algorithms for other operators and present a new physical operator called the Match-and-Augment. Finally, we present an optimization algorithm that is highly effective at finding good plans for a subset of queries.

4.4.1 Algorithms for Match

The algorithms for evaluating the match operator varies depending on the match model. In the simplest case - the exact match - a linear scan of the database can be used. The *Scan* algorithm scans the sequence from start to finish and compares each sequence with the query pattern for an exact match. With a match model such as a k-mismatch model, a Finite State Automaton (*FSA*) is constructed for the query, and each sequence is run through this automaton. The cost of this algorithm is $O(n \times q_{eq})$ where n is the length of the database, and q_{eq} is the expected number of states of the automaton that are traversed before deciding

Data Type	SMS	PSTMO
MGEN	3.1	66.1
SPROT	7.2	17.8

Table 4.3: Estimation Time (in microseconds)

on a hit or a miss. For the more complex model using a substitution matrix, the linear scan or the FSA scan algorithm cannot be used directly. For this complex match model, we can use the Smith-Waterman [136] (*SW*) algorithm, which is a dynamic programming local-alignment algorithm. Its time complexity is $O(m \times n)$ where m is the size of the query and n is the size of the database. The BLAST [5, 6] family of algorithms is a heuristic approach to local-similarity searching that runs faster than SW, and finds *most* matches for a given query.

The OASIS [105] algorithm is a suffix tree based technique for sequence similarity that can be used with any match model (including the substitution-based matrix model with affine gap penalties). In the case of the exact match, one can simply traverse down the suffix tree along the query string and collect all the leaf nodes under that node (this is essentially a simple suffix tree query). The cost of this algorithm is $O(q + r)$ where q is the length of the query and r is the number of matches. The cost of a k-mismatch search with a suffix tree is typically similar to an OASIS search.

Choosing the right algorithm can not only impact the performance greatly, but sometimes even the accuracy. If BLAST is used, then there is a possibility that some of the hits might be missed - it should be used only in cases when this is acceptable. Smith-Waterman and OASIS on the other hand never miss matches and could always be used in all situations, though these algorithms can be more expensive to execute.

Algorithms for other operators like augment, contains, not-contains are similar to a traditional join. Instead of a simple equality, the join condition tends to be a complex predicate involving match types. A nested loop style algorithm is used to evaluate the

match-augmentation and the contains operator.

4.4.2 A New Combined Operator

We have designed a new physical operator that combines matching with the match augmentation operator. We call this the Match-and-Augment (MA) operator. It can be used to extend a set of matches with another set of matches on the same dataset. For instance, consider the following expression:

AUGMENT(MATCH(A.seq,"ATTA",MM(BLOSUM62)),

MATCH(A.seq."CA",EXACT), 0,50).

A simple way to compute this expression is to evaluate each match independently, and then use a join to compute the augment. Alternately, we can evaluate the first MATCH, then scan 50 symbols to the right of each match that is found, and check for the occurrences of "CA". In this process, we select and augment only those matches where we find the "CA". This is essentially the approach used in the MA operator. The MA approach can often be cheaper than performing two matches separately and combining the results with the augment operation.

4.4.3 Optimization

Our current optimization strategy uses a two stage optimization method. In the first step, we optimize the portion of the query that refers to the complex sequence predicates, and in the second stage we call the Postgres optimizer to optimize the traditional relational components of the query. We acknowledge that this two step process may miss opportunities for optimization across the two components. Our eventual aim is to integrate these two steps, but we start with this two step optimization as it is more amenable for rapid prototyping. In this section, we describe the methods that we have developed for optimizing the complex sequence predicates.

The basic idea behind the optimization algorithm is as follows: Suppose that the query contains n match predicates connected together by operators like augments. We compute the selectivity of each match predicate, and pick the most selective predicate to start with. We examine the predicate *adjacent* to this and compute the cost of evaluating that match and combining it with the current predicate. Now, we compare this with the cost of using a match and augment operator. If it is cheaper, then we rewrite the plan to use a match and augment operation and examine another adjacent predicate in the same way. The algorithm terminates when an adjacent predicate cannot be combined using a match and augment or when all the predicates have been combined. The algorithm is outlined in Figure 4.11.

It is clear that the algorithm runs in time proportional to the number of match predicates. Although it explores a very small portion of the plan space, it is highly effective at finding good plans. We demonstrate this in Section 4.5 using extensive experimental evaluation.

The optimizer uses SMS for predicate selectivity estimation. The cost models are fairly straightforward and considers CPU cost and I/O cost. The cost models follow the complexity of the algorithms with empirically determined constants plugged in. The following section briefly describes the cost models.

4.4.4 Cost Models

In real database systems, the cost models for various operations are often finely tuned and returned over the lifetime of the system. The cost models presented here are simple initial estimates.

The match operator can be evaluated using many algorithms. The linear scan for the exact match will incur N reads, where N is the number of pages the database sequence occupies (every page is read once). The CPU cost for this is $(c_1 \times l_{exp} \times D) + (c_2 \times Q)$, where l_{exp} is the expected number of comparisons needed to determine if a match has occurred or not for the given string. Q is the number of results - every time a match is

obtained, it is copied into a buffer, and that incurs a cost. D is the length of the database sequence. So, the total cost for the scan operator is: $(c_1 \times l_{exp} \times D) + (c_2 \times Q) + (c_3 \times N)$, where c_3 is the cost of a disk I/O. The FSA scan operator has the same cost, except that l_{exp} is computed differently, and c_1 has a larger value.

When a suffix tree is used to compute exact matches, we first traverse down the suffix tree until we find the node at the end of the query path, and collect all leaves below that node. The first part requires computational time proportional to the length of the query. The computational cost of the second part is proportional to the size of the subtree below the node. The number of I/O's incurred depends on the size of the buffer pool, and the buffer replacement policy. To simplify the analysis, we assume that the top few levels of the suffix tree are kept in memory. So the first part does not incur any I/O (for short queries). The second part incurs at least as much I/O as the number of pages that the leaf nodes occupy. This is approximately $Q \times f$ where f is the number of nodes per page. Therefore the cost for this operation is approximately $(c_1 \times |S|) + (c_2 \times Q \times f)$, where $|S|$ is the length of the query string and Q is the number of matches. The first part tends to be very small, so we use $c_2 \times f \times Q$ as the cost estimate. c_2 accounts for the I/O cost and also includes a correction factor to account for the non-leaf nodes.

The OASIS and BLAST algorithms are more complex. The OASIS algorithm has a worst case cost, W, which is equal to $min(c_1 \times |S|^{|A|}, l)$, where $|S|$ is the length of the query, $|A|$ is the size of the alphabet, c_1 is a constant, and l is the number of symbols in the database. The constant c_1 is roughly the time it takes to compare an entry in a cell of the Smith-Waterman matrix [105]. The average cost of an OASIS operation is often smaller than this. Assuming that the top few levels of the suffix tree are cached in memory, the algorithm incurs roughly $k \times Q$ page reads where Q is the number of results, and k is an empirical constant. (This I/O estimate is very crude, but represents a good starting point.

In reality the I/O complexity depends on the parameters of the search, such as the E-value, the characteristics of the substitution matrix, and the affine gap penalty model.) The total cost is therefore $W + (c_2 \times k \times Q)$.

The BLAST algorithm has a computational cost of $(c_1 \times D) + (c_2 \times c_3 \times Q)$. D and Q are as described above. c_1 is the cost of a hash lookup, and c_2 is the cost of expanding a word hit, which we set to a constant (actually, this depends on the method used like the 1-hit or the 2-hit extension and the scoring model.) Finally, c_3 is the number of word hits produced by the word matching component of BLAST, which we set to a fixed constant. The I/O cost for the first phase (finding word hits) in BLAST is modeled as a search of the entire database sequentially - this is N reads. The word extension phase reads c_3 random blocks out of these N. This leads to approximately $N[1 - \Pi_{i=1}^{k}(D - B - i + 1)/(D - i + 1)]$ page accesses, where B is the number of symbols per page. This formula is an approximation [168] to Yao's formula [175] used for estimating page accesses.

The match augmentation and the contains operators are join-based algorithms. We use a nested loops style join for these operators, and estimate these costs using traditional join cost models [132].

The match-and-augment operator's cost is similar to the cost of the FSA scan. Suppose the left operand is a set of A_1 matches, and distance to which we need to search is L symbols, then a total of $A_1 \times L$ symbols need to be compared. The computational cost is $(c_1 \times l_{exp} \times A_1 \times L) + (c_2 \times Q)$. If f is the number of symbols per page, the I/O cost incurred is roughly $A_1 \times \lceil L \times f \rceil$ page accesses.

4.5 Experimental Validation

In this section, we present the results of various experimental studies that we conducted to examine the performance of our system. Using several synthetically generated query

Algorithm Optimize
1. Compute selectivity s(i) of each predicate
2. Compute cost c(i) of evaluating each predicate
3. Let f be the most selective predicate
4. Let g be an adjacent predicate
5. t = cost of evaluating g, then combining it with f.
6. u = cost of using a match-and-augment operator
7. If t $>$ u , then rewrite the plan as match-and-augment
8. If there is another adjacent predicate that has
   not been considered, pick it to be g. Go to step 5.
9. End

Figure 4.11: The Optimization Algorithm

loads, we explore a wide range of query situations. In addition, we also present results that are based on a real-life workload that was captured while a scientist was performing explorative querying using our tools. We used the full mouse genome [57] (2.6 billion symbols) as the dataset for the experiments in this section. We also performed experiments on several other genetic datasets and protein datasets, which show similar trends.

4.5.1 Impact of SMS-based Estimation

In order to understand the benefits of increased accuracy from the new SMS based estimation algorithm, we performed the following experiment. We randomly generated a hundred queries having three match predicates each. One of the predicates used a *k*-mismatch model, while the others used an exact match. The query load was executed for $k = 0$, 1, and 2. (We use these relatively small values since k is usually a small number in practice. Our methods also work for larger values of k.

The lengths of each of these predicates was randomly chosen to be between 6 and 14. Neither the suffix tree index, nor the match and augment operator is used in evaluating these queries. Each query was optimized by *exhaustively* searching over the plan space. (Note that in this experiment we are *not* using the linear optimization algorithm of Section 4.4.3, but rather, a simple exhaustive enumeration of *all* the query plans. This

exhaustive optimization is guaranteed to pick the plan with the best estimated cost, thereby isolating any effects related to the optimization algorithm.)

We optimized the queries in two ways: In one case we used PSTMO [79] to estimate the selectivities while optimizing the query, and in another case, we used the SMS based estimation algorithm. We used a one percent summary in both cases. We found that the average running time of the query plan (which does not include the optimization time) was higher by about 43% when using PSTMO. Of the 100 queries, 90 queries were optimized identically by both algorithms, and 10 queries were optimized differently. These 10 query plans took roughly 4.6 times as long to execute when optimized using PSTMO as opposed to using SMS. The reason for this behavior is because PSTMO had overestimated the selectivity of some of the predicates by a margin large enough that it led to a different execution plan in each of these ten queries.

4.5.2 Impact of Using Match and Augment

In this experiment, we explore the effectiveness of using the new match and augment operator (MA), which was described in Section 4.4.2. For this experiment, we ran the set of 100 queries generated as above in two different ways. One plan was optimized with the match and augment operator and the other plan without it. For this experiment also, we used an exhaustive search optimization algorithm. The query plan evaluation times are summarized in Table 4.4 for each value of k. As is evident, the use of the new operator can lead to significant savings. The plan that used the match-and-augment operator executed 10 to 80 times faster on average!

In Table 4.4, we also provide the standard deviation of the times for the 100 queries. To get a better understanding of how often and how much the match and augment operator helps, we split the queries into three sets: the first set, where the new operator provides at most a 2X speedup (small advantage), the second bin where the speedup was greater than

k	**Without MA** Average (*Std-Dev*)	**With MA** Average (*Std-Dev*)
0	3.04 (*11.5*)	0.19 (*0.08*)
1	46.71 (*142.08*)	0.55 (*0.65*)
2	226.76 (*808.5*)	13.55 (*41.46*)

Table 4.4: Query Plan Evaluation Times (in minutes)

2 but less than 10 (significant advantage), and the third bin where the speedup exceeded a factor of 10 (large advantage). We observed that for $k = 0$, in 65% of the queries were in the first category, around 20% in the second, and 15% in the third. Similarly for the case where $k = 1$, the split-up was 35%, 30%, and 35% respectively. Finally for $k = 2$, the query set split was 30%, 20%, 50% into the three categories. It is clear from the evidence that the new operator can be very useful in a significant number of queries.

4.5.3 Optimizer Evaluation

In this experiment, we compare two optimization algorithms. The first one is a conventional algorithm that *exhaustively* searches the plan space for the best plan. The second algorithm is the linear time optimization algorithm described in Figure 4.11. For this experiment, a suffix tree index is available on the data, which increases the number of algorithms that the optimizer can choose from. We generated three sets of hundred queries each with 3, 5, and 7 predicates. One of the predicates in each query was randomly selected to use a k-mismatch model with k randomly chosen as one of 0, 1, 2. The average query optimization time and the evaluation time in each case is shown in Figure 4.12. The plan obtained using the linear time optimization algorithm always runs within 6% of the optimal plan's running time. For the exhaustive query optimization method, the time take to optimize the query is low for a small number of predicates (3 or 5), but is unacceptably large when more predicates (7 and above) are used. Performing an exhaustive search to find the optimal plan is a better option only in the case of 3 predicates. Overall, what

this experiment shows is that the linear query optimization method is quite robust. The exhaustive optimization method can produce slightly better plans, but should only be used when the query has a small number of predicates.

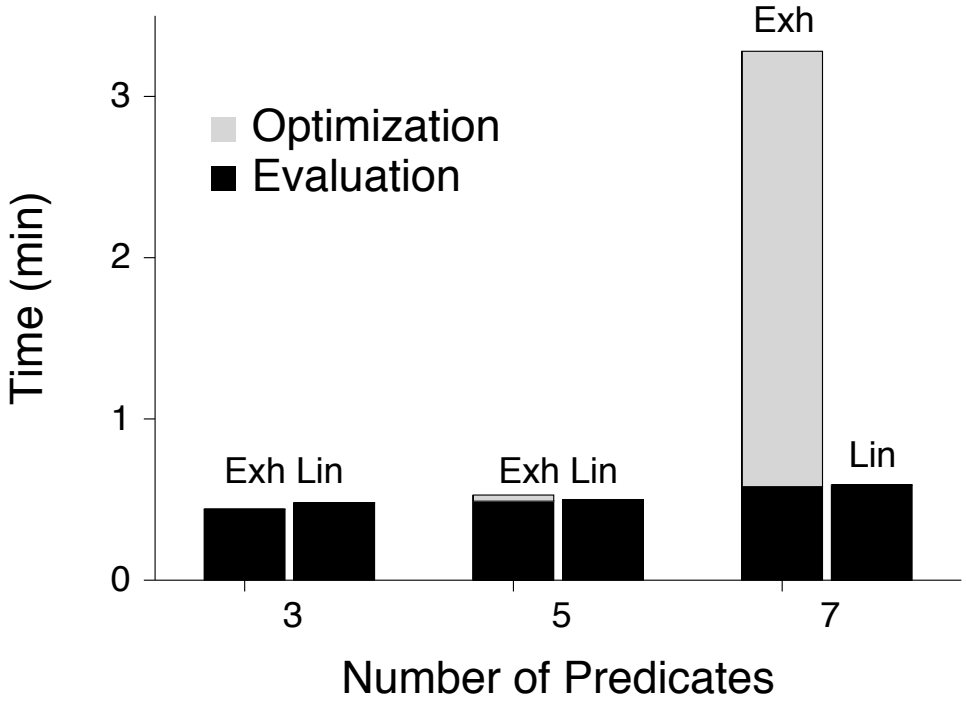

Figure 4.12: Optimization and Evaluation Times

4.5.4 GeneLocator: An Application

The current prototype implementation of Periscope/SQ has been used in an web-based application called GeneLocator that we have built in collaboration with researchers at the Kellogg Eye Institute at the University of Michigan. GeneLocator is a tool for finding target *promoter regions*. In order to understand certain genetic factors associated with eye diseases, our collaborators are trying to identify all genes that are regulated by a particular transcription factor (a regulatory protein, also called a promoter). Such proteins typically bind to a "signature" binding site: a short sequence of DNA about 10-15 bases

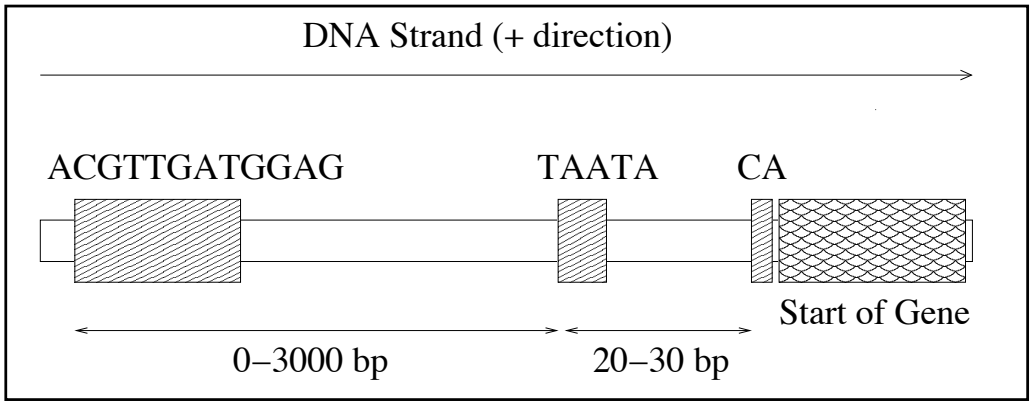

Figure 4.13: Promoter Binding Region

long. The pattern usually allows for a few mismatches. The presence of a TATA-box (a pattern such as "TAATA") or a GC-box (a pattern like "GCGC") within a certain distance downstream of the match to the signature often indicates that it is a potential binding site. Also, transcription almost always begins at a "CA" site, which is a short distance following the TATA-box or the GC-box. Figure 4.13 pictorially represents the kind of pattern our collaborators are looking for. In PiQL, this query can be expressed as:

SELECT AUGMENT(AUGMENT(
M1.match, M2.match, 0,2988),
M3.match, 15,35) AS res, G.name *FROM*
MATCH(DB.dna,"ACGTTGATGGAG",KM(1)) M1,
MATCH(DB.dna,"TAATA",EX) M2,
MATCH(DB.dna,"CA",EX) M3,
GeneAnnotations as G, *WHERE* score(res) $>$ 15 AND
G.start $>$ start(res) AND G.start - start(res) $\leq$ 5000 AND
G.chromosome = DB.chromosome

The extra conditions in the Where clause filter out the matches to report only those that are a short distance upstream of a known gene. In the above query, GeneAnnotations is a table with the following schema: *GeneAnnotations (id, chromosome, start, end, type, annotation)*, and is loaded with the gene annotation data from NCBI [57].

