Towards a Database System for Large-scale Analytics on Strings

Dissertation by

Majed Ali Sahli

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

King Abdullah University of Science and Technology

Computer, Electrical, and Mathematical Sciences and Engineering Division

Computer Science Department

Thuwal, Makkah Province, Kingdom of Saudi Arabia

June, 2015

The dissertation of Majed Ali Sahli is approved by the examination committee.

Committee Chairperson:	Panos Kalnis, PhD, Professor
Committee Member:	Mikhail Moshkov, PhD, Professor
Committee Member:	Peter Wonka, PhD, Professor
Committee Member:	Khaled Salama, PhD, Associate Professor
External Examiner:	Themis Palpanas, PhD, Professor

ABSTRACT

Towards a Database System for Large-scale Analytics on Strings

Majed Ali Sahli

Recent technological advances are causing an explosion in the production of sequential data. Biological sequences, web logs and time series are represented as strings. Currently, strings are stored, managed and queried in an ad-hoc fashion because they lack a standardized data model and query language. String queries are computationally demanding, especially when strings are long and numerous. Existing approaches cannot handle the growing number of strings produced by environmental, healthcare, bioinformatic, and space applications. There is a trade-off between performing analytics efficiently and scaling to thousands of cores to finish in reasonable times.

In this thesis, we introduce a data model that unifies the input and output representations of core string operations. We define a declarative query language for strings where operators can be pipelined to form complex queries. A rich set of core string operators is described to support string analytics. We then demonstrate a database system for string analytics based on our model and query language. In particular, we propose the use of a novel data structure augmented by efficient parallel computation to strike a balance between preprocessing overheads and query execution times.

Next, we delve into repeated motifs extraction as a core string operation for large-scale string analytics. Motifs are frequent patterns used, for example, to identify biological functionality, periodic trends, or malicious activities. Statistical approaches are fast but inexact while combinatorial methods are sound but slow. We introduce ACME, a combinatorial repeated motifs extractor. We study the spatial and temporal locality of motif extraction and devise a cache-aware search space traversal technique. ACME is the only method that scales to gigabyte-long strings, handles large alphabets, and supports interesting motif types with minimal overhead.

While ACME is cache-efficient, it is limited by being serial. We devise a lightweight parallel space traversal technique, called FAST, that enables ACME to scale to thousands of cores. High degree of concurrency is achieved by partitioning the search space horizontally and balancing the workload among cores with minimal communication overhead. Consequently, complex queries are solved in minutes instead of days. ACME is a versatile system that runs on workstations, clusters, and supercomputers. It is the first to utilize a supercomputer and scale to 16 thousand CPUs.

Merely using more cores does not guarantee efficiency, because of the related overheads. To this end, we introduce an automatic tuning mechanism that suggests the appropriate number of cores to meet user constraints in terms of runtime while minimizing the financial cost of cloud resources. Particularly, we study workload frequency distributions then build a model that finds the best problem decomposition and estimates serial and parallel runtimes. Finally, we generalize our automatic tuning method as a general method, called APlug. APlug can be used in other applications and we integrate it with systems for molecular docking and multiple sequence alignment.

ACKNOWLEDGEMENTS

He has not thanked Allah (God)
who has not thanked people.

Prophet Muhammed
570 — 632 CE

I cannot be thankful enough to God — God created me.

I thank my parents, Ali and Hussah, for believing in me. My father passed away during the writing of this dissertation but his memory remains alive and fuels my life. I am filled with gratitude to my wife, Asma, for taking the PhD roller coaster ride with me. I find myself out of words in front of the joy and constant inspiration I receive from my children: Eyad, Alaa, and Elias.

I am grateful to Saudi Aramco for supporting my graduate-level studies. My research journey would not be as fruitful without the invaluable guidance of my advisor, Professor Panos Kalnis, and the immense assistance of my mentor and dear friend, Dr. Essam Mansour. I also thank my examination committee members for their constructive comments and insights.

I am thankful to all the members of the InfoCloud group and my friends at KAUST. Special thanks go to Razen Alharbi, Ahmad Showail, and Mohammed Alarawi for enriching my life. Finally, my thanks go to my cubicle neighbors, Fuad Jamour and Dimitrios Kleftogiannis, for baring with me over the years.

TABLE OF CONTENTS

Examination Committee Approval **2**

Copyright **3**

Abstract **4**

Acknowledgements **6**

List of Figures **10**

List of Tables **12**

List of Algorithms **14**

1 Introduction **15**

1.1 Large-scale String Analytics . . . 16

1.2 Contributions and Thesis Organization . . . 20

2 Background **24**

2.1 Related Work . . . 25

2.1.1 String Databases and Query Languges . . . 25

2.1.2 Repeated Motifs . . . 27

2.1.3 Automatic Tuning and Elasticity . . . 30

2.2 Notation . . . 33

3 A Unified Model and Query Language for Strings **34**

3.1 A Unified Model for Strings . . . 35

3.1.1 String Operations . . . 35

3.1.2 Extensibility . . . 40

3.2 A Declarative String Query Language . . . 41

3.2.1 Syntax and Semantics 41
3.2.2 Execution Plans 46
3.3 A DBMS for Large-Scale String Analytics 48
3.3.1 System Architecture 48
3.3.2 Data Structures 51
3.3.3 User-Defined Functions 57
3.3.4 Implementation 58
3.3.5 Scenarios and Workloads 60
3.4 Discussion 62

4 Cache Efficiency Applied to Motif Extraction 63
4.1 Background 64
4.1.1 Motifs 64
4.1.2 Trie-based Search Space and Suffix Trees 65
4.2 Cache-optimized Motif Extraction (CAST) 68
4.2.1 Spatial and Temporal Memory Locality 68
4.2.2 CAST Algorithm 70
4.3 Evaluation 74
4.3.1 Cache efficiency 74
4.3.2 Comparison against state-of-the-art 76

5 High Degree of Parallelism 82
5.1 Background 83
5.1.1 Parallel Computation Paradigm 83
5.2 Horizontal Search Space Partitioning 84
5.2.1 Prefix length trade-off 85
5.2.2 FAST algorithm 88
5.3 Supermaximal Motifs 88
5.3.1 Proof for Supermaximal Motifs Algorithm 91
5.4 Evaluation 93
5.4.1 Parallel scalability 93
5.4.2 Comprehensive motif extraction support 95

6 Automatic Tuning and Elasticity 100
6.1 Background 101

6.1.1 Distribution of Workload Frequency 103
6.2 Tuning ACME 108
6.2.1 Automatic Decomposition 108
6.2.2 Elastic Deployment 111
6.3 Generalization to Bag-of-Tasks Applications 112
6.3.1 Scientific Applications 114
6.3.2 The APlug Framework 116
6.3.3 API and Integration 118
6.4 Evaluation 119
6.4.1 Accuracy 121
6.4.2 Sensitivity Analysis 127
6.4.3 Effective Scalability after using APlug 129

7 Concluding Remarks **132**
7.1 Summary of Contributions 133
7.2 Future Research Directions 135

References **135**

LIST OF FIGURES

1.1 Growth of biological sequences in the GenBank 16
1.2 Example for motif extraction 18
1.3 Thesis Roadmap 23

3.1 Extraction operations applied to an example collection of strings . . 39
3.2 Abstract BNF for StarQL 42
3.3 Architecture of our large-scale string database system 49
3.4 Example generalized suffix tree 53
3.5 StarDM representation of strings in Figure 3.4 54
3.6 The meta-data of StarDM 55
3.7 Sample StarDB interface screens 59

4.1 Example DNA sequence with occurrences of a motif candidate . . . 64
4.2 The search space over a DNA sequence 66
4.3 A suffix tree index over a DNA sequence 67
4.4 Cache-efficient motif extraction example 70
4.5 Correlation between caches misses and motif extraction time 75
4.6 Serial ACME versus MADMX and VARUN 76
4.7 Serial ACME versus FLAME with variable length and size 77
4.8 Serial ACME versus FLAME with variable frequency and alphabet 78
4.9 Memory usage of serial ACME 80