GeneLocator is accessed by a web interface, which allows the end user to pose queries by filling out a simple form. Our collaborators are working with the mouse genome, and use this tool for posing interactive queries. With their permission we logged the queries

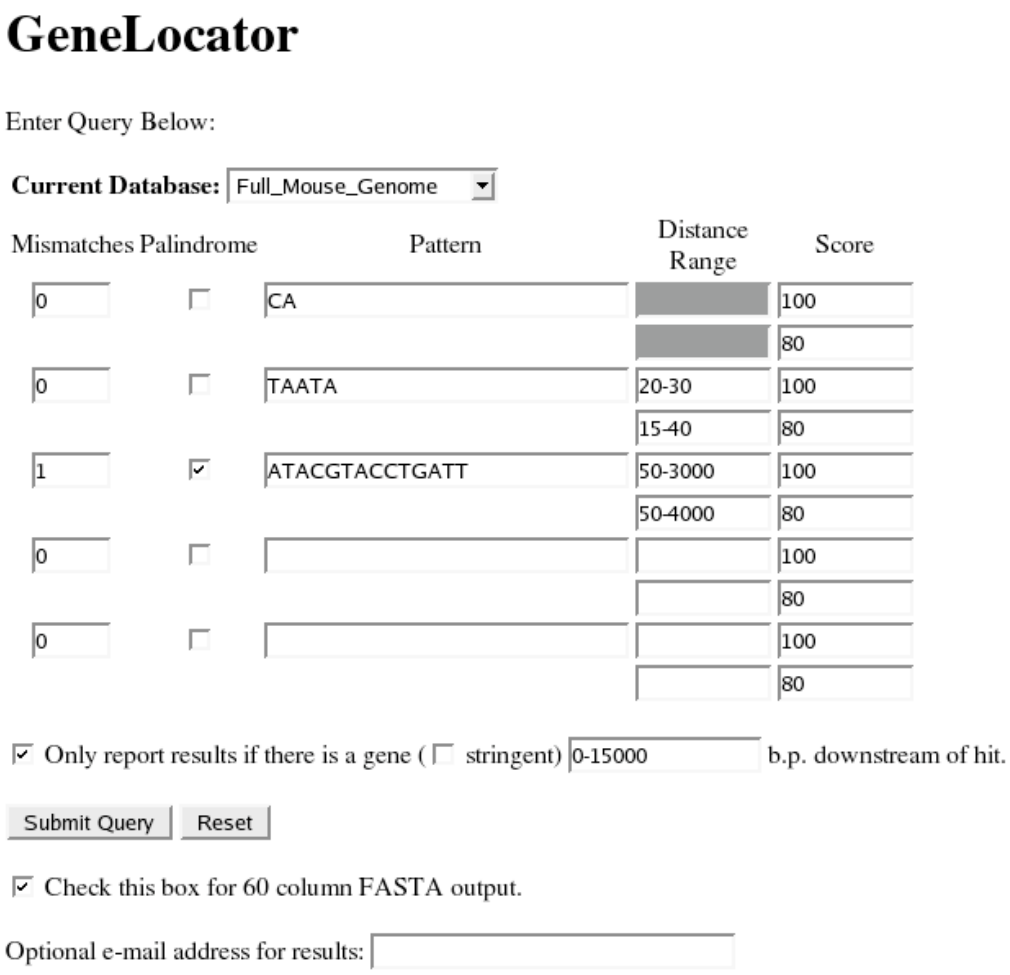

Figure 4.14: Screenshot of the GeneLocator Interface

that they issued. Most of their queries had three match predicates. The inter-predicate distance and the number of mismatches allowed in the match model varied across the queries. One or two of the predicates often used an exact match model. The others used a *k*-mismatch model. The actual queries are not presented in order to protect the privacy of the research. For this application we built a suffix tree on the mouse genome using our suffix tree construction method [143]. A screenshot of the GeneLocator interface is shown in Figure 4.14. The search results are displayed as in Figure 4.15.

Algorithm	Time (min)
Unoptimized Plan (No Index)	473.05
Optimized, No MA (With Index)	9.76
Optimized, With MA	1.02

Table 4.5: Execution Times

4.5.5 Performance of GeneLocator

We compared the execution times of the set of queries logged using three different query plans. The first query plan does not use any indexes, and uses no optimization - a naive left to right evaluation of the augments is used to compute the result. The second plan uses a suffix tree and an exhaustive search to choose the cheapest plan. It does not use the match-and-augment operator. The third plan is optimized using the linear optimization method and includes the match-and-augment operator. The dataset used was entire the mouse genome (2.6 billion symbols). The execution times are as are shown in Table 4.5.

The first observation we can make from Table 4.5 is that using the suffix tree can dramatically improve the query execution time. This does not come as a surprise, since suffix tree index based algorithms are usually very efficient. Second, we observe that the plan with the match and augment operator executes faster than the version without it by nearly an order of magnitude. The current procedural methods that are used in life sciences research labs tend to resemble the first plan (no indexes, no optimization, simple operators) and therefore take an extremely long time to run. *The contribution of Periscope is not only that it provides a declarative and easy way to pose complex queries, but also that it executes them up to 450 times faster than existing procedural approaches!*

4.5.6 Results

Using GeneLocator, the eye genetics researchers were able to identify several potential targets for the transcription factor of interest, which are now being verified using wet-lab experiments. These targets were computationally identified using our system after just a

few days of explorative querying. This process could easily have taken several weeks or months to accomplish using conventional methods. Encouraged by these results, we are now planning more ambitious queries in comparative genomics.

4.6 Related Work

Miranker *et al.* suggest an approach for querying biological sequences in [106]. They borrow some constructs from our previous algebraic proposal PiQA [145], to describe complex queries, and largely focus on designing and exploiting metric space indexing structures for querying sequences. Our work does not require a similarity measure to be a metric and focuses on providing a declarative way of posing complex queries while being able to evaluate them efficiently.

A closely related previous effort is the work by Hammer and Schneider [68], which outlines an approach to expressing complex biological phenomenon through algebraic operations. Their approach aims to build a completely new algebra that is very powerful in expressing *all* biological operations such as transcription, translation, crossover, mutations, etc. However, our approach more carefully charts out the operations for querying sequences and aims at extending relational algebra so that we can take advantage of all the existing relational infrastructure.

In [121], the authors propose an alignment calculus on strings to query string databases. They also describe a system that was built based on this algebra [62]. The language lets a user express very complex queries, by permitting complex string processing predicates to be written using alignment calculus declarations. However, the notion of an approximate match is hard to capture in this context. Also, to our knowledge, no performance evaluations have been carried out for this system.

Previous work in querying sequences by Seshadri, Livny, and Ramakrishnan [110,134],

describe techniques for storing and declaratively querying sequences. However, this work is tailored towards handling time series style data where windowing, projecting, aggregating over subsequences are important. In our work, we are interested in operations on biological sequences which are quite different as it involves approximate pattern matching queries with complex match models.

Recognizing the need for supporting sequence query matching in a relational framework, commercial DBMS vendors have recently started supporting BLAST calls from SQL statements [48,137]. However, these methods only provided limited sequence searching capabilities, allowing only simple pattern search (for example match-augmentation is not supported), and can only work with the BLAST match model.

Krishnan, Vitter, and Iyer presented one of the earliest approaches for estimating the selectivity of exact wildcard string predicates in [89]. The more recent work by Jagadish *et al*. [79] improves on [89] by using a short-memory Markovian assumption instead of an independence assumption. These methods employ pruned suffix trees as the summary of the text in the database. Suffix trees are versatile data structures, however, they have the drawback of being biased towards storing more frequent patterns. The SMS based approach we propose does not have this bias and is more accurate than existing techniques.

Chaudhuri, Ganti, and Gravano [26] recently proposed a technique which takes advantage of the frequency distribution properties of the English text to increase the accuracy of estimation techniques. The method is based on the fact that English text often has a short identifying substring. This has not been shown to be applicable to other datasets such as DNA and protein sequences. The estimation methods that we propose here can easily fit into the overall framework of [26] for use in text databases.

4.7 Conclusions and Future Work

In this chapter, we have presented Periscope/SQ - a DBMS for declarative querying on biological sequences. We presented PiQL, a language that extends SQL to permit complex queries on biological sequences, and have also described a novel and effective sequence predicate estimation method. In addition, we have presented techniques for efficiently optimizing and evaluating queries using these complex sequence predicates. We also described a real world application built using Periscope/SQ, which clearly demonstrates the huge impact that this approach can have for scientists querying biological sequences.

Search Results

Gross Hits: 1780.

Query Details

Pattern	Mismatches	Palindrome	Distance
ATACGTACCTG	1	1	0-0
TATA	0	0	25-40
CA	0	0	50-300

Gene Dist: 0-15000

>gil38083781|reflNT_039340.2|Mm6_39380_32 Mus musculus chromosome 6 genomic contig, strain C57BL/6J at 2019136 **Match** #1 **Score** 300
GTCCATGCATGACATGGACGTGTATATGTGTATCCATGTGTCCATGTGTGTGCGTGTGTG
TGTGTGTGTGTGTGTGTGTGTGTGTGCATGTGTGTGGGTCTATGTGTTGTGGGTGTGTGG
ATATATATACTCATATATAATACAGAAATCCTATGAGGACA
Potential features nearby:
Type: GENE Name: LOC381751

>gil38083781|reflNT_039340.2|Mm6_39380_32 Mus musculus chromosome 6 genomic contig, strain C57BL/6J at 3471908 **Match** #2 **Score** 300
GTACATGCATATATATAATCATAAAAGAAAGCAAAGAAATGTATAACATACTATTCAGGG
TTTTGGTTACCTCTGAAAGGCAAGAGCTGATGGAACTGGAAAGAATCCACAGCAAAGAAA
AAAGAAAAGAAAAGAAAAGAAAAGAAAAGAAAAGAAAAGAAAAGAAAAGAAAAGATTCGG
AGGCAACTATGTTCTATTGCTTAACAGGAAGGTGGGTTTATGACTCTTCATTTTATATTT
CTCCTGTATGGTGTTATAATACTTTCCTATATGTTTCAAGAAGTCA
Potential features nearby:
Type: GENE Name: LOC381752

Figure 4.15: Screenshot of the Search Results

CHAPTER V

Mining for Patterns

5.1 Introduction

In a number of emerging sequential data mining applications, the goal is to discover frequently occurring patterns. To illustrate the characteristics of such an operation, consider Figure 5.1. This figure shows the percentage change in the stock price for IBM over the previous minute's average price, for several minutes in a day. An interesting data mining question on this sequence dataset is: "Are there any frequently recurring patterns in this time series dataset?" Finding patterns in stock price data can provide valuable insights to stock traders about short-term market fluctuations. In fact, technical analysts in financial markets often try to discover price patterns through visual inspection. These patterns are often given descriptive names such as "Head and Shoulders" [22] and "Adam and Eve" [22], and are used in designing short-term trading strategies.

In the example shown in Figure 5.1, the bold segments highlight a pattern that occurs four times in the dataset. Note that the recurring subsequences are similar, but not identical. The challenge in discovering such frequent patterns is to allow for some *noise* in the matching process. At the heart of such a method is the definition of a pattern, and the definition of similarity between two patterns. This definition of similarity can vary from one application to another. A simple approach in the case of stock price data such as in

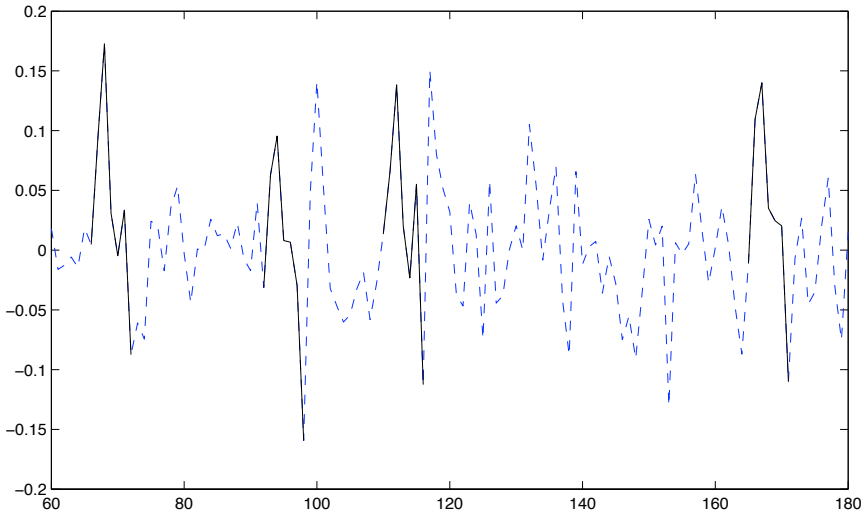

Figure 5.1: IBM Stock Data: The bold segments represent a frequently occurring approximate pattern.

Figure 5.1 is to define a tolerance value, ϵ, and consider two sequences to be similar if corresponding values in the sequences are within ϵ of each other.

The approximate subsequence mining problem also has a number of applications in many existing scientific database applications. A challenging problem in computational biology is to detect short sequences, usually of length 6–15, that occur frequently in a given set of DNA sequences. (A DNA sequence is a string over an alphabet of size four.) These short sequences can provide clues regarding the locations of so called "regulatory regions", which are important repeated patterns along the DNA sequence. The repeated occurrences of these short sequences are not always identical, and some copies of these sequences may differ from others in a few positions. The similarity metric that is often used here is the Hamming distance between the two sequences, or more simply, the number of positions in which they differ. These frequently occurring patterns are called *motifs* in the computational biology world. In the rest of this chapter, we use this term to describe *frequently occurring approximate sequences*.

Clearly, different applications require different similarity models to suit the kind of noise that they deal with. It is desirable for a motif mining algorithm to be able to deal with a variety of notions of similarity. In this chapter, we present a powerful new model for approximate motif mining that fits several applications with varying notions of approximate similarity, including the examples described above. We also present FLAME (**FL**exible and **A**ccurate **M**otif D**E**tector) – a novel motif mining algorithm which can efficiently find motifs that satisfy our model.

We draw the reader's attention to the fact that the problem of motif mining is related to the traditionally studied problems of mining for frequent itemsets [3], and frequent subsequences [4]. The problem of finding frequently occurring (non-contiguous) subsequences in large sequence databases has been extensively studied in previous works [4, 164, 173, 174, 181]. Traditionally, B is called a subsequence of A, if B can be constructed by projecting out some of the elements of sequence A. For instance, if A is the sequence "a,b,a,c,b,a,c", the sequence "a,b,b,c" is a subsequence constructed by choosing the $1^{st}, 2^{nd}, 5^{th}$, and 7^{th} elements from the original sequence and omitting the rest. While mining for frequent *non-contiguous* subsequences has many uses, it is not appropriate for many applications such as the DNA and stock price examples above. A subsequence constructed by gluing together distant parts of the original sequence is not meaningful in these applications. In mining for motifs, we are interested in *contiguous* subsequences. Furthermore, previous work on non-contiguous subsequence models cannot easily incorporate noise tolerance in the way that contiguous motif models can. In short, subsequence mining and motif mining are different data mining operations, and there are distinct applications of each of these. We focus on the contiguous subsequence (motif) mining problem.

Readers closely familiar with traditional (non-contiguous) subsequence mining algorithms may note that some of these methods can be adapted to mine for contiguous sub-

sequences [116, 180]. In this work, we compare our method with one such algorithm, namely cSPADE [180], and show that FLAME is faster by an order of magnitude.

The motif mining problem is also related to similarity searching for time series data analysis. A host of techniques have been developed to find sequences in a time series database that are similar to a given query sequence [27, 55, 104, 107, 161, 182]. The motif mining problem that we consider is a different data mining operation, where the user is looking for frequent motifs based on some skeleton of a pattern. Interestingly, a similar problem of finding motifs in traditional time series datasets has recently been identified in [32, 114]. While mining for motifs, these algorithms use models that are similar to the models used in the stock price example and the DNA example in favor of other measures. For instance, [32] employs a motif finding method called Random Projections (RP) [21]. We compare our method with RP, and show that for larger database sizes, FLAME often outperforms RP by more than an order of magnitude.

Motivated by the problem of finding frequent patterns in DNA sequences, which has profound importance in life sciences, the computational biology community has developed numerous algorithms for detecting frequent motifs using the Hamming distance notion of similarity. YMF [135], Weeder [115], MITRA [50], and Random Projections [21] are examples of algorithms in this category. Compared to this class of algorithms, we show that FLAME is more flexible, and can use more powerful match models. We also demonstrate through empirical evaluation that FLAME is more scalable than these existing methods and can be an order of magnitude faster for mining large databases.

There are several applications of motif mining in addition to those already mentioned. It is often the first step in discovering association rules in sequence data ("basic shapes" in [36] and "frequent patterns" in [72]). It can also be used to find good seeds for clustering sequence datasets [114]. Records of medical signals, like ECG or respiratory data [170]

from patients can also be mined to find signals that can indicate a potentially critical condition.

We make the following contributions here:

1. We present a powerful new model that is very general and applicable in many emerging applications. We demonstrate the power and flexibility of this model by applying it to datasets from several real applications.

2. We describe a novel motif mining algorithm called FLAME (**FL**exible and **A**ccurate **M**otif D**E**tector) that uses a concurrent traversal of two suffix trees to efficiently explore the space of all motifs.

3. We present a comparison of FLAME with several existing algorithms (YMF [135], cSPADE [180], Weeder [115], and Random Projections [21, 32]). FLAME never misses any matches (as opposed to some of these methods that apply heuristics). In fact, we show that FLAME is able to identify many true biological motifs that existing algorithms miss.

4. We show that our algorithm is scalable, accurate, and often faster than existing methods by more than an order of magnitude!

The remainder of the chapter is organized as follows: Section 5.2 presents related work, and Section 5.3 describes our model for motifs. In Section 5.4, we present the FLAME algorithm. Section 5.5 contains our experimental results, and Section 5.6 contains the summary and conclusions.

5.2 Related Work

There is a vast amount of literature on mining databases for frequent patterns. Early work focused on mining association rules [3]. The problem of mining for subsequences

was introduced in [4]. Subsequence mining has several applications, and many algorithms like SPADE [181], BIDE [164], CloSpan [173] (and several others) have been proposed as improvements over [4].