5.1 Architecture of parallel ACME 83
5.2 A partitioned combinatorial trie 85
5.3 Search space coverage for a query over a DNA sequence 86
5.4 Extracting supermaximal motifs from intermediate results 90
5.5 Supermaximal motifs versus exact-length motifs 97

6.1 Problem decomposition effect on ACME speedup efficiency 102

6.2 Poor speedup efficiency in a coarse-grained decomposition 104
6.3 Optimal speedup efficiency in a moderate decomposition 106
6.4 Extremely poor speedup efficiency in a fine-grained decomposition . 107
6.5 Dataset workload effect on VinaLC speedup efficiency 113
6.6 Workload frequency distributions for 2 datasets 116
6.7 Flowchart of the APlug framework 117
6.8 C++ API foundation of APlug framework 117
6.9 Actual and estimated workload frequency distributions in P-SSW . 124
6.10 Actual and estimated workload frequency distributions in VinaLC . 124
6.11 Actual and estimated workload frequency distributions in ACME . 125

LIST OF TABLES

2.1 Comparison of previous combinatorial repeated motif extractors. . . 27

3.1 Set operations supported by StarDM 36
3.2 Matching operations supported by StarDM 37
3.3 Filtering operations supported by StarDM 37
3.4 Extraction operations supported by StarDM 38
3.5 Generation operations supported by StarDM 39
3.6 Aggregation operations supported by StarDM 40

4.1 Specifications of the systems used to evaluate CAST 74

5.1 Specifications of the systems used to evaluate FAST 93
5.2 Scalability of PSmile to 160 cores 94
5.3 Scalability of ACME to 16K cores 95
5.4 Supermaximal Motifs from the complete human DNA 96
5.5 Analysis of gigabyte-long strings of different alphabets 96
5.6 Overhead of extracting supermaximal motifs over maximal motifs . 96

6.1 Workload statistics from an example query running on 240 workers 103
6.2 Accuracy of baseline method when tasks workloads are uniform . . 122
6.3 Accuracy of APlug compared to actual parallel times of P-SSW . . 122
6.4 Accuracy of APlug compared to actual parallel times of VinaLC . . 123
6.5 Accuracy of APlug compared to actual parallel times of ACME . . 123
6.6 Accuracy of APlug decomposition of ACME on a supercomputer . . 125
6.7 Accuracy of APlug estimations of ACME on a public cloud 126
6.8 Accuracy of APlug elasticity features for VinaLC 127
6.9 Sample size sensitivity analysis of APlug with VinaLC 128
6.10 Sample size sensitivity analysis of APlug with P-SSW 128
6.11 Sample size sensitivity analysis of APlug with ACME 128

6.12 Sample size sensitivity analysis of APlug decomposition with ACME 129
6.13 APlug accurate prediction for VinaLC on a Linux cluster 130
6.14 APlug accurate prediction for P-SSW on a Linux cluster 130
6.15 APlug accurate prediction for ACME on a supercomputer 131

LIST OF ALGORITHMS

1 Cache efficient motif extraction (CAST) 71
2 Problem decomposition and scheduling (FAST) 89
3 Extraction of supermaximal motifs 90
4 Automatic Tuning 109
5 Elasticity 112

Chapter 1

Introduction

Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.

Albert Einstein
1879 — 1955 CE

In this chapter, we introduce the problem and challenges of string analytics. We then describe our solution and give an overview of the remaining chapters, including the roadmap of our thesis and a list of our contributions.