More recently, interest in domains such as financial time series, medical time series, biological sequences like DNA and proteins, etc., has led to research in algorithms for finding frequent patterns in the presence of noise. Yang et al. [174] use a statistical sampling based method with a compatibility matrix to tolerate noise. However, they are primarily concerned with subsequence mining, while we focus on contiguous patterns.

Some algorithms have been proposed that incorporate constraints in subsequence mining. Constraints which limit the maximum gap between two items in the subsequence make it possible to use these algorithms to mine for contiguous patterns. Algorithms such as cSPADE [180], Pei et al. [116] can be adapted to mine for exact contiguous motifs. An obvious reason why these are unsuitable for approximate frequent pattern mining is that these algorithms do not include a notion of noise or an approximate match. Furthermore, they tend to be inefficient even when used for exact substring mining. FLAME, on the other hand is extremely efficient even for approximate substrings. (We demonstrate the performance advantage of FLAME in Section 5.5.)

Many algorithms have been proposed in the bioinformatics community for finding patterns in long noisy DNA sequences. These algorithms can be divided into two classes – pattern based and statistical. The patterns based algorithms typically search through the space of potential patterns and find a motif that satisfies the minimum support. Marsan and Sagot [102] proposed a suffix trie based algorithm to find structured motifs tolerating a few mismatches as noise. This method is primarily focused at finding pairs (or sets) of motifs that co-occur in the dataset within a short distance of each other. This method only considers a simple mismatch based definition of noise, and does not consider other more

complex motif models such as a substitution matrix or a compatibility matrix as in [174]. Furthermore, Marsan and Sagot do not have optimizations, such as the ones we describe in Section 5.4.1. These optimizations make FLAME faster by an order of magnitude.

Several other algorithms such as the Yeast Motif Finder [135] (YMF), Weeder [115], MITRA [50] have been used for finding motifs. YMF is a simple algorithm that scans the dataset using a sliding window and counts the number of occurrences of *every* possible motif of a given length. Once it has these counts, it computes the statistical significance of each motif, and outputs the best ones. YMF scales very poorly with increasing complexity of motifs, and thus cannot be easily adapted to other applications. Weeder is a suffix tree based algorithm that makes certain assumptions about the way the mismatches in an instance of the motif are distributed. This makes Weeder extremely fast, but it is not guaranteed to always find the motif. Weeder too, cannot be adapted for other motif models. MITRA is a mismatch tree based algorithm which uses clever heuristics to prune the large space of possible motifs. MITRA is very resource intensive and requires large amounts of memory.

Statistical approaches use techniques such as Expectation Maximization [15], Sampling [150], Random Projections [21], etc. to search for frequent patterns in the data. All of these heuristic approaches run the risk of finishing at a local optimum, and may not be able to find the right motif. Furthermore, these methods are specifically tailored for the problem of simple mismatch based motifs, and cannot easily be adapted for more complicated models.

A recent study by Tompa et al. [152] compared several different statistical and pattern based motif finding algorithms on a variety of real and synthetic datasets, and identified Weeder [115] and YMF [135] as the most effective methods. In our evaluations, we extensively compare with these two methods.

Surprisingly, there is little published work in finding motifs in time series databases. Time series data such as stock prices, economic indexes, time varying measurements from sensors and medical signals like Electrocardiograms can be mined for motifs, and all have compelling applications. Patel et al. [114] show that time series data can be discretized and converted into a sequence over a fixed alphabet and mined using existing motif mining algorithms. Another algorithm that finds frequent trends in time series data was proposed by Udechukwu, Barker, and Alhajj in [155]. However, these algorithms mine for *exact* frequent patterns, and are difficult to employ in the case of noisy datasets. Chiu et al. describe an algorithm in [32] (based on the Random Projections algorithm [21]) which accounts for noise in the data. However, this algorithm is also limited to a simple mismatch based noise model. In addition, this is a probabilistic algorithm, and is not always guaranteed to find all existing patterns. FLAME, on the other hand provides the options of a variety of models, and is guaranteed to find the motif (i.e. it is an accurate algorithm and not a heuristic method).

5.3 The Model

A critical aspect of the motif mining problem is defining the model under which two or more sequences are considered to match (approximately). Developing such models poses an interesting challenge: On the one hand, we want a model that is robust enough to detect the occurrence of a pattern even in the presence of noise, and on the other hand, we do not want it to be so general that it matches unrelated subsequences. Since different applications may have different criteria for how to strike this balance, a natural approach is to develop a flexible model with a few intuitive parameters that can be set by the user based on the application characteristics. In this section, we present a powerful new model for motifs that can be used for pattern mining in many different domains.

Throughout this section, we will assume that the input sequence is composed of symbols from a discrete alphabet set. However, our methods can also be applied to continuous time series datasets by converting such datasets into a *symbolic* sequence dataset by simply discretizing the numeric data. In fact such a transformation is frequently carried out for mining continuous time series datasets [32, 114].

We call our motif model the (L, M, s, k) model after the four parameters that determine it. L is the length of the motif, M is a distance matrix that is used to compute the similarity between two strings, s is the maximum distance threshold within which two strings are considered similar, and finally, k is the minimum support required for a pattern to qualify as a motif.

The (L, M, s, k) model is a very intuitive and powerful model, and permits the user a lot of flexibility in making the right tradeoff between specificity and noise tolerance of a model. As we describe below, much of this power comes from the ability to use any matrix M as the distance matrix. This property makes it useful for a variety of complex motif mining tasks. The matrix M allows us to define a *distance penalty* when a symbol X in the model matches a symbol Y in the data sequence. The penalty is specified by M(X,Y), an entry in the matrix. The total distance between the two strings is computed by summing the distance penalties of the corresponding symbols. That is, if $A = a_1a_2a_3...a_n$ and $B = b_1b_2b_3...b_n$ are two strings, then the distance between A and B under this model is $d(A, B) = \Sigma_{i=1}^{n} M(a_i, b_i)$.

Formally speaking, a string S is an (L, M, s, k) motif if there exist at least k strings $T_1, ..., T_k$ in the database such that each of them is of length L, and $d(S, T_i) \leq s$, where $d(A, B) = \Sigma_{i=1}^{n} M(a_i, b_i)$ is the distance function. Every string S that satisfies the above is an (L, M, s, k) motif. Note that the string S need not actually appear in the database for it to qualify as a motif. Only the instances T_i need to be in the database.

Sym	A	B	C	D	E	...	I	J	K
A	0	1	4	9	16	...	64	81	100
B	1	0	1	4	9	...	49	64	81
C	4	1	0	1	4	...	36	49	64
D	9	4	1	0	1	...	25	36	49
...	...	...	...	...	...	...	...	...	...
J	81	64	49	36	25	...	1	0	1
K	100	81	64	49	36	...	4	1	0

Table 5.1: An example distance matrix that implements the sum of squared differences measure

A domain which requires a matrix based measure of similarity is protein motif mining. Finding regions in protein sequences that appear frequently in different proteins is useful in inferring the functional sites in proteins. As in the case of DNA, the patterns in protein sequences do not repeat exactly. The instances of the pattern usually differ from the model in a few positions. To complicate things further, not all mismatches are equally bad. Some amino acids are very similar to each other, while some are very different. For instance Alanine and Valine are both hydrophobic amino acids, while Glycine and Serine are both hydrophilic. The matrix can be used to award a small penalty for M(X,Y) when X and Y are similar (Alanine and Valine, for instance) and a larger penalty otherwise (say, Alanine and Glycine) [70]. Popular substitution matrices such as PAM [38] and BLOSUM [70] can easily be used in our model.

Next, we demonstrate how this model can also be applied to the stock price example of Section 5.1. Suppose that we had normalized the data for firm ABC. Assume that the normalized stock price values are between 0-10. If we discretized them to integers, we could use letters A – K to represent 0 – 10. Suppose further that we wanted to find sequences of length 10 that appeared (approximately) in the database at least 20 times. If we wanted to use the sum of squared differences as the distance metric to check for similarity, we can simply use the matrix shown in Table 5.1. In this table, M(X,Y) is set to $(v(X) - v(Y))^2$ where v(X) is the numerical value corresponding to the symbol X. Using

this matrix, we can specify that an instance matches the model if the Euclidean distance between them is within a given threshold. We model this problem as a $(10, M, s, 20)$ motif finding problem, where s is an appropriately chosen similarity threshold.

The matrix can be adapted to allow other kinds of models. In fact, the matrix approach lets us simulate any L_p-norm (Manhattan distance, Euclidean distance, etc.). If we wanted to match two sequences only if the corresponding values (in the two sequences) were within 2 units of each other, (the ϵ-error tolerance model from Section 5.1), we would just set M(X,Y) = 0 where |v(X) - v(Y)| $\leq$ 2, and ∞ everywhere else. In general, any measure that can be computed in an incremental fashion by comparing the symbols in the corresponding positions can be simulated by constructing an appropriate distance penalty matrix.

We now discuss two special cases of the (L, M, s, k) model that are commonly used in computational biology and other domains - the (L, d, k) and (L, f, d, k) models.

5.3.1 Special Case: The (L, d, k) Model

The (L, d, k) model is a mismatch based model commonly used in computational biology for finding DNA motifs. The distance measure between two strings is the Hamming distance, or merely the number of mismatches. The (L, d, k) model is parameterized by the length of the string that we want to find (L), the maximum Hamming distance (d), and the support (k). The parameter d controls the amount of noise we wish to tolerate.

The (L, d, k) model is a special case of the (L, M, s, k) model. It can easily be simulated by a matrix by setting $M(X, Y) = 1$ if $X \neq Y$ and $M(X, Y) = 0$ if $X = Y$. This way, the distance function simply counts the number of mismatches. We set s to d and use the k from (L, d, k) as our minimum support.

One of the applications of this model is in the field of computational biology. The (L, d, k) model and its derivatives have been considered a good fit for DNA regulatory

motifs [152]. Briefly, the related problem of using this model to find regulatory motifs in DNA is as follows: Biologists today are interested in understanding how different genes in the genome are regulated and the way they interact with each other. To this end, biologists often study genes that exhibit similar expression patterns to extract clues about the proteins that control their expression. It is believed that genes that are co-regulated by the same protein (called a transcription factor) share some signal that allows the transcription factor to recognize the gene and turn it on. This signal is usually present in the region upstream of a gene (within a few thousand base pairs) called the promoter region. The signature is usually a short string of DNA 6-15 bases long. As is often the case in biology, these signatures are seldom identical, and differ in a few positions from one gene promoter region to another. Finding this noisy signature that is common across all the genes is a very important step towards locating the binding site for the transcription factor. Modeling the set of promoter regions as our database, and the signature binding site as an (L, d, k) pattern, we can simply apply the FLAME algorithm to solve this problem. We show in Section 5.5, that this is indeed an effective approach.

In most practical situations we don't know the exact value of L, and therefore, we might have to try several values. In the case of DNA regulatory patterns, we know that the signature is usually between 6 to 15 bases long, and therefore we can try these lengths with varying number of mismatches.

The (L, d, k) model can also be used in other applications where we wish to tolerate an occasional burst of noise. If two sequences were identical except for the addition of a noise spike in one of them, they will match under a 1-mismatch model. Consider the two sequences shown in Figure 5.2. The two bold segments are identical except for the single spike in the lower sequence. Such spikes may occur due to measurement error or other reasons, and an (L, d, k) model will be able to tolerate this noise and correctly match the

two sequences.

We show in Section 5.5 that the FLAME algorithm is faster than several existing algorithms that deal with finding only (L, d, k) motifs.

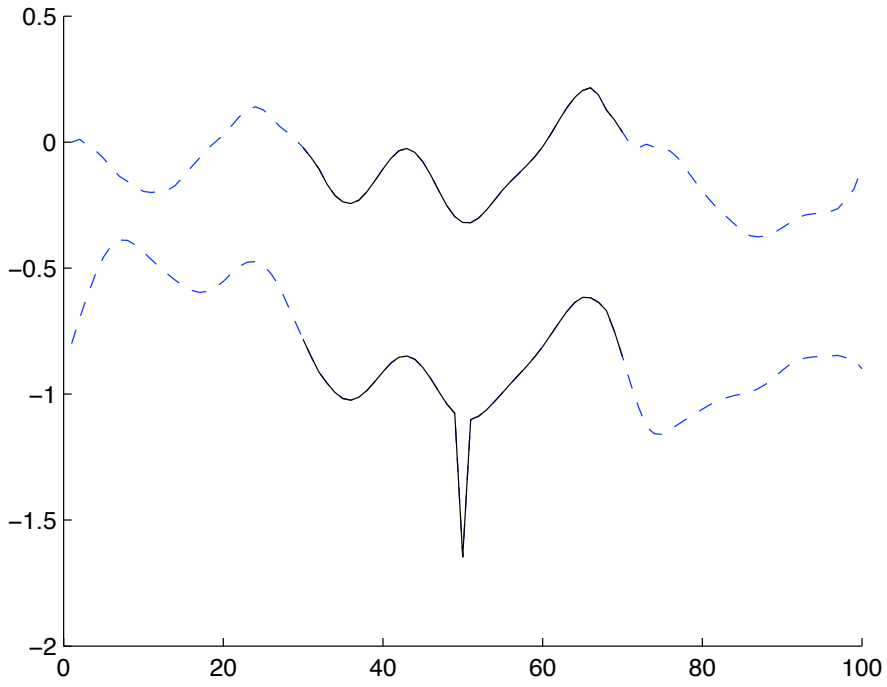

Figure 5.2: Potential uses of the LDK model - the lower segment is identical to the upper segment except for the single spike. The (L, d, k) model can match these.

5.3.2 Special Case: The (L, f, d, k) Model

The (L, f, d, k) builds on the (L, d, k) model to include some positional constraints on the mismatches. We introduce this model using an example: Consider the three sequences {`ABCD, ACCD, ABCA`}. If `ABCD` is the model sequence, the other two sequences are within 1 mismatch of the motif, so these sequences would constitute a $(4, 1, 3)$ motif in the (L, d, k) model. Now consider the sequences {`ABCD, ACCD, ADCD`}. This set also forms a $(4, 1, 3)$ motif, but the mismatches, whenever they occur, are always in position two (`AcCD`, `AdCD`). The (L, f, d, k) model allows us to specify the number of *fixed-position*

mismatches (f) along with just the number of free mismatches (d). This allows us to screen out patterns of the first kind, and focus on patterns of the latter kind. In other words, instead of allowing a mismatch anywhere in the substring, we look for all model strings whose instances always differ from it (if they differ at all) in the same positions.

The (L, f, d, k) model is also a special case of the (L, M, s, k) model. In order to model the fixed position mismatches, we simply augment the alphabet A with a wildcard symbol, say "?". For symbols in A, the distance matrix M is as in the (L, d, k) model, with $M(X, Y) = 1$ if $X \neq Y$ and zero everywhere else. The wildcard symbol is allowed to match any symbol with no penalty, so we set $M(?, X) = 0$ for all X. The space of model strings that FLAME considers is strings of length L over the augmented alphabet such that there are at most f occurrences of the wildcard symbol. This way, the (L, M, s, k) model can simulate the (L, f, d, k) model.

The (L, f, d, k) model can also be very useful when mining for regulatory elements in DNA since the mismatches tend to have a positional bias. In general, this model is useful in applications where the noise has a positional bias as it allows us to be more specific in finding the right patterns while ignoring extraneous matches. Some DNA motif finding applications [135] use models that are somewhat similar to the (L, f, d, k) model.

We illustrate the advantage of being able to use positionally biased scoring with an example. Consider a DNA dataset consisting of 5 sequences, each of length 500. Assume that each sequence has in it the motif `GTGAACAC`, and each instance of the motif has a mismatch at the fifth position. In other words, the dataset contains an $(8, 1, 0, 5)$ motif. Note than an $(8, 1, 0, 5)$ motif is also an $(8, 1, 5)$ motif in the (L, d, k) model since a free mismatch can capture a fixed mismatch. If we use the (L, d, k) model to retrieve the pattern, we will end up with many extraneous hits that might not be meaningful. When we search for an $(8, 1, 0, 5)$ pattern, FLAME (correctly) returns the result `GTGA?CAC`. On the

other hand, if we search for $(8, 1, 5)$, FLAME returns several other hits that satisfy $(8, 1, 5)$ but not $(8, 1, 0, 5)$. A post-processing step is needed to check if these are actually fixed position mismatch motifs. An (L, d, k) model can be used to simulate an (L, f, p, k) model if $f + p = d$ with some post processing. However, as we will explain in Section 5.4.1, using an (L, f, d, k) model produces a huge cost saving when compared to $(L, d + f, k)$ with post-processing.

5.4 The FLAME Algorithm

In this section, we describe the FLAME algorithm, which can be used to find (L, M, s, k) motifs. For ease of exposition, we explain the algorithm using an (L, d, k) model, and then describe how we extend it to the full-fledged (L, M, s, k) model.

Recall that an (L, d, k) motif is a string of length L that occurs k times in the dataset, with each occurrence being within a Hamming distance of d from the model string. Given, L, d, and k, a naive algorithm is to consider all possible strings of length L over the alphabet (the space of all models), and compute the support for each of them by scanning the dataset. This algorithm is exponential and becomes infeasible with large L and d values. One might be tempted to improve this method by considering only those strings of length L that actually occur in the dataset. However, this approach might miss motifs as the model string might not actually occur in the dataset even once. To illustrate this point, suppose that the string `ABCDEF` is the true motif. Assume that we are looking for a $(6, 2, 3)$ pattern, and that the instances of this pattern in the dataset are `FFCDEF`, `ABFFEF`, and `ABCDAA`. Each instance is at a distance of 2 from the model `ABCDEF`, but the distance between any two instances is 4. If we consider only instances from the dataset (which need not contain `ABCDEF`), then we will not find the motif.

The approach we take in FLAME explores the space of *all* possible models. In order

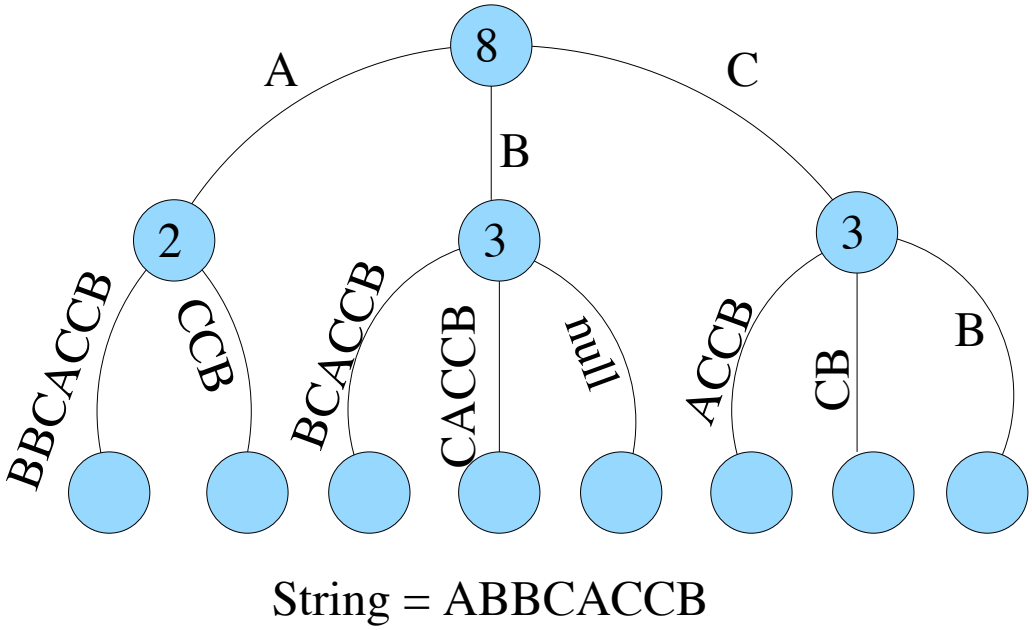

Figure 5.3: A count suffix tree on the string ABBCACCB. The counts are indicated inside the node.

to carry out this exploration in an efficient way, we first construct two suffix trees: a count suffix tree on the actual dataset (called the *data suffix tree*), and a suffix tree on the set of all possible model strings (called the *model suffix tree*). This second set is typically the set of all strings of length L over the alphabet. As we describe below, the model suffix tree helps guide the exploration of the model space in a way that avoids redundant work. The data suffix tree helps us quickly compute the support of a model string. Recall that a count suffix tree is merely a suffix tree in which every node contains the number of leaves in the subtree rooted at that node. In other words, every node contains the number of occurrences of the string corresponding to that node. (An example suffix tree is shown in Figure 5.3.) The basic intuition here is that the data suffix tree helps us combine the work common to finding the support for models like `ABCDE` and `ABCDF` (having a common prefix) and perform it only once.

Since the second suffix tree (built on all possible model strings) can be extremely large, FLAME does not actually construct this suffix tree. Rather, it algorithmically generates

portions of this tree as and when needed. FLAME then explores the model space by traversing this (conceptual) model suffix tree. Using the suffix tree on the dataset, FLAME computes support at various nodes in the model space and prunes away large portions of the model space that are guaranteed not to produce any results under the model. This careful pruning (described in more detail below), ensures that FLAME does not waste any time exploring models that do not have enough support. The FLAME algorithm simply stops when it has finished traversing the model suffix tree and outputs the model strings that had sufficient support.

To understand our strategy of pruning the model suffix tree, consider the following example: Assume that the dataset consists of sequences over the alphabet $\{\texttt{A,B,C,D,E}\}$. The dataset and the values of L, d, and k are specified as input. All the strings of length L starting with the symbol `A` form a subset of the model space. We call this the `A` partition. This partition corresponds to all the nodes in the model suffix tree under the subtree corresponding to node `A`. This partition is further divided into sub-partitions with prefix `AA`, `AB`, `AC`, `AD`, and `AE`. These partitions continue on for L levels, and at the last level, we have only one model string for each partition.

Suppose that we start by considering the models in partition `A`. Assuming no mismatches are allowed, if the support for `A` is less than k, then, clearly any model that starts with `A` cannot qualify as a valid motif since there will be fewer than k instances of it, and it will not have the minimum support. Consequently, we can safely toss away the entire space of models starting with the symbol `A`. This step essentially prunes away the subtree corresponding to `A` in the model suffix tree. After pruning `A`, we proceed to consider the `B` partition. An important step here is to compute the support for models starting with `A`. This value is simply the number of times `A` occurs in the dataset, and this value can be quickly looked up from the count suffix tree on the dataset.

Node	Number of mismatches	Count
A	0	100
B	1	50
C	1	45
D	1	120
E	1	15
Support	-	330

Table 5.2: The list of matches for the model `A`.

When mismatches are allowed, computing the support of a (partial) model string is more complicated. Suppose that $d = 1$. When considering matches for models starting with `A`, we cannot rule out strings that start with `B` (or any other symbol), since a string starting with `B` could match a model starting with `A` by only differing in the first position. Now assume that the data suffix tree nodes at depth 1 labeled `A, B, C, D`, and `E` have counts of 100, 50, 45, 120, and 15 respectively. The possible number of strings starting with `B` that could match a model starting with `A` is simply the count of node `B`, namely 50. In a similar fashion, the count value from other nodes at most d mismatches away is read, and a list of potential matches for `A` is constructed as shown in Table 5.2. The list contains the node in the data suffix tree, the number of mismatches corresponding to this node, and the count from that node. For instance, node `A` in the data suffix tree has a count of 100 and perfectly matches the model string (`A`) - we store this information in the list as (`A`, 0, 100). The total support for the partial model is now computed by summing up the individual counts. In the example for Table 5.2, this sum is 330. Those nodes where the number of mismatches with the model being considered is greater than d are pruned away and not included in the list of matches. The algorithm then proceeds to consider the next partial model – `AA`.

Observe that the list of matches for any partial model can be constructed incrementally using the list of matches for that model's longest prefix. For instance, the list of matches for `AC` can be constructed using the list for `A`(Table 5.2). We take each string from the

list, and extend it by one symbol. The first string `A`, for instance can be extended by one symbol to `AA, AB, ..., AE`. The string `AC` has 0 mismatches itself, the remaining strings have 1 mismatch each. The support for each of these string can be quickly looked up in the count suffix tree. We locate the model suffix tree node corresponding to `A` (stored in the list of matches). This node points to its children, namely `AA, AB, ..., AE`. The support for each of these models can simply be read from the suffix tree, and a new list of matches is constructed for `AC` to compute its support. Similarly, when `B` is extended to length 2, all strings except `BC` have more than one mismatch with the model string `AC`. Therefore only `BC` is included in the match list for `AC`. The remaining nodes (`C`, `D`, and `E`) are expanded similarly.

We take advantage of this method for incrementally computing the support by traversing the model suffix tree in the depth first order. If $L = 3$, the partitions will be considered in the order `A, AA, AAA, AAB, AAC`, etc. At each node, the match list and the support for the parent node has already been computed, and can be used to compute the support of the current node.

The pseudocode for FLAME is given in Figure 5.4. The algorithm simply puts together the ideas described above. FLAME uses a suffix tree on the model space and a count suffix tree on the dataset. It starts by traversing the nodes of the model space in depth first order. At each node in the model suffix tree, the subroutine `Evaluate_Support` is called to compute the list of matches and the new support. This routine uses the match list from the parent node to speed up the computation. The routine `Expand_Matches` ensures that the number of mismatches to the model string does not exceed d. At any node, if FLAME discovers that the support is lower than k, it prunes away that subtree in the model suffix tree, and continues its traversal. If it finds a model of length L with the required support, it simply outputs the result.

The algorithm described in Figure 5.4 works with (L, d, k) models. For the (L, M, s, k) model, the `Evaluate_Support` and `Expand_Matches` functions become more sophisticated. Instead of merely keeping track of the number of mismatches, they keep track of the substitution distance score. That is, for each node, the match list stores $\Sigma_{i=1}^{n} M(x_i, y_i)$ where x_i is the symbol from the prefix of the partition, and y_i is the symbol it is being matched to in the data set. If this distance score exceeds the preset threshold (s), we prune the model suffix tree at that point, and continue the depth first traversal just as in the case of the simpler (L, d, k) model. The new `Evaluate_Support` function is shown in Figure 5.4.

For the (L, f, d, k) model, we use the augmented alphabet to generate model strings that contain at most f wildcard characters and use the scoring matrix described in Section 5.3.

5.4.1 Optimizations

We now describe two opportunities for optimization when applying FLAME to practical problems.

Combining Computation

Very often in a real application, the exact length of the motif is not known a priori. Often, the user only has a rough idea of the range in which the length may lie. For instance, in regulatory DNA motif finding, scientists believe that motifs are typically 6 to 15 bases long. One often ends up trying several (L, d, k) values such as $(6-15, 1, 100\%)$, $(6-15, 2, 100\%)$, $(6-16, 1, 70\%)$, $(6-15, 2, 70\%)$, etc. Given the way in which FLAME computes the support for various candidate models, the algorithm can easily combine the computation for many different lengths if the number of mismatches is the same across all lengths.

Recall that the suffix tree of all models is traversed in a depth first fashion. We build the

suffix tree on all strings of length L_{max} – the longest length in the range we are examining. At any node, if the length of the model happens to be in the range of lengths considered, and the support is greater than the minimum support, we output that model, and *continue* the traversal. When we were considering only one length at a time, a valid model would only be found at a leaf node of the suffix tree since it consisted of strings only of length L. We allow lengths in the range of L_{min} to L_{max} by returning valid models starting at depth L_{min}.

This optimization can be applied to (L, M, s, k) models in general. The speedup obtained from this technique is often as high as a factor of $(L_{max} - L_{min})$. For instance, while mining motifs in DNA datasets of [152], we look for motifs of lengths 6–15. In this case, combining the computation gives FLAME an advantage of 8X over the unoptimized algorithm.

Optimizing (L, f, d, k)

When mining a database for an (L, f, d, k) pattern, a special opportunity for faster execution exists if $d = 0$. When $d = 0$, the pattern must have all the mismatches in fixed positions and have no free mismatches. Therefore, instead of considering all strings of length L with at most f wildcard characters (?'s) over the alphabet A $U\{?\}$, we can consider a smaller model space. Since there are no free mismatches, we consider only those strings that occur *in the dataset* with at most f of the characters replaced with a wildcard character. We are still guaranteed to find the motif.

This reduced model space can be constructed by enhancing the *data suffix tree* by adding a node with an edge labeled "?" as a child for every existing node. The algorithm proceeds as described before with this new model tree. Before a (partial) model is evaluated, the algorithm checks to make sure that the number of "?"s is no greater that f.

As a result of this smaller model space, (L, f, d, k) searches with this optimization are

orders of magnitude faster when $d = 0$.

5.5 Evaluation

In this section, we present results from various experiments that were designed to test the effectiveness and performance of FLAME. We also compare FLAME with pattern mining algorithms from different application domains. Most existing algorithms can only work with (L, d, k) motifs and do not support the more general (L, M, s, k) model. Therefore, we carry out the comparison between FLAME and these existing methods using only the (L, d, k) model. Since we do not have a competing algorithm to compare the performance of FLAME on (L, M, s, k), we present a detailed analysis of the performance as different parameters in (L, M, s, k) are varied.

We use a variety of datasets including financial time series data, DNA sequences, protein sequences, and synthetically generated sequence data for our comparison. The characteristics of these datasets are summarized below:

Snake: This is a snake protein dataset from [81] that was considered for subsequence mining in [164]. It consists of 352 different snake venom protein sequences of varying lengths. The size of the dataset is about 28,000 symbols. The alphabet of amino acids (that make up the proteins) is of size 20. Such protein datasets are often analyzed in bioinformatics to find common patterns that might provide insights into their function.

Washington: A recent paper [152] compares several different DNA motif finding tools on a variety of datasets. The Washington dataset is actually a collection of 52 different datasets. It includes DNA sequences taken from several genes in Yeast, Mouse, Fruit Fly, and Humans, and also includes a few synthetic sequences. For a complete description, see [152]. The total size of this collection is 1.3 Million symbols.

IBM: This dataset contains second by second average price of IBM stock for all the trading

days in December 1999 [74]. To reduce the noise in the detailed dataset, we preprocess the data using the following standard data processing techniques that are designed to deal with short term volatility in stock price information [154]: First, the data is converted into a minute wise average price using a sliding window. And next, the price values are transformed into a percentage change with respect to the price in the previous minute. This technique is routinely used to compare movement data across different stocks that have a different face value. The resulting dataset contained 21 sequences from 21 days, each of length approximately 400 numbers, totaling 8,400 numbers.

Synthetic: In order to fully explore the space of data sizes and alphabet sizes, we use a synthetic data generation method that has been extensively used in several previous efforts [21, 50, 115, 118]. The data is generated as follows: Given the alphabet size, the number of sequences, and the size of each sequences, we generate random sequences by uniformly drawing symbols from the alphabet. We then randomly choose k sequences and *implant* a pattern of length L with d mismatches at random positions in each of the k sequences. This results in a dataset containing an (L, d, k) motif. The sizes of datasets we generate are comparable to those used in previous related papers [21, 50, 115, 118].

All the experiments in this section were performed on a 2.8 GHz Intel Pentium 4 processor with 2 GB of main memory. The operating system was Fedora Core 4 Linux, kernel version 2.6.11. All suffix trees were constructed using the TDD suffix tree construction algorithm [144]. We used the implementation of cSPADE available at [35]. The YMF implementation was obtained from [176], the Weeder implementation was obtained from [165], and the Random Projection implementation was obtained from [122].

5.5.1 Comparison with cSPADE

We first compare FLAME with cSPADE [180], a traditional subsequence mining algorithm, by mining for exact contiguous patterns. cSPADE [180] is a constrained subse-

quence mining algorithm based on SPADE [181]. cSPADE can be adapted to mine for contiguous subsequences by specifying an upper limit on the gaps in sequences to be zero. Since cSPADE was designed without approximate matches in mind, we can only compare it against FLAME on *exact* motif mining.

In this experiment, we use the Snake dataset. cSPADE requires the data to be preprocessed into a special format [181]. We do not include this preprocessing time in the comparison. The time shown for FLAME *includes* the time taken to construct the suffix tree. We run both algorithms to find exact motifs of lengths 3-14. The results are shown in Figure 5.6.

As is evident from Figure 5.6, FLAME is faster than cSPADE by an order of magnitude in each case. This result is perhaps not very surprising if one considers the fact that cSPADE and other subsequence mining algorithms like it, are designed for a different data mining problem (namely subsequence mining). Adapting cSPADE to mine for even a simple exact contiguous motif results in relatively poor performance compared to FLAME, which is specifically designed for motif mining.

5.5.2 Comparison with Random Projections

The Random Projections (RP) algorithm proposed by Bulher and Tompa [21] has recently been applied to time series data for motif mining [32]. RP is an approximate motif finding technique that works only for the special case of (L, d, k) patterns, and cannot work with the more general (L, M, s, k) model. This algorithm has also been applied to finding DNA motifs and is considered faster [21] than several popular algorithms such as MITRA [50] and WINNOWER [118].

The RP algorithm is based on the idea of "locally sensitive hashing" from [61]. Given L, d, and k, the algorithm chooses a p-position mask as a hash function. Then, the algorithm hashes all the l-mers in the database. If a sufficient number of l-mers hash to the

same bucket, it is likely that there is a motif that is similar to the l-mers in the bucket. Once a candidate bucket is identified, any local search algorithm can be used to search in the vicinity of the l-mers in the bucket for the (L, d, k) motif. In particular, RP uses an expectation maximization based algorithm like MEME [14] to search in the vicinity of "enriched" buckets. The main contribution in [21] is that they describe how to compute p, and the number of iterations for which the algorithm needs to be repeated for a certain level of confidence.

We compare FLAME and RP by performing a typical (L, d, k) motif mining task on datasets of varying sizes. In order to explore a wide range of database sizes, we use synthetically generated datasets (following the well established methods that have been used before for similar comparisons [21,50]). These datasets are generated (as described above) on a DNA alphabet of size 4. Each dataset contains 20 sequences. We vary the length of each sequence from 200 to 1000 symbols for each dataset. The datasets are implanted with a motif of length between 8 and 14 (chosen randomly). The algorithms do not know the actual length of the motif in advance (as is the case in any real task [152]). Both algorithms try to find $(L, d, 20)$ motifs for d = 1, 2 and L varying from 8 to 14. FLAME takes advantage of the technique described in 5.4.1 to combine the computation from different lengths. RP is run once for each value of L, and we add up the time from each run. (RP does not lend itself to combining computation.) RP is a heuristic technique, and in our evaluation we set it to find motifs with 95% confidence (the default setting). There is a 5% probability that RP might miss some motifs. The time taken by each algorithm for this task as the database size (i.e. the sequence length) varies is shown in Figure 5.7. The time taken to construct the suffix tree is a one time cost, and is less than 1 second for each dataset. It is not included in the execution times that we report for FLAME in the remainder of this section.

For the task of finding motifs with L varying from 8 to 14, and d=1 (denoted as $(8-14,1,20)$ in Figure 5.7), the RP algorithm works well for small database sizes. However, as the database size increases, we see that its performance begins to deteriorate rapidly. The reason for this deterioration is that with larger datasets most choices for the hash functions (the p-position mask) lead to a large number of "enriched" candidate buckets. This is especially true with shorter patterns. Exploring a candidate bucket is an expensive operation since it involves running an Expectation Maximization search.(The refinement step in [21].) When many buckets need to be explored to find the real (L,d,k) pattern, RP ends up taking much longer. FLAME, one the other hand, is relatively less sensitive to increases in the database size (Figure 5.7). A larger database will lead to a model being pruned deeper in the model tree, but FLAME still manages to avoid a lot of redundant computation by virtue of using the suffix tree to efficiently prune the model space. For the $(8-14,2,20)$ task, RP takes too long to complete for sequence lengths beyond 400, and we do not report these times in Figure 5.7.

5.5.3 Comparison with Weeder and YMF

Many algorithms have been proposed in the field of computational biology for finding motifs. Most of these algorithms deal with (L,d,k) type motifs [21, 50, 118, 135]. A recent study [152] compared several algorithms, and determined that Weeder [115] and YMF [135] performed among the best. Weeder scored highest on many performance metrics, and YMF did nearly as well. In this section, we compare FLAME with these two algorithms.

Comparison with Weeder

Weeder is a very fast heuristic algorithm that was specifically designed to find motifs in DNA datasets. The algorithm is limited to the (L,d,k) model and does not work with

the more powerful (L, M, s, k) model. Weeder is extremely fast because it assumes that the mismatches are distributed uniformly across the length of the motif. While looking for a motif of length 10, if it finds 2 mismatches after examining the first 3 symbols of a sequence, it eliminates the string because it assumes that it is highly unlikely that the remaining 7 symbols will match correctly. As a result of this assumption, Weeder can prune the search space very quickly, but it is not guaranteed to be accurate. Weeder cannot find motifs whose instances have mismatches *not* distributed uniformly across the length of the motif.