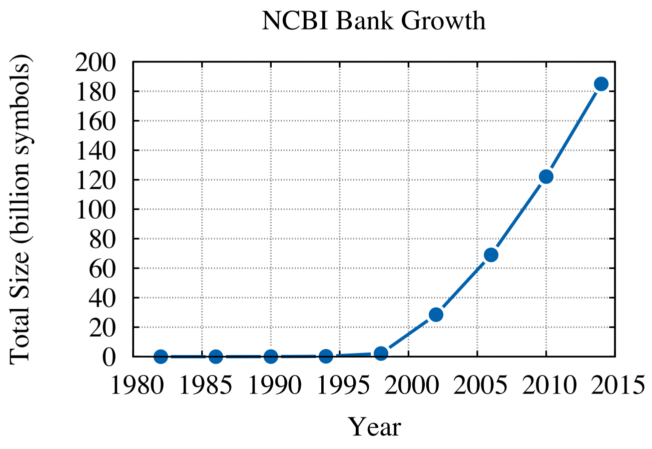


Figure 1.1: The size of the GenBank has doubled approximately every 18 months.

1.1 Large-scale String Analytics

Recent technological advances are causing an explosion in sequential data production. For instance, strings in bioinformatics are generated by high-throughput sequencing technologies [1]. According to the National Center for Biotechnology Information[1] (NCBI), the size of the genomic sequences stored in the GenBank repository has doubled approximately every 18 months as shown in Figure 1.1. Genomics can be used to find treatments and to predict how a treatment will affect a certain person. However, only the gene sections of the human genome, which constitute 5%, are discovered. Therefore, we need to analyze larger and more sequences to understand how diseases work, diagnose and treat them. Governmental and industrial bodies aim at sequencing individuals with cancer, rare diseases, and infectious diseases. Ambitious projects include the 100,000 Genomes Project[2] in the UK and the Cancer Genome Atlas[3] in the US. Similarly, string collections re-

[1]ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt
[2]http://www.genomicsengland.co.uk
[3]http://cancergenome.nih.gov

quire extensive processing in astronomy[4] and other fields [2]. For such projects to succeed, proper management and processing of the growing string datasets are of paramount importance.

String management and string analytics were long recognized as critical for applications that query genetic sequences, process images, or mine textual content [3]. Since operations on strings require accessing their internal structure, one could map a string to a relation and its symbols to attributes. However, the order of string elements matters and the number of symbols is usually large and not fixed. Several attempts were proposed in the last two decades to handle strings in database systems differently. Some Works extened the relational calculus [4, 5, 6, 7]. Periscope/SQ [8] extended PostgreSQL with matching operations over biological sequences. But it is challenging to express common string operations, such as motifs and k-mers, using matching only. To handle strings as first-class types, researchers moved to object-oriented databases [9, 10]. However, simple extensions are limited by their original data models and require the modification of mature systems. Unfortunately, there is no standard mechanism for managing and analyzing strings. Therefore, string analytics are currently performed by running multiple disjoint applications.

For example, BLAST [11] is used for matching; it finds regions of local similarity between biological sequences. Another example is KAT[5], a k-mer counting tool used to analyze substring frequency spectra. Users need to move data between applications that use different formats and have different requirements in order to draw conclusions. This gave rise to string analysis pipeline systems, such as SeqWare Pipeline[6]. In these systems, users need to define the steps and order of

[4]http://www.skatelescope.org
[5]http://www.tgac.ac.uk/KAT/
[6]https://seqware.github.io/docs/6-pipeline/

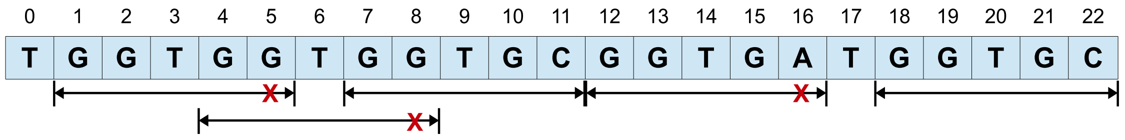


Figure 1.2: Example string S over DNA alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Matches for `GGTGC` are indicated, allowing one mismatch and overlapping matches.

execution required to analyze the underlying strings. However, current datasets are too large or too long for existing methodologies to manage them and process them efficiently. Inspired by the well-established and successful relational database management systems, we envision a database management system for strings. Such system needs to natively represent strings and support their operations while allowing users to declaratively query their data.