We perform a simple experiment to determine the accuracy of Weeder. We use the Washington dataset [152] that is based on the real motifs found in the TRANSFAC [153] database. We run both algorithms on the Washington dataset using a variety of models. The implementation of Weeder [165], only works for motifs of even lengths between 6 and 12, so we limit FLAME to these lengths too. We present the number of motifs found by Weeder as a percentage of the total number of motifs present in the dataset. Since FLAME is an accurate algorithm, it always finds all the motifs in the dataset, and we do not show its accuracy (100%) in the graph. These results are summarized in Figure 5.8. As one can readily observe, Weeder find a large portion of the simpler patterns, but as the patterns get more complex, Weeder misses a large number of them. In fact, for motifs such as $(12, 2)$, Weeder finds less than 5% of the total number of motifs found by FLAME. However, the one point in favor of Weeder is speed. It takes only one second to find a (10,2,20) motif while FLAME takes close to 40 seconds. Weeder pays the price for this speed with a very low accuracy.

The task of predicting regulatory elements is a two step process. First a pattern finding tool such as Weeder or FLAME can be used to find all the patterns that frequently occur in the dataset being considered. The second step is to examine these patterns and score them

on various factors such as strength of the motif, biological importance, statistical significance, etc. The second step requires domain knowledge to distinguish between patterns that are real regulatory sequences versus random matches to the background "junk DNA". Biologists employ many heuristics for the second phase to varying degrees of success, and often requires some manual processing. The first phase is orthogonal, and any pattern finding tool can be used and paired with a different scoring/ranking procedure.

Figure 5.8 shows that while FLAME finds all the candidate motifs, Weeder might miss a significant fraction. Finding more results in the first phase of the computation is certainly beneficial since we will be better informed going into the second phase of ranking the patterns found, and therefore stand a better chance of identifying the best motifs.

To demonstrate the effectiveness of FLAME in finding real biological motifs that are missed by Weeder, we performed the following experiment: We list all the candidate motifs found by FLAME in the Washington dataset and rank them using the *same* scoring function as Weeder's. We observed that FLAME was able to correctly identify several motifs that Weeder missed. For instance, FLAME reports TCGTAACG on human dataset *hm08r*, CGACGTATGC on *hm11g*, and CGTACGAT on *hm16r*. Weeder offers no prediction on any of these data sets. These motifs do not appear in the list of several hundred potential motifs that Weeder finds in the first phase before it starts scoring them. Therefore, irrespective of the scoring method used, Weeder could not have reported the motifs for these and several other similar datasets. Since FLAME explores the entire model space, it does not miss any motifs, and is therefore able to detect the correct motif.

Since Weeder has a very low accuracy, we do not consider it for experiments in the remainder of this section.

Comparison with YMF

Another algorithm that performed well in the comparison in [152] is YMF (Yeast Motif Finder). YMF is a simple and accurate algorithm that finds *all* patterns that appear more frequently than expected in a set of DNA sequences. Like Weeder, YMF too cannot be used for (L, M, s, k) models. It simply has a counter corresponding to each possible motif in the model space. It scans the database once using a sliding window and augments the count for each motif that matches the sliding window. One can easily see that YMF will scale linearly with the size of the database, but will scale very poorly with the size of the model space since it keeps a counter for each possible model. YMF becomes impractical for longer, complex motifs.

We demonstrate this behavior using a synthetic dataset containing 20 sequences, each 600 symbols long. We implant different (L,d,20) motifs in the sequence. We run YMF and FLAME on a variety of (L, d, k) motifs. The results are averaged over 50 datasets. The results of this experiment are presented in Figure 5.9. For the (8,1) motif, both YMF and FLAME finish very quickly. However, we can easily see that YMF does not scale well as the motif complexity increases. For the $(12, 3)$ motif, YMF did not finish in a reasonable amount of time, and we had to terminate the program after two hours. FLAME, on the other hand, completes in less than two minutes. We conducted similar experiments by varying the sequence length from 200 to 1000. FLAME continues to be faster than YMF for these settings, and we omit presenting the results in the interest of space.

We devote the rest of the evaluation section to study the performance characteristics of FLAME as different parameters in the problem setting are varied.

5.5.4 Performance Characteristics of FLAME

Alphabet Size

Our next experiment studies the effect of alphabet size on execution time. For this task, we again use the synthetic dataset generator. We vary the alphabet size from 5 to 50, and at each point evaluate the execution time for various implanted patterns. Each dataset consisted of 20 sequences, each of length 600, totaling 12,000 symbols. The execution time for various implanted motifs is summarized in Figure 5.10.

As can be seen in the figure, execution times for simpler motifs such as $(6, 1), (8, 1)$, and $(10, 1)$ grow slowly with alphabet size. Complex motifs, such as $(8, 2)$ and $(10, 2)$, which inherently require the algorithm to search a larger space, grow faster with alphabet size. Nevertheless, the mining task is often completed within a few hours even for very large alphabets. Several real world applications such as DNA sequence mining, and protein sequence mining typically require an alphabet of size less than 25, and can be mined very quickly with FLAME.

Mining Time Series Data

We now study the performance of FLAME for different parameters of the (L, M, s, k) motif model. (Since existing algorithms do not support the (L, M, s, k) model, we do not compare FLAME with any other algorithms for the rest of this section.)

In this experiment, we use the (L, M, s, k) model to mine the IBM dataset. We use a 20 bucket histogram that partitions the dataset into roughly equal sized buckets. We then assigned a symbol to each bucket, and encoded the numerical series into a symbolic sequence. The dataset totaled about 8,400 symbols. The distance penalty matrix is a squared error matrix using the numerical values corresponding to each symbol. That is, $M(A, B) = |v(A) - v(B)|^2$, where $v(A)$ is the numerical value corresponding to the

symbol A (the midpoint of the bucket in the histogram). In effect, this is the L_2 norm.

We present the time taken by FLAME to find several (L, M, s, k) motifs. First we set the support to be 21 (equal to the number of sequences in the dataset). We run FLAME for L = 5, 8, and 11, while varying the distance threshold s. The results of this experiment are shown in Figure 5.11. We observe from the figure that as the threshold is increased the time taken to execute the search increases. This is because at higher thresholds, the pattern is more relaxed, and the space of potential models that needs to be searched is larger. FLAME is able to find models of length 11 within 16 seconds. We then repeated the experiment for higher support values of 60 and 120. These results are shown in Figures 5.12 and 5.13 respectively. As can be seen in these figures, increasing the support decreases the amount of time taken. This is because of the fact that a higher support causes more aggressive pruning of the search space, and hence, a lower execution time.

Mining Protein Sequences

Next, we examine the performance of FLAME on the Snake dataset [81]. Protein motif mining is a good example of an application where the (L, M, s, k) model offers a significant advantage over using less powerful models. The (L, d, k) and (L, f, d, k) models cannot capture the notion of similarity required for mining protein motifs. The (L, M, s, k) model is the only model that allows us to use popular similarity matrices like PAM30 [38] and BLOSUM [70], and is therefore essential for applications such as protein motif mining. (PAM30 is a substitution matrix that is commonly used in life sciences application to compute scores when searching for proteins based on sequence similarity.)

In this experiment, we look for (L, M, s, k) motifs using PAM30 as the distance matrix. We fix the support to be 175 (roughly half the number of sequences) to find patterns that are common to snake venom proteins. (Protein sequence mining typically uses high thresholds [81].) We varied (L, s) as $(6, 10)$, $(8, 10)$, $(10, 10)$, $(10, 20)$, $(12, 20)$, and $(12, 30)$.

The results are shown in Figure 5.14. As we can see from this figure, the computation time increases with an increase in the distance threshold. A higher distance threshold indicates a more relaxed pattern – which in turn means that FLAME has to proceed deeper down the model tree before it can start eliminating models. As can be observed from Figure 5.14, even the longest motifs are found reasonably quickly.

Scaling to Large Datasets

Finally, we demonstrate the scalability of the FLAME algorithm for mining motifs on very large datasets. Motif mining is a difficult task, and existing algorithms focus on relatively small datasets (of the order of 10,000 symbols). We show that using FLAME, it is possible to scale to much larger database sizes. We generate synthetic datasets, and embed a motif of length chosen randomly between 8 and 14 in 10% of the sequences. The datasets contain sequences of length 1000, and the number of sequences is increased gradually to generate database of increasing sizes. The total database size is varied from 20,000 symbols to 1 million symbols. We run FLAME on these datasets to find $(8-14, 1)$ and $(8-14, 2)$ models with 10% support. The results for this experiment are shown in Figure 5.15.

The execution time increases relatively slowly (Figure 5.15) as we increase the database size. In the case of $(8-14, 1, 10\%)$ motifs, the time increases from 7 seconds to 55 seconds over the entire range. In the case of $(8-14, 2, 10\%)$ motifs, the time increases from 290 seconds to 5900 seconds. As one would expect, the time to mine more complex motifs grows a little faster. However, even patterns of length 14 in a database this large can be mined in a few hours. To our knowledge, none of the existing algorithms can accurately scale to such large database sizes.

5.5.5 Summary

In this section, we evaluated FLAME on a number of real and synthetic datasets. The results demonstrate that for motif mining, FLAME is an order of magnitude faster than the (constrained) subsequence mining algorithm cSPADE. The results also show that FLAME is faster, and scales better than other algorithms that have been used for time series mining, such as Random Projections. In addition, comparison of FLAME with two of the best algorithms used in computational biology, namely Weeder and YMF, shows that:

1. Weeder is fast, but misses a significant number of motifs (more than 90% for complex motifs). On the other hand, FLAME is guaranteed to find *all* motifs in the dataset.

2. YMF, like FLAME, is 100% accurate, but is very slow. Compared to YMF, FLAME is faster by more than an order of magnitude.

We also conducted experiments to test various characteristics of FLAME. These experiments reveal that FLAME performs well in a variety of mining situations, and scales to datasets much larger (1 million symbols) than has been attempted before.

5.6 Conclusions and Future Work

In this chapter, we presented a powerful new model: (L, M, s, k) for motif mining in sequence databases. The (L, M, s, k) model subsumes several existing models and provides additional flexibility that makes it applicable in a wider variety of data mining applications. We also presented FLAME, a flexible and accurate algorithm that can find (L, M, s, k) motifs. Through a series of experiments on real and synthetic datasets, we demonstrate that FLAME is a versatile algorithm that can be used in several real motif mining tasks. We also show that FLAME outperforms existing subsequence mining algorithms (cSPADE) and time series mining algorithms (Random Projections) by more than an order of magni-

tude. FLAME is also superior to motif finding algorithms used in computational biology (more accurate than Weeder, significantly faster than YMF).

```
FLAME (modelTree, dataTree, l, d, k)
1. model = modelTree.FirstNode()
2. While (model ≠ modelTree.LastModel())
3.      Evaluate_Support(model,dataTree)
4.      If ( isValid(model) )
5.          Print "Found Model: ", model
6.      Else If(model.support() < k)
7.          modelTree.PruneAt(model)
8.      model = NextNode(model,modelTree)
9. End While
10.End

Sub Evaluate_Support (model, dataTree)
1. newsymbol = last symbol of model.String
2. oldmatches = model.Parent().Matches()
3. newmatches = EmptyMatches()
4. If (model.Parent() == root)
5.      newmatches = Expand_Matches(root,newsymbol,dataTree)
6. Else
7.      ForEach match x in oldmatches
8.          newmatches = newmatches U
                Expand_Matches(x,newsymbol,dataTree)
9.      End ForEach
10.model.SetMatches(newmatches)
11.Return

Sub Expand_Matches (x, newsymbol, dataTree)
1. Let Y = Set of all single character expansions of x.String
       in dataTree
2. ForEach element b in Y
3.      If b's last symbol ≠ newsymbol
4.          b.mismatches ++
5.          If b.mismatches > max_mismatches
6.              Remove b from Y
7.      End ForEach
8. Return Y
```

Figure 5.4: The FLAME Algorithm

Sub Expand_Matches_lMsk (*x, newsymbol, dataTree*)
1. Let Y = Set of all single character expansions of x.String
   in dataTree
2. ForEach element b in Y
3. b.distance += Distance_Matrix(b.lastsymbol,newsymbol)
4. If b.distance > max_distance
5. Remove b from Y
6. End ForEach
7. Return Y

Figure 5.5: Functions for (L,M,s,k)

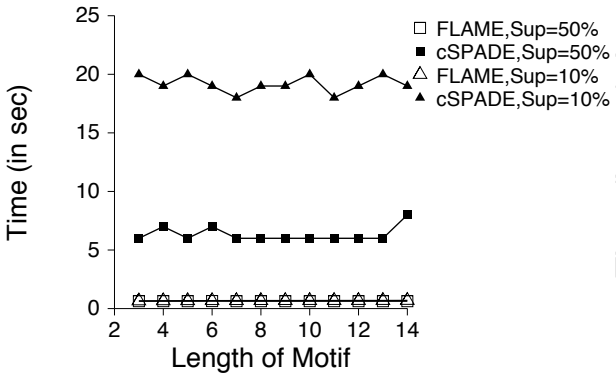

Figure 5.6: cSPADE vs FLAME on the Snake dataset for different length exact motifs at supports of 10% and 50%.

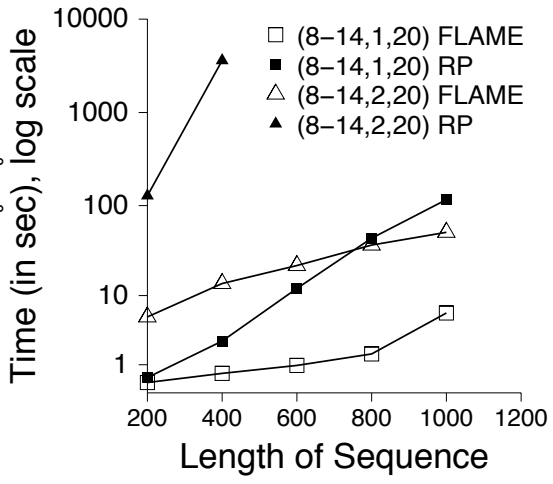

Figure 5.7: RP vs FLAME for varying database sizes. Note that the time axis is on a log scale.

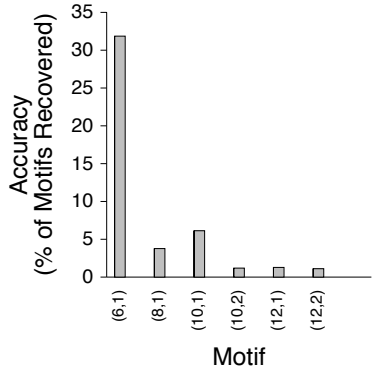

Figure 5.8: Weeder - Accuracy on real DNA datasets. FLAME is guaranteed to be 100% accurate, and is not shown here.

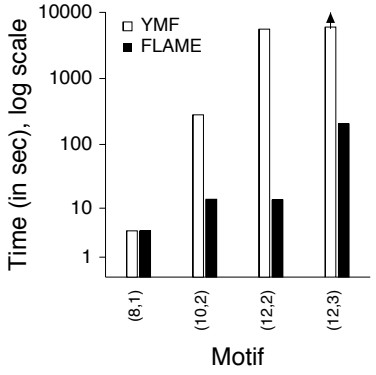

Figure 5.9: YMF vs FLAME on synthetic datasets. Note that the time axis uses a log scale.

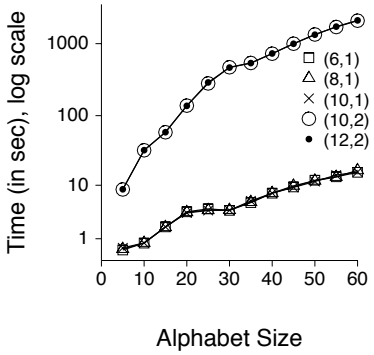

Figure 5.10: Performance as alphabet size varies.

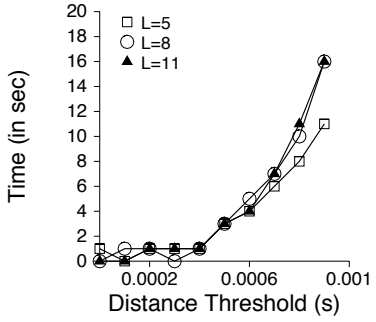

Figure 5.11: FLAME: Distance threshold vs time taken for (L,M,s,k) motifs on IBM stock price data at support = 21.

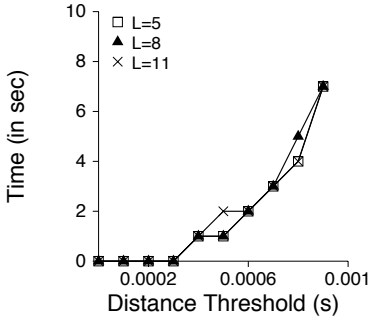

Figure 5.12: FLAME: Distance threshold vs time taken for (L,M,s,k) motifs on IBM stock price data at support = 60.

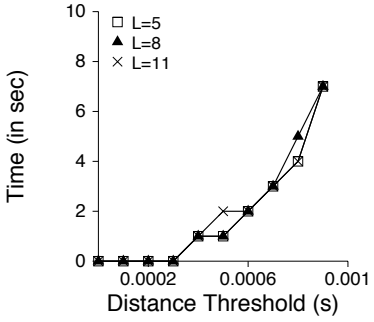

Figure 5.13: FLAME: Distance threshold vs time taken for (L,M,s,k) motifs on IBM stock price data at support = 120.

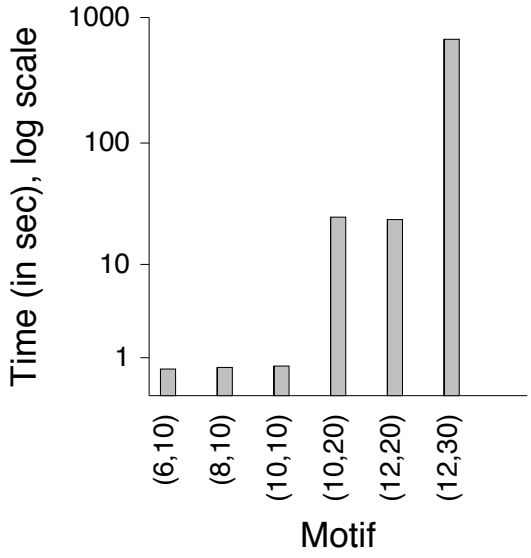

Figure 5.14: FLAME: (L,M,s,k) motifs on the Snake dataset.

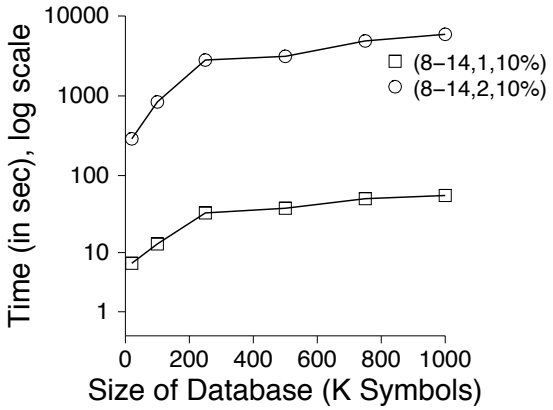

Figure 5.15: Scalability of FLAME with increasing database size.

CHAPTER VI

Application – GeneFinder

6.1 Introduction

Having sequenced the genomes of several organisms, the next major challenge for the life sciences community is to understand the regulatory networks of an organism. A key step to understanding the regulatory code is determining all the binding sites and therefore the regulatory targets of each transcription factor. Much of the work in this area focuses on two critical tasks: a) discovering motifs/binding sites through computational analysis and b) using a known motif to predict other regulatory targets of the transcription factor.

Many high quality algorithms have been developed for discovering a motif or a binding site by analyzing the promoter regions of genes thought to be regulated by the same transcription factor [139]. This is frequently done by clustering genes based on gene expression data, and then using sequence analysis techniques on the promoter regions. Experimentally verified binding sites are available for several transcription factors in databases like TRANSFAC [153] and JASPAR [162]. Several motif discovery algorithms are described and compared in [152]. This problem is known to be difficult since a given transcription factor may bind to significantly different sites.

In this chapter, we do not focus on the motif discovery algorithm. Instead, we focus on problem of representing the information about binding sites and using it to predict other

binding sites. This problem has received much attention in recent years. A solution to this problem is key to understanding transcription and regulation. The binding site signatures of a transcription factor are represented using the consensus sequence, the position weight matrix (surveyed in [139]), or the sequence logo [130]. Locating approximate matches to a consensus sequence or a position weight matrix in the promoter regions of the gene of interest is the predominant strategy used to predict if a gene is a potential target of a transcription factor. Merely locating matches to a consensus sequence can result in a large number of false positives. For instance, consider the case of the activating transcription factor ATF3. The consensus sequence for ATF3 is TGACGTCA [172]. If we simply search the promoter regions of all the genes in the mouse genome, we find nearly 1000 hits. ATF3 is however known to regulate less than a few dozen genes. Clearly, locating the targets of a transcription factor is not an easy task, and many other pieces of information in addition to the binding site consensus sequence are required to reduce the false positives and solve this problem.

Our current understanding of the mechanisms of transcription may be insufficient to computationally determine *all* targets of a transcription factor. In fact, the very question might be ill-posed in the sense that different binding sites are likely to be effective to different extents, and therefore lead to different levels of activation (or repression). However, it is reasonable to expect that it is possible to computationally predict the major targets of a transcription factor with high confidence.

In this chapter, we present GeneFinder – a program that combines various pieces of information to produce a ranked list of candidate targets for a given transcription factor. Given the position weight matrix from a source like TRANSFAC, JAPSAR, or from literature, GeneFinder uses the position of the binding site relative to the transcription start site, the degree of conservation of this binding site across closely related species, tissue specific

expression data, and multiplicity of binding sites to compute a more accurate ordering of the candidate targets.

Previous approaches have employed only one or two of these techniques [86, 172] or have restricted themselves to a smaller scale [120]. Some tools like those provided by Genomatix [58] perform a subset of this analysis one promoter region at a time. However, they do not offer genome scale analysis. GeneFinder builds on these approaches by solving this problem in a scalable way that permits genome-wide searches in a fraction of the time that other tools take to perform less detailed analyses. At the core of this approach is a suffix tree based algorithm that can locate matches to position weight matrices extremely quickly. In addition GeneFinder can rapidly incorporate other data sources to reorder the candidate list. Further, GeneFinder makes it extremely easy to add new sources of data that can further refine the scoring of candidate hits. This is made possible through a modular approach that weights each data source independently and incorporates it using Bayesian reasoning.

The rest of the chapter is outlined as follows: Section 6.2 describes the algorithms we use to search the sequence, and then refine the list of hits to produce a re-ordered list of candidate targets. Section 6.3 describes the results of a few GeneFinder queries. We show that GeneFinder is not only able to find known targets of transcription factors, but also offer several predictions for new targets that seem very promising. Finally, Section 6.4 summarizes the chapter, and presents our conclusions.

6.2 Methods

This section details the algorithms used by GeneFinder. We first describe the suffix tree based algorithm that is used to locate sequence matches. We then describe the techniques used to incorporate position information, phylogenetic information, tissue specific

expression data, and the multiplicity of binding sites to rescore these candidate targets.

6.2.1 Searching the genome

Given a position weight matrix, the first step is to locate all candidate matches to the position weight matrix that score above a certain threshold in the genome of interest. A simple algorithm for this is to use a sliding window of size equal to the length of the matrix and scan the entire sequence. This approach is slow, and may take several hours if we wish to scan multiple genomes. GeneFinder overcomes this hurdle by using a suffix tree based algorithm. The basic idea of the algorithm is to first construct a suffix tree for the sequences that need to be searched. This is a one time cost that gets amortized over many searches. For each position weight matrix, we explore the suffix tree to prune out branches of the tree that do not have promising matches. This approach lets us evaluate each subsequence against the matrix only once, irrespective of the number of times it appears in the tree. We briefly describe the suffix tree and the algorithm below.

Suffix Trees

A suffix tree is described by [64] as a tree type data structure on a string S where each of its n suffixes is represented as a path from the root to a leaf. The out-degree from each of the nodes is $O(\mid \Sigma \mid)$ where Σ is the alphabet. Thus given a suffix tree, a substring of length p can be found or proved to not exist in time $\Theta(p)$.

Figure 6.1 shows a suffix tree on the sequence “ATTAGT$”. By traversing from the root downward, one can determine if a substring is present in the tree. The suffix tree provides unambiguous paths for a traversal algorithm.

GeneFinder uses suffix trees constructed by the TDD algorithm described in [151]. This is a disk-based suffix tree construction algorithm that makes it possible to construct suffix trees on disk for very large sequences. By comparison, popular in-memory construction

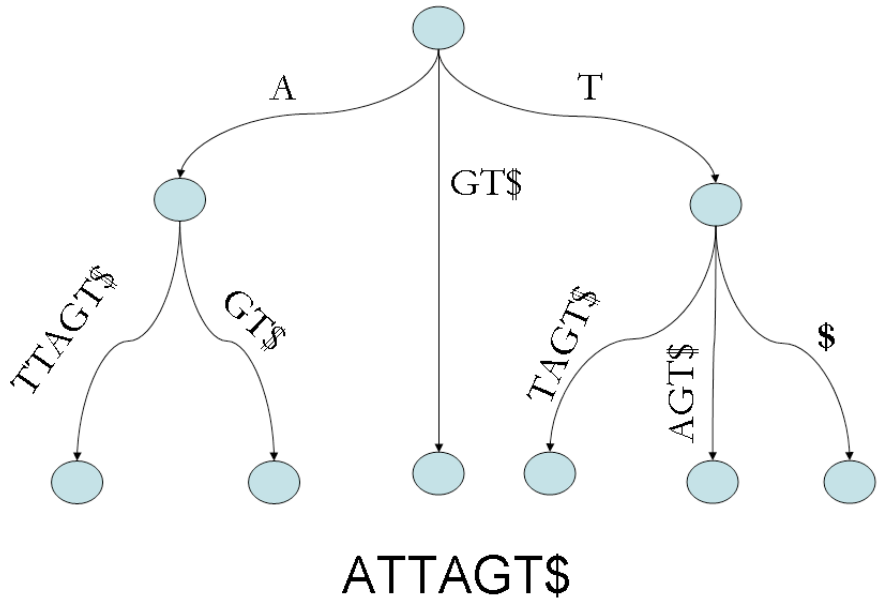

Figure 6.1: A Suffix Tree

algorithms like Ukkonen [156] or McCreight [103] are several orders of magnitude slower when dealing with very large sequences where the size of the resulting suffix tree exceeds the amount of main memory.

Algorithm

The algorithm used to search the suffix tree for matches to the matrix is similar to the approach in [45, 171]. However, our algorithm uses a disk-based suffix tree in order to scale to large sizes. We use a best first exploration of the space of all possible matches. We start by matching the matrix with all nodes in the first level in the tree, we store each (partial) match with the current score and the maximum possible final score. The maximum possible final score can be computed as the score that will be obtained if the partial match is expanded using the best possible symbols that lead to the highest score. If

this maximum possible score is below the threshold level, the partial match is discarded. The highest scoring partial match is then expanded to the next level, and so on. A priority queue is used to store the partial matches so that they can be retrieved in best first order efficiently.

A	C	G	T
12	13	50	25
80	10	5	5
100	0	0	0
0	90	0	10
0	80	10	10

Table 6.1: A Sample Position Weight Matrix

Table 6.1 shows a position weight matrix of length 5. If we assume that the scores are additive, the score for the sequence "CCATG" would be 13 + 10 + 100 + 10 + 10 = 143. The maximum possible score is for "CAACC" = 400. In GeneFinder, the weights in the position weight matrix are normalized so they add up to 1. The scoring is multiplicative – that is the score from each position is multiplied with the score from the next position and so on. Equivalently, if we store the position weight matrix with log values, we use additive scoring.

6.2.2 Refining Candidates

GeneFinder uses four main ideas to refine the rank of candidate matches:

1. Position of the binding site relative to transcription start site
2. Conservation across related species
3. Tissue Specific Gene Expression
4. Multiple Sites

We discuss these factors in more detail in the following sections.

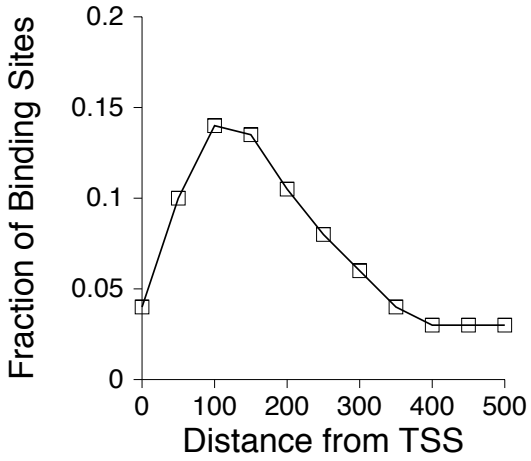

Figure 6.2: Distance of Binding Site from the Transcription Start Site

Position

The position of the binding site relative to the transcription start site has been shown to be of significance [120,172] in eukaryotes. Qian et. al. [120], show that transcription factor binding sites may have a strong preferred location. A binding site located at a 'preferred' distance is more likely to be a real candidate than one located at a highly 'non-preferred' distance. The distribution of the distance of the binding site from the transcription start site is shown in Figure 6.2. This is based on examination of over 200 binding sites from TRANSFAC [153]. As we can see from the figure, a majority of the binding sites occur between 50 and 250 bases upstream of the transcription start site. GeneFinder uses simple Bayesian reasoning to incorporate the position information into the scoring algorithm. We use the following equation:

$P(H|P = p) = \frac{P(P=p|H) \times P(H)}{P(p=p)}$. Here, $P(H)$ is the probability that a given hit is a true motif. $P(H|P = p)$ is the probability that the hit being considered is a true binding site given its distance from the transcription start site, and $P(P = p)$ is the probability that a

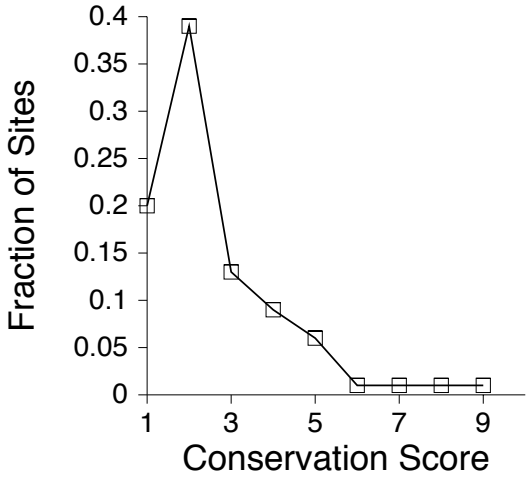

Figure 6.3: Distribution of Conservation Scores

random sequence (of the same length as the binding site) would match at a distance of p from the transcription start site.

Given the position of the hit, we can compute $P(P = p|H)$ from Figure 6.2. $P(H)$ is empirically estimated from the raw match score as in [120]. We use a uniform random distribution to model $P(P = p)$.

Conservation across related species

A binding site that is conserved across multiple closely related species is more likely to be a functional element than one that is not. The reasoning behind this is that a conserved sequence is likely to have been evolutionarily selected for because of its function. Several studies have used this approach [86,172]. Again, we use Bayesian reasoning to incorporate this information into our scoring model. We compute $P(H|C = c) = \frac{P(C=c|H)\times P(H)}{P(C=c)}$. Here, $P(H)$ is the probability of the hit being a true binding site. $P(C = c|H)$ is the probability that the site will obtain a conservation score of c given it is a true motif, and finally, $P(C = c)$ is the probability of a random sequence obtaining a conservation score of c. Although any complex conservation score is admissible, we use a simple metric – c

as the number of species across which the binding site is conserved.

In order to compute the numerical values for each of these probabilities, we constructed a training data set based on [172]. This chapter presents a list of regulatory elements identified by comparing the promoters of orthologous genes across multiple species. They identify conserved segments that appear in multiple promoters to identify them as potential binding sites. Using the motif conservation score from this data, we computed the distribution for $P(C = c|H)$. Figure 6.3 shows this distribution. $P(C = c)$ is computed assuming a conservation rate of 6.8% for a random 8-mer. If the average phylogenetic distance between the species being compared is much different from this, then the number can be appropriately adjusted to accurately reflect the significance of conservation of the binding site.

Tissue Specific Gene Expression

Tissue specific gene expression data can also be incorporated using Bayesian reasoning to help differentiate random hits from true candidates. If the transcription factor and the target are not known to be expressed in the same tissues, it is less likely that the putative target is real. However, if expression data shows that the tissue and the target are both expressed in some tissues, this can be interpreted in favor of the target. This reasoning has been used in previous approaches such as [120]. However, instead of using it as a strict filter, we use a Bayesian formula to incorporate this information:

$P(H|T = t) = \frac{P(T=t|H) \times P(H)}{P(T=t)}$. Here, $P(T = t|H)$ is the probability that tissue specific expression score is t given that the target gene is a true hit. $P(H|T = t)$ is the probability that the hit being a true binding site given the tissue specific expression score. Finally, $P(T = t)$ is the probability that a random hit obtains a tissue specific expression score of

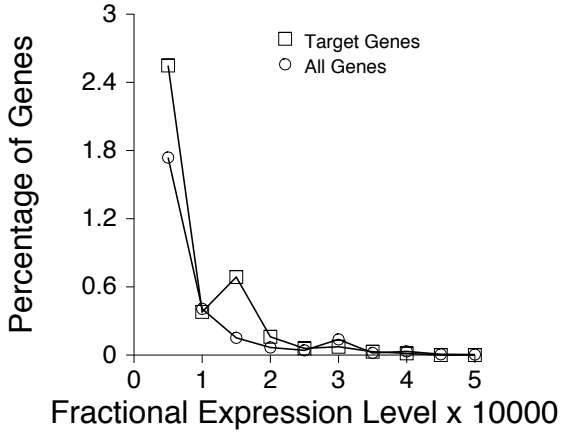

Figure 6.4: Tissue Expression Distribution For Eye Tissue

t. We approximate $\frac{P(T=t|H)}{P(T=t)}$ with $\frac{P(expressionlevel=l|H)}{P(expressionlevel=l)}$ where l is the expression level of the target being considered in the tissue of interest. These probabilities are estimated from the Unigene data [157] that lists the expression level for different genes in different tissues. If the transcription factor is known to be expressed in multiple tissues, we then use the geometric mean of the ratios (computed as above) for each tissue.

The distribution of $P(expressionlevel = l|H)$ and $P(expressionlevel = l)$ for eye tissue are shown in Figure 6.4. These are computed using data from [172] and Unigene [157]. As we can see from the figure, targets of transcription factors known to be expressed in the eye are more likely to have a higher level of expression in eye tissue than random genes.

Multiple Sites

Recent studies have observed that many genes are regulated by multiple transcription factors and have tried to exploit this information for predicting transcription factor binding sites [47, 67]. Promoter regions often have multiple occurrences of binding sites. Based on the data in [172], we plotted the number of binding sites in a promoter region as shown

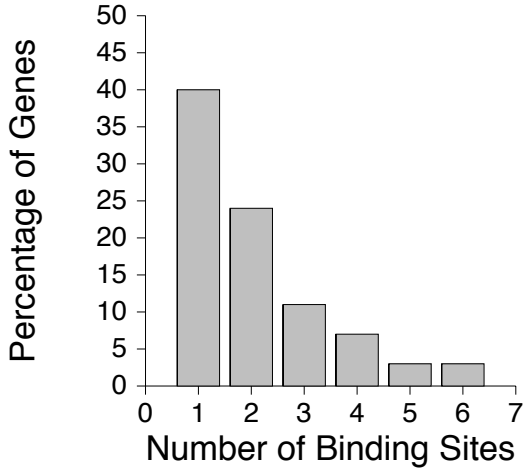

Figure 6.5: Number of Binding Sites per Promoter

in Figure 6.5. As we can observe from the figure, over 60% of the genes from [172] have more than one binding site in the promoter region.

GeneFinder exploits this information using the following equation: $P(H|R = r) = \frac{P(R=r|H)\times P(H)}{P(R=r)}$. Here, $P(R = r|H)$ is the probability of observing r occurrences of binding sites in the promoter region given that it is a true target. This value is computed using Figure 6.5. $P(R = r)$ is the probability of finding r binding sites by random chance. This is computed as the probability of finding r matches with a given stringency in a random string of length 5000 bases.

To compute the overall probability that a candidate is an actual target, we use the following formula that includes all of the above four factors:

$$P(H|P = p \wedge C = c \wedge T = t \wedge R = r)$$

$$= P(H)^4 \times \frac{P(P=p|H)\times P(C=c|H)\times P(T=t|H)\times P(R=r|H)}{P(P=p)\times P(C=c)\times P(T=t)\times P(R=r)}$$

where:

H is the event that the hit being considered is a true target.