There are many complex string operations, like extracting frequent pattern, called motifs. Motif extraction is a core string operation used to identify biological functionality in genomic sequences, periodicity in time series, or user activity trends in web logs. A repeated motif is a pattern that approximately appears many times in a single string. For example, Figure 1.2 shows an example DNA sequence. Consider a motif extraction query where motifs are of length 5 and we require at least 5 approximate matches for a pattern to be a valid motif. Assume we allow up to one mismatch between a motif candidate and its matches. To find the indicated matches in the figure, we need to scan the sequence once.

However, to find all repeated motifs, an accurate motif extractor faces a search space that is exponential with respect to the alphabet size. In this example, the number of motif candidates is 1,024 (i.e., 4^5). If the sequence was from a protein, there would be $20^5 = 160{,}000$ motif candidates. It is indeed challenging to efficiently extract all motifs. Hence, the classic apriori algorithm is used to stop short and full-text indexes are used to reduce scanning. Unfortunately, this introduces random data access patterns and significantly affects performance [12]. Existing combinatorial motif extractors, such as FLAME [13] and VARUN [14], are serial and very slow. Consequently, they are limited to very short strings (i.e., a few megabytes), small alphabets (typically 4 symbols for DNA sequences), and restricted types of motifs.

In addition, the search space of a motif extraction query is pruned at different depths depending on the query parameters and the underlying string. Parallelization is challenging because it is difficult to decompose the search space and maintain a balanced workload among many cores. In fact, the only existing parallel repeated motif extractor, PSmile [15], reported scaling to 4 cores only to extract motifs from a sequence of size 0.2MB. Their scalability is limited by their assumption of equal workload per search space partition. Obviously, high efficiency and massive parallelization are key for extracting motifs and enabling string analytics in general.

To sum up, traditional data management methods were designed and implemented to deal with challenges different than those of data-intensive applications. The challenges that face string management and large-scale analytic systems are manifold. (*i*) First, the representation of string collections is not straightforward. For example, strings can be represented as a relation or as a set of arrays, which affects how strings are stored and accessed. (*ii*) Nevertheless, it is key for string operations to be efficient in using the memory hierarchy because they are complex and repetitive. (*iii*) Moreover, a single machine cannot handle large-scale string analytics; so operations need to run in distributed environments. Given the unknown and variable workloads of string operations, scalability is challenging. (*iv*) Additionally, it is critical when using large infrastructures to work within a budget, such as time or money.

1.2 Contributions and Thesis Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we survey the related work in the area and introduce the string notation used through this dissertation. Subsequently, we introduce our specific contributions:

- In Chapter 3, we propose an architecture of a string database system for large-scale string analytics. Our architecture hides the complexity of the underlying infrastructure in order to enable efficient processing with minimal user intervention. Particularly, we introduce a novel data model and describe a declarative query language to separate data from systems implementations. We lay down the building blocks of a distributed string database system. We demonstrate how by using our proposed architecture, what used to be cumbersome separate tasks could be achieved by dealing with a single, possibly distributed, system.

  For example, using our string database system, a biologist would store all genomic sequences in a single repository and perform analytics by submitting the following declarative queries: `SELECT RMOTIFS FROM dna WHERE LEN=9; SELECT KMERS FROM dna WHERE LEN=3;`. Currently, this is not possible with any existing system.

- In Chapter 4, we introduce ACME, a combinatorial repeated motifs extractor. We exploit the memory hierarchy to address the challenge of random data access patterns in string operations. Particularly, we study the spacial and temporal locality of motif extraction and devise a cache aware search space traversal technique. We logically consider the search space tree as a set of branches. Each branch contains symbols that are conceptually adjacent with preserved order; allowing for spatial locality. Moreover, maintaining occurrences sets is a pipelined process; for instance, the occurrences set of `AA` is used to build the occurrences set of `AAA`, leading to temporal locality.

ACME is the only method that scales to gigabyte-long strings, handles large alphabets, and supports interesting motif types with minimal overhead. Cache efficiency allows ACME to achieve an order of magnitude performance gain in serial execution compared to the state-of-the-art accurate motif extractors: FLAME [13], VARUN [14], and MADMX [16]. However, scaling to larger strings and supporting more interesting motif types, such as supermaximal ones, require scaling out to finish in reasonable times.

- In Chapter 5, we extend ACME with an efficient parallel space traversal technique which enables us to scale to thousands of cores. High degree of concurrency is achieved by partitioning the search space horizontally and balancing the workload among cores with minimal communication overhead. We decompose the search space into a large number of independent sub-tries. Our target is to provide enough sub-tries per CPU to utilize all computing resources with minimal idle time.

  The only parallel accurate motif extractor, PSmile [15], reported scaling to 4 cores only; due to excessive load imbalance. Using our methodology, ACME solves complex queries in minutes using a supercomputer instead of days on a workstation. Nevertheless, merely using more cores does not guarantee efficiency, because of overheads such as duplicate work and communication.

- In Chapter 6, we introduce our automatic tuning mechanism that suggests the appropriate number of cores to meet the user constraints in terms of run time, while minimizing the financial cost of cloud resources. Particularly, we study workload frequency distributions then build a model that finds the best problem decomposition and estimates serial and parallel runtimes. Using a sample of tasks runtimes, our framework approximates the workload and uses a discrete event simulator to calculate the optimal decomposition parameter and accurately estimate serial and parallel runtimes. The decomposition is to be fine to avoid heavy tasks that affect balancing the work across many cores. However, the decomposition must not be too fine because the parallel overheads then increase drastically. Given a speedup efficiency threshold, the goal of our automatic tuning method is to maximize parallelism while maintaining efficiency.

Finally, we generalize our automatic tuning method as a standalone framework, called APlug. Our automatic tuning framework can be integrated in bag-of-tasks applications or used as a standalone tool to guide user decisions. We use APlug to estimate serial and parallel runtimes of a molecular docking application and a sequence alignment tool. This allows us to predict and optimize speedup efficiency and choose the right number of cores according to the data workload, which is not known in advance.

This dissertation contains published work and work to be submitted. Specifically, Chapter 3 describes our system accepted for demonstration in the Very Large Data Bases Conference (VLDB 2015) [17] and a paper of our declarative string query language, to be submitted to the Very Large Data Bases Conference (VLDB 2016). Chapter 4 is based on our paper published in the proceedings of the ACM International Conference on Information and Knowledge Management (CIKM 2013) [18]. The work in Chapter 5 is published in the Very Large Data Bases Journal (December 2014, VLDBJ) [19]. Our automatic tuning framework in Chapter 6 is part of the proceedings of the IEEE International Conference on Data Engineering (ICDE 2015) [20]. Figure 1.3 shows the roadmap.

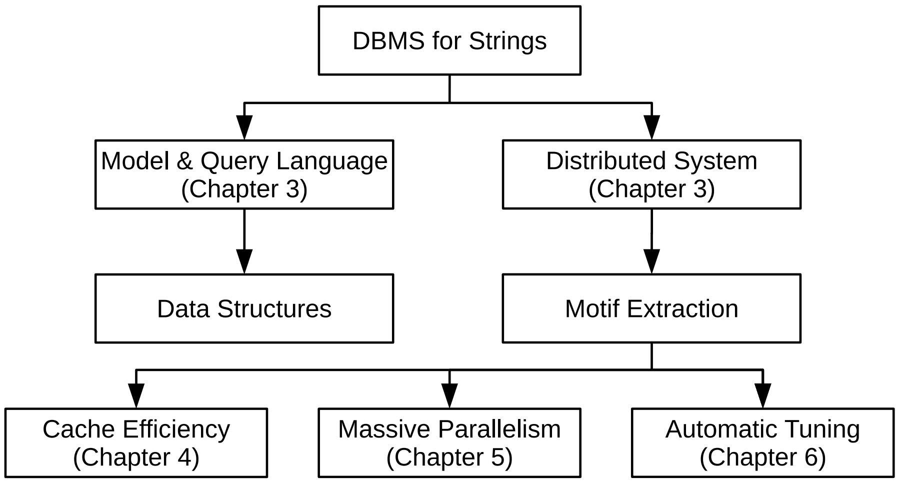

DBMS for Strings
Model & Query Language (Chapter 3)
Distributed System (Chapter 3)
Data Structures
Motif Extraction
Cache Efficiency (Chapter 4)
Massive Parallelism (Chapter 5)
Automatic Tuning (Chapter 6)

Figure 1.3: The roadmap of this thesis.