Gene	Score	Known	Description
Pde6a	0.496	√	phosphodiesterase 6A, cGMP-specific, rod, alpha [129]
Dnajc13	0.238		DnaJ (Hsp40) homolog, subfamily C, member 13
LOC624446	0.130		
Pla2g7	0.097	√	phospholipase A2, group VII (platelet-activating factor acetylhydrolase, plasma) [178]
Nr2e3	0.075	√	nuclear receptor subfamily 2, group E, member 3 [83]
LOC620844	0.074		
LOC547349	0.061		
1110037F02Rik	0.044		
Serpinb8	0.039		serine (or cysteine) peptdiase inhibitor, clade B, member 8
LOC620173	0.033		
2410075B13Rik	0.029		
Kctd8	0.027		potassium channel tetramerisation domain containing 8
LOC624988	0.027		
LOC628754	0.027		
Snrpd2	0.023		small nuclear ribonucleoprotein D2
Cct4	0.022		chaperonin subunit 4 (delta)
Arhgap5	0.021		Rho GTPase activating protein 5
1300006C06Rik	0.016		
Slc6a9	0.015		solute carrier family 6 (neurotransmitter transporter, glycine), member 9
Iqgap1	0.014		IQ motif containing GTPase activating protein 1
LOC629548	0.014		
Olfr1069-ps1	0.013		
Gm381	0.012		gene model 381
LOC623881	0.012		
LOC625988	0.012		
Nav3	0.012		neuron navigator 3
LOC629553	0.012		
Tesk1	0.012		testis specific protein kinase 1
Il17e	0.010		interleukin 25
LOC625838	0.010		

Table 6.2: Predicted Targets of Nrl

P is the distance of a hit of given score from the transcription start site.

C is the extent of conservation of a hit of given score.

T is the tissue expression score of the target gene with respect to the TF (described below).

R is the number of times a hit of given score repeats in the promoter region.

In the above approximation, we make the assumption that each of these four factors is independent.

6.3 Results

In this section, we show through multiple experiments that GeneFinder not only recovers known targets of transcription factors, but also predicts several novel targets. In the following experiments, GeneFinder uses the genomes of Human, Mouse, and Dog as downloaded from NCBI [109]. We extract the promoter regions (5kb upstream of the

Gene	Score	Known	Description
LOC624541	0.231		
LOC620617	0.092		
Mapk10	0.088		mitogen-activated protein kinase 10
Rho	0.087	√	Rhodopsin (retinitis pigmentosa 4, autosomal dominant) [123]
Chi313	0.087		
Syne2	0.082		synaptic nuclear envelope 2
Ccna1	0.081		cyclin A2
A330044P14Rik	0.080		
Nr1d1	0.072		nuclear receptor subfamily 1, group D, member 1
Rom1	0.0645	√	Retinal outer segment membrane protein 1 [178]
Rab11b	0.057		RAB11B, member RAS oncogene family
4931419H13Rik	0.055		
LOC625262	0.053		
F830045P16Rik	0.051		
Serpinb12	0.047		serine peptidase inhibitor, clade B (ovalbumin), member 12
Sdk1	0.046		sidekick homolog 1
LOC624994	0.045		
LOC625300	0.042		
LOC627883	0.032		
LOC383546	0.030		
Rnf190	0.026		ring finger protein 190
AI607873	0.024		
Trpa1	0.024	√	transient receptor potential cation channel, subfamily A, member 1 [178]
LOC620926	0.23		
LOC210714	0.022		
Mcpt2	0.022		mast cell protease 2
Bsdc1	0.021		BSD domain containing 1
Gm847	0.019		gene model 847, (NCBI)
LOC621042	0.019		
Slc16a6	0.018		solute carrier family 16 (monocarboxylic acid transporters), member 6

Table 6.3: Top Results For NRE

transcription start site) from each annotated gene in the genome, and construct a suffix tree on this set of sequences for each organism. In each case, the promoter sequence file contained about 110 million bases, and the suffix trees were each about 1 GB. We present three anecdotal pieces of evidence that demonstrates how GeneFinder works.

A Note on Statistics We do not compute the overall E-value of the score that is reported here. GeneFinder however makes it possible to compute the E-value of the sequence score and filter out results based on the E-value before further rescoring based on position, tissue expression, and phylogenetics. In this section, we present the score and the details for the top thirty hits.

Gene	Score	Known	Description
LOC646737	0.120		Similar to ribosomal protein S14
DKFZP781I1119	0.114		Mesoderm induction early response 1
SLD5	0.110	$\surd$	Component of GINS, heterotetramer that is regulated by ERα [69]
NUP205	0.108		Nucleoporin
FRG1	0.090		FSG1, FSHD region gene (muscular dystrophy)
FLJ38608	0.087		
FUT7	0.086		1,3 fucosyltransferase
LAMA4	0.085		Laminin α 4
LOC642515	0.082		
GTF2IRD1	0.082		MusTRD1/BEN, interacting with RB1
MGC72104	0.081		Putative FRG1-like protein C20orf80
LOC442293	0.075		
Loc284751	0.068		
C3orf27	0.067		
LOC647190	0.063		
SLC2A2	0.059	$\surd$	GLUT2, GLUT1,3,4 expression augmented by EE [29]
C9orf112	0.055		
MTDH	0.040	-	Metadherin, overexpressed in metastatic breast cancer [20]
ACTL7	0.039	$\surd$	Actin-like-7-β, Actin reported to be ER binding [159]
HSPA9B	0.039	$\surd$	Mortalin, Induced by estrogen [167] [97]
DEPDC6	0.039		
LOC389124	0.038		
LOC645515	0.038		
GML	0.037	-	LY6DL, signal transduction by p53 type mediator [11]
POU5F1	0.034		Octamer binding TF3
YTHDF1	0.034		DACA-1
BAT5	0.033	-	G5, HLA-B associated transcript 5, related to cancer development/progression [163]
TPD52L2	0.032	$\surd$	Expressed in breast cancer and known to be estrogen responsive [19] [82]
C9orf86	0.029		
ITFB8	0.028		Integrin β8

Table 6.4: Top ER results

6.3.1 Nrl binding

Nrl is a basic motif-leucine zipper DNA binding protein known to be expressed in the retina [141]. Nrl is suspected to play an important role in regulating expression of various retina specific genes. We queried GeneFinder using the consensus sequence for the Nrl binding site in mouse [123]: TGATCCTCATRATC. (Recall that 'R' represents a position where A and G may occur with equal likelihood.) The mouse genome was the target genome. The thirty best matches are shown in Table 6.2

The first hit is Pde6a, a well known target of Nrl in mice [129]. Several of the hits are predicted but as yet uncharacterized genes. Pla2g7 was shown to be downregulated in the absence of Nrl in [178], and is suspected to be directly or indirectly downstream of Nrl. Nr2e3 is also known to be downstream of Nrl [178]. Of the thirty genes that we present in Table 6.2, 15 genes have been characterized in literature, and 3 of these are known targets

of Nrl. Several of the remaining candidates are promising. Serpina3n was shown in [178] to be downregulated in the absence of Nrl. We conjecture that Serpinb8 might be a target of Nrl. Further, Cct4 and Arhgap5 are also likely to be novel targets of Nrl.

6.3.2 Nrl Targets with multiple sites

Transcription factors often work as part of larger regulatory modules where several of them are involved in regulating each gene in the module. We use the technique for rescoring using multiple sites as described in Section 6.2.2 and simultaneously search for three motifs. We use the Nrl Response Element sequence from the Pde6a binding site (TGATCCTCATRACT) [123], the sequence from the Rho binding site (TGCTGAATCAGCC) [123], and the Crx binding site (YTAATCC) [28]. The results are presented in Table 6.3. Rho is a known downstream targets of Nrl [123]. Rom1 and the calcium channel gene Trpc-1 were downregulated in the absence of Nrl [178].

6.3.3 ERα binding site

Estrogen is a steroid and is well known as the female sex hormone. Ethinyl estradiol (EE) is the common compound that is studied. Investigations into estrogen action frequently focus on the estrogen receptor transcription factor ER. Researchers in pharmacology and toxicology have particular interest in ER because of the drugs, chemicals, and natural and synthetic environmental pollutants that can raise levels of estrogen to hazardous, cancer-causing levels [93] [77]. ER itself is classified as 2 genes ERα and ERβ. ER$\alpha\beta$ is known to bind DNA either in an EE-ER complex or as a heterodimer with another DNA binding transcription factor [98]. We wish to study the targets of ER and thus have used a position weight matrix describing the estrogen responsive element (ERE) [24]. We have used this matrix and targeted the human promoter regions.

After looking at the top 30 hits found, we have found 5 target genes that have been discussed in literature. SLD5 is a component of GINS which is a heterotetramer that is regulated by ER [69]. SLC2A2, is a GLUT family member which has many members known to be augmented by estrogen [29]. ER is known to bind to actin [159]. HSPA9B, also known as mortalin, and TPD52L2 are known to be estrogen responsive [19, 82, 97, 167]. It is interesting to note that the 31^{st} hit is a sorting nexin. Sorting nexin has been shown to contain an ERE that deviates from the agreed consensus sequence for ER. However in [159], it is shown by ChIP that it is indeed responsive and binds ER. This is evidence that our method allows us to discover true targets that have differences in the consensus binding regions. Three other targets are cancer related. Table 6.4 lists the top 30 hits.

6.4 Conclusions

In this chapter, we presented the GeneFinder algorithm for predicting targets of transcription factors given the binding site signature. GeneFinder takes advantage of the position of the binding site, the phylogenetic conservation, tissue expression data, and binding site multiplicity. Previous approaches used only one or two of these approaches and were often limited to a smaller scale. We showed that GeneFinder can find several well known targets of known transcription factors such as Nrl and ERα. We also showed that we are able to offer promising predictions for novel targets.

CHAPTER VII

Conclusions

In this thesis, we described a collection of related database methods for managing and querying large sequence databases. In Chapter II, we described an algebra called PiQA that extends relational algebra to permit querying on sequence data. We showed that PiQA can be used to express complex queries on both primary and secondary structure data simultaneously thereby providing a greater expressive power than existing approaches.

In Chapter III, we outlined the usefulness of the suffix tree as an index for sequence databases. Existing algorithms are very slow at constructing suffix tree indexes for large sequences because of the high amount of random disk I/O they incur. Much of this random I/O comes from using suffix links, which are an essential mechanism used by linear time construction algorithms. We discard the use of suffix links, and adapt a top-down, worst case $O(n^2)$ algorithm (WOTD) to formulate a Top-Down Disk-based approach (TDD). TDD buffers data structures used in the WOTD algorithm and manages them carefully to reduce the time taken to construct the suffix tree by nearly an order of magnitude.

Once the input string becomes too large to fit into main memory, the performance of TDD begins to deteriorate. The WOTD algorithm accesses the input string randomly, and a large amount of random I/O is incurred regardless of the buffering policy. To address this problem, we proposed ST-Merge– a merge based algorithm that constructs suffix trees

for portions of the input string, and merges them together. We show that for cases where the input string is much larger than main memory, ST-Merge outperforms TDD and scales better.

We described the design and implementation of Periscope/SQ– our extension to PostgreSQL to support sequence queries from PiQA. Periscope uses PiQL– an extension of SQL as the query language. In Chapter IV, we describe the various challenges in building such a system. Notably, we described a new technique for estimating the selectivity of string predicates based on a novel summary structure called the Symmetric Markovian Summary. We also described new physical operators such as match-and-augment, and a simple optimization algorithm that optimized the sequence portion of the query. Using the indexes, the different physical operators, and the optimization algorithm, we show that the declarative approach of Periscope/SQ is two orders of magnitude faster than existing procedural approaches for some bioinformatics sequence processing queries.

In Chapter V, we go beyond sequence querying into an important application in sequence analysis, namely sequence mining. We describe a highly versatile substitution matrix based similarity model that captures several existing models in addition to providing the power to describe several new and useful models. We describe an algorithm called FLAME which simultaneously traverses two suffix trees to explore the space of all frequent patterns. FLAME leverages the TDD suffix tree construction algorithm. We show that FLAME is not only more versatile than existing approaches, but also extremely fast in comparison.

Finally, in Chapter VI, we demonstrate the power of the Periscope/SQ infrastructure by building an application to predict novel targets of transcription factors. GeneFinder uses multiple sources of data such as sequences, expression data, phylogenetics and combines them using the sequence querying as well as relational processing abilities of Periscope/SQ.

We show that GeneFinder not only finds several well known targets of transcription factors such as Nrl, but also offers several promising predictions of novel targets. We expect GeneFinder to be one of several applications that Periscope/SQ enables. By providing a platform for easily developing applications that need to query large amounts of sequence data, we hope that Periscope/SQ helps speed up the pace of bioinformatics research.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Mohamen Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, March 2004.

[2] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, pages 591–600, 2001.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *VLDB*, pages 487–499, 1994.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *ICDE*, pages 3–14, 1995.

[5] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[6] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[7] S. Aluru. *Suffix Trees and Suffix Arrays, Handbook of Data Structures and Applications*. CRC Press, 2004.

[8] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software: Practice and Experience*, 25(2):129–141, 1995.

[9] Alberto Apostolico and Wojciech Szpankowski. Self-alignments in words and their applications. *Journal of Algorithms*, 13(3):446–467, 1992.

[10] R. Apweiler, A. Bairoch, C. H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. Natale, A. C. O'Donovan, N. Redaschi, and L. L. Yeh. Uniprot: The universal protein knowledgebase. *Nucleic Acids Research*, 32(D):115–119, 2004.

[11] Michael Ashburner et al. Gene Ontology: Tool for the Unification of Biology. *Nature Genetics*, 25:25–29, 2000.

[12] Malcolm Atkinson and Mick Jordan. Providing orthogonal persistence for java. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 383–395, 1998.

[13] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Publishing, 4th edition, 2002.

[14] Timothy L. Bailey and Charles Elkan. Fitting a Mixture Model by Expectation Maximization to Discover Motifs in Biopolymers. In *ISMB*, pages 28–36, 1994.

[15] Timothy L. Bailey and Charles Elkan. Unsupervised Learning of Multiple Motifs in Biopolymers using EM. *Machine Learning*, 21(1-2):51–80, 1995.

[16] Srikanta J. Bedathur and Jayant R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proceedings of the 20th International Conference on Data Engineering*, pages 720–731, 2004.

[17] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[18] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and DAWGs. *Discrete Applied Mathematics*, 24(1):37–45, 1989.

[19] R. Boutros and J.A. Byrne. D53(TPD52L1) is a Cell Cycle-regulated Protein Maximally Expressed at the G2-M Transition in Breast Cancer Cells. *Experimental Cell Research*, 310:152–165, 2005.

[20] D.M. Brown and E. Ruoslahti. Metadherin, a Cell Surface Protein in Breast Tumors that Mediate Lung Metastasis. *Cancer Cell*, 5:365–374, 2004.

[21] Jeremy Buhler and Martin Tompa. Finding Motifs Using Random Projections. *Journal Computational Biology*, 9(2):225–242, 2002.

[22] Thomas N. Bulkowski. *Encyclopedia of Chart Patterns*. Wiley Trading, 2nd edition, May 2005.

[23] C. A. Orengo, A. E. Todd, and J. M. Thornton. From Protein Structure To Function. *Current Opinion in Structural Biology*, 9:374.

[24] C. Klinge. Estrogen Receptor Interaction with Estrogen Response Elements. *Nucleic Acids Research*, 29:2905.

[25] Alexandra Carvalho, Ana Freitas, Arlindo Oliveira, and Marie-France Sagot. A parallel algorithm for the extraction of structured motifs. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 147–153, 2004.

[26] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. In *ICDE*, pages 227–238, 2004.

[27] Lei Chen, M. Tamer Ozsu, and Vincent Oria. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*, pages 491–502, 2005.

[28] Shiming Chen, Qing-Liang Wang, Zuqin Nie, Hui Sun, Gregory Lennon, Neal Copeland, Debra Gilbert, Nancy Jenkins, and Donald Zack. Crx, a Novel Otx-like Paired-Homeodomain Protein Binds to and Trnasactivated Photoreceptor Cell-Specific Genes. *Neuron*, 19:1017.

[29] Clara M. Cheng, Matt Cohen, Jie Wang, and Carolyn A. Bondy. Estrogen Augments Glucose Transporter and IGF1 Expression in Primate Cerebral Cortex. *FASEB*, 15:907–915, 2001.

[30] Lok-Lam Cheng, David Cheung, and Siu-Ming Yiu. Approximate string matching in DNA sequences. In *Proceeings of the 8th International Conference on Database Systems for Advanced Applications*, pages 303–310, 2003.

[31] Ching-Fung Cheung, Jeffrey Xu Yu, and Hongjun Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.

[32] Bill Yuan-Chi Chiu, Eamonn J. Keogh, and Stefano Lonardi. Probabilistic Discovery of Time Series Motifs. In *KDD*, pages 493–498, 2003.

[33] Raphael Clifford and Marek J. Sergot. Distributed and paged suffix trees for large genetic databases. In *Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching*, pages 70–82, 2003.

[34] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory and its applications. *Algorithmica*, 32(1):1–35, 2002.

[35] cSPADE Source Code. `http://www.cs.rpi.edu/~zaki/software/`.

[36] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule Discovery From Time Series. In *KDD*, pages 16–22, 1998.

[37] S. B. Davidson. Tale of Two Cultures: Are There Database Research Issues in Bioinformatics? In *14th International Conference on Scientific and Statistical Database Management*, 2002.

[38] Margaret O. Dayhoff, R. M. Schwartz, and B.C. Orcutt. A Model for Evolutionary Changes in Proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.

[39] Deep-Shallow Suffix Array and BWT Construction Algorithms. http://www.mfn.unipmn.it/~manzini/lightweight/.

[40] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[41] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparision. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[42] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, 2005.

[43] Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A note on the height of suffix trees. *SIAM Journal of Computing*, 21(1):48–53, 1992.

[44] Digital Bibliography and Library Project (DBLP), `http://dblp.uni-trier.de/`.

[45] B. Dorohonceanu and C.G. Nevill-Manning. Accelerating Protein Classification Using Suffix Trees. *The Eighth International Conference on Intelligent Systems for Molecular Biology*, 2000.

[46] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ. Press, 1st edition, 1999.

[47] E. Segal, M. Shapira, A. Regev, D. Pe'er, D. Botstein, D. Koller, and N. Friedman. *Nature Genetics*, 34(2):166, June 2003.

[48] B. A. Eckman and A. Kaufmann. Querying BLAST within a Data Federation. *IEEE Data Engineering Bulletin*, 27(3):12–19, 2004.

[49] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377.

[50] Eleazar Eskin and Pavel A. Pevzner. Finding Composite Regulatory Patterns in DNA Sequences. In *ISMB*, pages S354–63, 2002.

[51] External Memory Suffix Array Construction Project. http://i10www.ira.uka.de/dementiev/esuffix/docu/index.html.

[52] F. Moussouni, N. W. Paton, A. Hayes, S. Oliver, C. A. Goble, and A. Brass. Database Challenges for Genome Information in the Post Sequencing Phase. In *DEXA*, 1999.

[53] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE Computer Society, 1997.

[54] Martin Farach-Colton, Paolo Ferragina, and S.Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of The ACM*, 47(6):987–1011, 2000.