In a nutshell, our contributions are:

- A scalable data model and a declarative query language for strings.
- An architecture of a database system for large-scale string analytics.
- A cache efficient search space traversal method applied to motif extraction.
- A scalable problem decomposition technique applied to motif extraction.
- An automatic tuning and elasticity framework applied to motif extraction and generalized to bag-of-tasks applications.

Chapter 2

Background

We often miss opportunity
because it's dressed in overalls
and looks like work.

Thomas A. Edison
1847 — 1931 CE

This chapter first presents previous work related to string databases and query languages. Following our thesis roadmap, we discuss serial and parallel repeated motif extraction methods then present areas and works related to automatic tuning and elasticity. Finally, the notation used through this dissertation is introduced.

2.1 Related Work

2.1.1 String Databases and Query Languges

Traditional databases do not provide adequate support for strings and their operations. Present-day data management methods were designed and implemented to deal with challenges different than those of large-scale string analytics. To handle strings in database systems differently, several attempts were proposed in the last two decades. They were either mere extensions or novel approaches.

Relational databases can be used to store strings in tables using columns of text or binary large object (blob) data types. However, relational databases deal with strings as atomic entities and queries over their internal structure are limited to the `LIKE` construct in the de facto query language, SQL. Simple extensions build on the rich and well-established data management literature and systems by introducing strings as relational domains [21]. Works of this type include extensions to the relational calculus [4, 5, 6, 7]. Periscope/SQ [8] extended PostgreSQL with matching operations over biological sequences. It is challenging to express common string queries, such as motifs and k-mers, with only matching operations. Moreover, complex queries require the efficient utilization of large infrastructures to finish in reasonable times. Therefore, Periscope/SQ reported simple matching queries over sequences of 5,000 symbols.

Most relational databases support a limited number of data types. To handle strings as first-class types, researchers moved to object-oriented databases [9, 10]. Nevertheless, support of string operations in object-oriented databases does not provide an ultimate solution as meta data overhead grows. Generally, extentions of existing databases are limited by their original data models and require the modification of mature systems [22].

Attempts for native string support in databases exist, but most cases take an application-specific approach. SRS is an information indexing and retrieval system [23]. It targets flat files and makes use of the internal structure of their formats. SRS only allows users to draw links between different files using atomic non-sequence fields [24]. For example, SRS users can only query the description fields of FASTA-formatted and EMBL-formatted nucleotide and peptide sequences. SEQ is a string database system based on distinct domains for string elements and their underlying order type [25]. This is beneficial if users need to compute moving averages on time series. However, SEQ model is limits parsing tasks, such as matching.

There is no standard mechanism for managing and analyzing strings at the scale needed nowadays. Algorithms that handle long strings or large collections of strings are implemented within disjoint applications. For example, BLAST [11] is used for matching; it finds regions of local similarity between biological sequences. Another example is KAT[1], a k-mer counting tool used to analyze substring frequency spectra. Currently, string analytics require running multiple standalone applications. Users need to move data between applications that use different formats and have different requirements in order to draw conclusions. This gave rise to string analysis pipeline systems, such as SeqWare Pipeline[2], where users need to define the steps and order of execution manually.

Several string data models and languages are theoritically sound but impractical to implement [9]. Richardsons introduced one of the early declarative query language for strings [26]. In his model, a string starts with a symbol and the every symbol is considered the next instance of the symbol to its left. However, it is known that temporal logic modalities have limited support for recursion or iteration [27], both needed for string queries.

[1]http://www.tgac.ac.uk/KAT/
[2]https://seqware.github.io/docs/6-pipeline/

Table 2.1: Comparison of previous combinatorial repeated motif extractors.