[55] Ada Wai-Chee Fu, Eamonn J. Keogh, Leo Yung Hang Lau, and Chotirat (Ann) Ratanamahatana. Scaling and Time Warping in Time Series Querying. In *VLDB*, pages 649–660, 2005.

[56] G. Jaeschke and H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *PODS*, page 124, 82.

[57] GenBank, NCBI, 2004.
`www.ncbi.nlm.nih.gov/GenBank`.

[58] Genomatix. `http://www.genomatix.de`.

[59] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[60] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, 2003.

[61] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[62] Gosta Grahne, Raul Hakli, Matti Nykanen, Hellis Tamm, and Esko Ukkonen. Design and Implementation of a String Database Query Language. *Information Systems*, 28(4):311–337, 2003.

[63] Growth of GenBank, National Center for Biotechnology Information (NCBI). *www.ncbi.nlm.nih.gov/Genbank/genbankstats.html*, 2004.

[64] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[65] H. M. Berman et al. The Protein Data Bank. *Acta Crystallographica*, D58:899–907, 2002.

[66] H. M. Berman, T. Battistuz, T. N. Bhat, W. F. Bluhm, P. E. Bourne, K. Burkhardt, Z. Feng, G. L. Gilliland, L. Iype, S. Jain, P. Fagan, J. Marvin, D. Padilla, V. Ravichandran, B. Schneider, N. Thanki, H. Weissig, J. D. Westbrook, and C. Zardecki. The protein data bank. *Biological Crystallography*, 58:899.

[67] Marc S. Halfon, Yonatan Grad, George M. Church, and Alan M. Michelson. Computation-Based Discovery of Related Transcriptional Regulatory Modules and Motifs Using an Experimentally Validated Combinatorial Model. *Genome Research*, 12:1019.

[68] Joachim Hammer and Markus Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *CIDR*, 2003.

[69] R. Hayashi, T Arauchi, M Tatequ, Y Goto, and K Yoshida. A Combined Computational and Experimental Study on the Structure-regulation Relationships of Putative Mammalian DNA Replication Initiator GINS. *Genomics Proteomics Bioinformatics*, 4:156–164, August 2006.

[70] S. Henikoff and JG. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. *National Academy of Sciences, USA*, 89(22):10915–9, 1992.

[71] Klaus Heumann and Hans-Werner Mewes. The hashed position tree (HPT): A suffix tree variant for large data sets stored on slow mass storage devices. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 101–115, 1996.

[72] S. Hoppner. Discovery of Temporal Patterns – Learning Rules about the Qualitative Behaviour of Time Series. In *5th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 192–203, 2001.

[73] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A database index to large biological sequences. *The VLDB Journal*, 7(3):139–148, 2001.

[74] Data Sets from Analysis of Financial Time Series. `http://www.gsb.uchicago.edu/fac/ruey.tsay/teaching/fts/`.

[75] Intel Corporation. *The IA-32 Intel Architecture Optimization Reference Manual*. Intel (Order Number 248966), 2004.

[76] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual: System Programming Guide*, volume 3. Intel (Order Number 253668), 2004.

[77] J. Carrol et al. Genome-wide Analysis of Estrogen Receptor Binding Sites. *Nature Genetics*, 38:1289, 2006.

[78] J. L. Thorne and N. Goldman and D. T. Jones. Combining Protein Evolution and Secondary Structure. *Molecular Biology and Evolution*, 13(5):666.

[79] H. V. Jagadish, Olga Kapitskaia, Raymond Ng, and Divesh Srivastava. One-dimensional and Multi-dimensional Substring Selectivity Estimation. *The VLDB Journal*, 9(3):214–230, 2000.

[80] Joachim Hammer and Markus Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *CIDR*, 2003.

[81] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding Flexible Patterns in Unaligned Protein Sequences. *Protein Science*, 4(8):1587–1595, 1995.

[82] Sitharthan Kamalakaran, Senthil K. Radhakrishnan, and William T. Beck. Identification of Estrogen-responsive Genes Using a Genome-wide Analysis of Promoter Elements for Transcription Factor Binding Sites. *Journal of Biological Chemistry*, 205:21491–21497, 2005.

[83] Atsuhiro Kanda, James Friedman, Kaa M Nishiguchi, and Anand Swaroop. Retinopathy Mutations in the bZIP protein NRL Alter Phosphorylation and Transcriptional Activity. *Human Mutation*, 2007.

[84] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 13th International Conference on Automata, Languages and Programming*, pages 943–955, 2003.

[85] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, 2001.

[86] Manolis Kellis, Nick Patterson, Matt Endrizzi, Bruce Birren, and Eric Lander. Sequencing and Comparison of Yeast Species to Identify Genes and Regulatory Motifs. *Nature*, 423:241–254, May 2003.

[87] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 186–199, June 2003.

[88] P. Ko and S. Aluru. Space efficient linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 200–210, June 2003.

[89] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *SIGMOD*, pages 282–293, 1996.

[90] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12), 2004.

[91] Stefan Kurtz. Reducing space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[92] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29:4633–4642, 2001.

[93] John C. Kwekel et al. A Cross-species Analysis of the Rodent Uterotrophic Program: Elucidation of Conserved Responses and Targets of Estrogen Signaling. *Physiological Genomics*, 2005.

[94] L. Fegaras and David Maier. Optimizing Object Queries Using and Effective Calculus. *ACM TODS*, 25(4):457.

[95] L. Hammel and J. M. Patel. Searching on the Secondary Structure of Protein Sequences. In *VLDB*, 2002.

[96] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. In *SIGMOD*, page 273, 1989.

[97] L.Cicatiello et al. A Genomic View of Estrogen Actions in Human Breast Cancer Cells by Expression Profiling of the Hormone-responsive Transcriptome. *Journal of Molecular Endocrinology*, 32:319–775, 2004.

[98] Xiaodong Li, Jing Huang, Ping Yi, Robert A. Bambara, Russel Hilf, and Mesut Muyan. Single-chain Estrogen Receptors (ERs) Reveal that the ERa/b Heterodimer Emulates Funcations of the ERa Dimer in Genomic Estrogen Signaling Pathways. 24:7681, 2004.

[99] M. A. Roth and H. F. Korth and Adam Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM TODS*, 13(4):389.

[100] Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, pages 372–383, 2004.

[101] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[102] L. Marsan and M.-F. Sagot. Algorithms for Extracting Structured Motifs Using a Suffix Tree with Application to Promoter and Regulatory Site Consensus Identification. *Journal of Computational Biology*, 7(3/4):345–360, 2000.

[103] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of The ACM*, 23(2):262–272, 1976.

[104] Colin Meek and William P. Birmingham. The Dangers of Parsimony in Query-by-Humming Applications. In *Proc. of Int. Sym. on Music Information Retrieval*, 2003.

[105] Colin Meek, Jignesh M. Patel, and Shruti Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 910–921, 2003.

[106] Daniel P. Miranker, Weijia Xu, and Rui Mao. MoBIoS: A Metric-Space DBMS to Support Biological Discovery. In *SSDBM*, pages 241–244, 2003.

[107] C. S. Myers and L. R. Rabiner. A Comparative Study of Several Dynamic Time-Warping Algorithms for Connected Word Recognition. *The Bell System Technical Journal*, 60(7):1389–1409, 1981.

[108] G. Navarro, R. Baeza-Yates, and J. Tariho. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[109] NCBI Genomes. `ftp://ftp.ncbi.nih.gov/genomes/`.

[110] P. Seshadri, M. Livny, R. Ramakrishnan. Sequence Query Processing. In *SIGMOD*, 1994.

[111] P. Seshadri, Miron L., R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *ICDE*, 1995.

[112] Jignesh M. Patel. The role of declarative querying in bioinformatics. *OMICS: A Journal of Integrative Biology*, 7(1):89–92, 2003.

[113] Jignesh M. Patel. The Role of Declarative Querying in Bioinformatics. *OMICS: A Journal of Integrative Biology*, 7(1):89–92, 2003.

[114] Pranav Patel, Eamonn Keogh, J.Lin, and S.Lonardi. Mining Motifs in Massive Time Series Databases. In *ICDM*, pages 370–377, 2002.

[115] Giulio Pavesi, Paolo Mereghetti, Giancarlo Mauri, and Graziano Pesole. Weeder Web: Discovery of Transcription Factor Binding Sites in a Set of Sequences From Co-Regulated Genes. *Nucleic Acids Research*, 32(Web Server issue):W199–W203, 2004.

[116] Jian Pei, Jiawei Han, and Wei Wang. Mining Sequential Patterns With Constraints in Large Databases. In *CIKM*, pages 18–25, 2002.

[117] Mikael Pettersson. Perfctr: Linux performance montioring counters driver. http://user.it.uu.se/~mikpe/linux/perfctr.

[118] Pavel A. Pevzner and S.-H. Sze. Combinatorial Approaches to Finding Subtle Signals in DNA Sequences. In *ISMB*, pages 269–278, 2000.

[119] Project Gutenberg. http://www.gutenberg.net.

[120] Jian Qian, Noriko Esumi, Yangjian Chen, Qungliang Wang, Itay Chowers, and Donald J. Zack. Identification of regulatory targets of tissue-specific trascription factors: application to retina specific regulation. *Nucleic Acids Research*, 33(11):3479–3491, 2005.

[121] R. Hakli, M. Nyknen, H. Tamm, and E. Ukkonen. Implementing a Declarative String Query Language with String Restructuring. In *Practical Aspects of Declarative Languages*, 1999.

[122] Random Projections Source Code. `http://www.cse.wustl.edu/~jbuhler/pgt/`.

[123] Alnawaz Rehemtulla, Ron Warwar, Rajan Kumar, Xiaodong JiDagger, Donald J. Zack, and Anand Swaroop. The Basic Motif-leucine Zipper Transcription Factor Nrl Can Positively Regulate Rhodopsin Gene Expression. *Proceedings of the National Academy of Sciences*, 93:191–195, January 1996.

[124] S. A. Teichmann, A. G. Murzin, and C. Chothia. Determination of Protein Function, Evolution and Interactions by Structural Genomics. *Current Opinion in Structural Biology*, 11:354.

[125] S. Dong and D. Searls. Gene Structure Prediction by Linguistic Methods. *Genomics*, 1994.

[126] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[127] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[128] S. J. Thomas and P. C. Fisher. Nested Relational Structures. *Advances in Computing Research*, 3:269, 1986.

[129] S. Pittler, Y. Zhang, S. Chen, A. Mears, D. Zack,Z. Ren, P. Swain, S. Yao, A. Swaroop, J. White. Functional Analysis of the Rod Photoreceptor cGMP Phosphodiesterase Alpha-subunit Gene Promoter: Nrl and Crx are Required for Full Transcriptional Activity. *Journal of Biological Chemistry*, 279:19800–7, 2004.

[130] Schneider, T.D. and Stepehsn, R.M. Sequence Logos: A New way to Display Consensus Sequences. *Nucleic Acids Research*, 18:6097, 1990.

[131] Klaus-Bernd Schurmann and Jens Stoye. Suffix-tree construction and storage with limited main memory. Technical Report 2003-06, Univeristy of Bielefeld, Germany, 2003.

[132] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[133] Serge Abiteboul and N Bidoit. Non 1st Normal-Form Relations - an Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, 33(3):361.

[134] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *VLDB*, pages 99–110, 1996.

[135] Saurabh Sinha and Martin Tompa. YMF: A Program for Discovery of Novel Transcription Factor Binding Sites by Statistical Overrepresentation. *Nucleic Acids Research*, 31(13), 2003.

[136] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[137] Susie M. Stephens, Jake Y. Chen, and Shiby Thomas. ODM BLAST: Sequence Homology Search in the RDBMS. *IEEE Data Engineering Bulletin*, 27(3):20–23, 2004.

[138] Michael Stonebraker, Dorothy Moore, and Paul Brown. *Object Relational DBMS: Tracking the Next Great Wave*. Morgan Kauffman, 2nd edition, 1999.

[139] G.D. Stormo. DNA Binding Sites: Representation and Discovery. *Bioinformatics*, 16:16, 2000.

[140] STXXL Library. http://i10www.ira.uka.de/dementiev/stxxl.shtml.

[141] Anand Swaroop, J Xu, H Pawar, A Jackson, Cskolnick, and N Agarwal. A Conserved Retina-Specific Gene Encodes a Basic Motif/Leucine Zipper Domain. *Proceedings of the National Academy of Sciences*, 89:266–270, 1992.

[142] W. Szpankowski. *Average-Case Analysis of Algorithms on Sequences*. John Wiley and Sons, 2001.

[143] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical Suffix Tree Construction. In *VLDB*, pages 36–47, 2004.

[144] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical Suffix Tree Construction. In *VLDB*, pages 36–47, 2004.

[145] Sandeep Tata and Jignesh M. Patel. PiQA: An Algebra for Querying Protein Data Sets. In *SSDBM*, pages 141–150, 2003.

[146] The Growth of GenBank, NCBI, 2004. `www.ncbi.nlm.nih.gov/genbank/genbankstats.html`.

[147] The LDC Corpus Catalog, `http://wave.ldc.upenn.edu/Catalog/`.

[148] The MUMmer Software. http://www.tigr.org/software/mummer/.

[149] The PostgreSQL Database System. *www.postgresql.org*.

[150] W. Thompson, E. C. Rouchka, and C. E. Lawrence. Gibbs Recursive Sampler: Finding Transcription Factor Binding Sites. *Nucleic Acids Research*, 31(13):3580–3585, 2003.

[151] Y. Tian, S. Tata, R.A. Hankins, and J.M. Patel. Practical Methods for Constructing Suffix Trees. *VLDB*, 14:281–299, September 2005.

[152] Martin Tompa et al. Assessing Computational Tools for the Discovery of Transcription Factor Binding Sites. *Nature Biotechnology*, 23:137–144, 2005.

[153] TRANSFAC. `http://www.gene-regulation.com/pub/databases.html`.

[154] Ruey S. Tsay. *Analysis of Financial Time Series*. Wiley-Interscience, 1st edition, October 2001.

[155] Ajumobi Udechukwu, Ken Barker, and Reda Alhajj. Discovering all frequent trends in time series. In *Proc. of Winter Int. Sym. on Information and Comm. Tech.*, volume 58, pages 1–6, 2004.

[156] E. Ukkonen. Constructing suffix-trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture: Information Processing*, pages 484–92, 1992.

[157] Unigene. `ftp://ftp.ncbi.nih.gov/repository/UniGene/`.

[158] UniProt Knowledgebase. http://us.expasy.org/sprot.

[159] Vinsenius B. Vega, Chin-Yo Lin, Koon Siew Lai, Say Li Kong, Min Xie, Xiaodi Su, Huey Fang Teh, Jane S Thomsen, Ai Li Yeo, Wing Kin Sung, Guillaume Bourque, and Edison T Liu. Multiplatform Genome-wide Identification and Modeling of Functional Human Estrogen Receptor Binding Sites. *Genome Biology*, 7, 2006.

[160] Jeffrey S. Vitter and M. Shriver. Algorithms for parallel memory: Two-level memories. *Algorithmica*, 12:110–147, 1994.

[161] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *ICDE*, page 673684, 2002.

[162] Vlieghe D, Sandelin A, De Bleser PJ, Vleminckx K, Wasserman WW, van Roy F, and Lenhard B. A new generation of JASPAR, the Open-access Repository for Transcription Factor Binding Site Profiles. *Nucleic Acids Research*, 34(D):95–97, January 2006.

[163] Dafang Wan et al. Large-scale cDNA Transfection Screening for Genes Related to Cancer Development and Progression. *PNAS*, 101:15724–15729, 2004.

[164] Jianyong Wang and Jiawei Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*, pages 79–90, 2004.

[165] Weeder Source Code. `http://www.pesolelab.it/Tool/ind.php`.

[166] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[167] Alessandro Weisz. New Insights on Estrogen Action from Gene Expression Profiling. *Atti dei Convegni Lincei*, 211:143–153, 2005.

[168] Kyu-Young Whang, Gio Wiederhold, and Daniel Sagalowicz. Estimating block accesses in database organizations: A closed noniterative formula. *Communicatoins of the ACM*, 26(11):940–944, 1983.

[169] C. H. Wu and D. W. Nebert. Update on Human Genome Completion and Annotations: Protein Information Resource. *Human Genomics, 95760-21:35, 2004.*, 1(3):1–5, 2004.

[170] Huanmei Wu, Betty Salzberg, Gregory C Sharp, Steve B Jiang, Hiroki Shirato, and David Kaeli. Subsequence Matching on Structured Time Series Data. In *SIGMOD*, pages 682–693, 2005.

[171] T.D. Wu, C.G. Nevill-Manning, and D.L. Brutlag. Fast Probabilistic Analysis of Sequence Function using Scoring Matrices. *Bioinformatics*, 16:233–244, 2000.

[172] Xiaohui Xie, Jun Lu, EJ. Kulbokas, Todd Golub, Vamsi Mootha, Kerstin Lindblad-Toh, Eric Lander, and Manolis Kellis. Systematic Discovery of Regulatory Motifs in Human Promoters and 3' UTRs by Comparison of Several Mammals. *Nature*, February 2005.

[173] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *SDM*, 2003.

[174] Jiong Yang, Wei Wang, Philip S. Yu, and Jiawei Han. Mining Long Sequential Patterns in a Noisy Environment. In *SIGMOD*, pages 406–417, 2002.

[175] S.B. Yao. Approximating Block Accesses in Database Organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[176] YMF Source Code. `http://bio.cs.washington.edu/software.html`.

[177] Shlomo Yona and Dotan Tsadok. ANSI C implementation of a suffix tree. http://cs.haifa.ac.il/~shlomo/suffix_tree.

[178] Shigeo Yoshida, Alan J. Mears, James S. Friedman, Todd Carter, Shirley He, Edwin Oh, Yuezhou Jing, Rafal Farjo, Gilles Fleury, Carrolee Barlow, Alfred O. Hero, , and Anand Swaroop. Expression Profiling of the Developing and mature $Nrl^{-/-}$ Mouse Retina: Identification of Retinal Disease Candidates and Transcriptional Regulatory Targets of Nrl. *Human Molecular Genetics*, pages 1487–1503, May 2004.

[179] Z. M. Ozsoyoglu and J. Wang. A Keying Method for a Nested Relational Database Management System. In *ICDE*, page 438, 1992.

[180] Mohammed J Zaki. Sequence Mining in Categorical Domains: Incorporating Constrains. In *CIKM*, pages 442–429, 2000.

[181] Mohammed Javeed Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[182] Yunyue Zhu and Dennis Shasha. Warping Indexes with Envelope Transforms for Query by Humming. In *SIGMOD*, pages 181–192, 2003.