Method	Largest reported input	Supported motif types	Execution
FLAME [13]	1.3 MB	Exact-length	Serial
VARUN [14]	3.1 MB	Maximal	Serial
MADMX [16]	0.5 MB	Maximal	Serial
PSmile [15]	0.2 MB	Exact-length	Parallel

2.1.2 Repeated Motifs

In this thesis, we use the repeated motifs problem as a string problem that faces the challenges of a string database system. This section presents the most recent methods for extracting motifs from a single string (i.e., combinatorial repeated motifs). Table 2.1 shows a summary.

Serial Methods

Motif extraction is a highly repetitive process making it directly affected by cache alignment and memory access patterns. For a motif extractor to be scalable, it needs to utilize the memory hierarchy efficiently and run in parallel. Existing methods do not deal with these issues. Therefore, they are limited to sequences in the order of a few megabytes [28].

The complexity of motif extraction grows exponentially with the motif length. Intuitively, extracting maximal and supermaximal motifs is more complex than exact-length ones because, if length is known, the search space can be pruned significantly. FLAME [13] supports only exact-length motifs. To explore a sequence, users need to run multiple exact-length queries. VARUN [14] and MADMX [16], on the other hand, support maximal motifs, without any length restriction. To limit the search space, VARUN and MADMX assume that the distance threshold varies with respect to the length of the current motif candidate. None of these techniques supports supermaximal motifs; therefore their output contains a lot of redundant motifs (i.e., motifs that are subsequences of longer ones). Despite these restrictions, the length of the largest reported input was only a few megabytes. It must be mentioned that none of these methods is parallel.

In Chapter 4, we introduce CAST, a cache-aware method for solving the combinatorial repeated motifs problem. CAST arranges the search space, the index, and intermediate results in continuous memory blocks, and accesses it in a way that exhibits spatial and temporal locality, thus minimizing cache misses. As a result, CAST improves performance of serial execution by an order of magnitude. CAST allows for extracting right-supermaximal motifs, which are motifs that are not prefixes of any other; eliminating redundant motifs that existing methods report.

Parallel Methods

Parallelizing motif extraction attracted a lot of research efforts, especially in bioinformatics [15, 29, 30, 31, 32, 33]. Challa and Thulasiraman [29] handle a dataset of 15,000 protein sequences with the longest sequence being 577 symbols only; their method does not manage to scale to more than 64 cores. Dasari et al. [30] extract common motifs from 20 strings of a total size of 12KB and scale to 16 cores. This work has been extended [31] to support GPUs and scaled to 4 GPU devices using the same dataset. Liu et al. [32] process a 1MB dataset on 8 GPUs. DMF, an implementation of Huang's work [34], has been parallelized by Marchand et al. [33] to run on a supercomputer. The aforementioned methods target the much simpler *common* motifs problem (i.e., they assume a dataset of many short strings), whereas we solve the *repeated* motifs problem [35] (i.e., one very long string). Moreover, most of these approaches are statistical. Therefore they may introduce false positive or false negative results, whereas we focus on the combinatorial case that guarantees correctness and completeness.

To the best of our knowledge, the only parallel and combinatorial method for extracting motifs from a single sequence is PSmile [15]. This method parallelizes SMILE [36], an existing serial algorithm that extracts structured motifs, composed of several "boxes" separated by "spacers" of different lengths. Intuitively, this corresponds to a distance function that allows gaps. SMILE (and PSmile) can support the Hamming distance by using only one box and zero-length spacers. PSmile partitions the search space using prefixes. Their contribution is the heuristic that groups search space partitions to balance workload among cores. Prefixes are grouped into coarse-grained tasks, and workload of different prefixes is assumed to be similar. Based on this assumption, a static scheduling scheme is used to distribute the tasks. However, their assumption is not satisfied in real datasets; therefore, in practice PSmile suffers from highly imbalanced workload and parallel overhead [37]. PSmile reported scaling to 4 cores only; the maximum input size was 0.2MB.

In Chapter 5, we introduce the first parallel method that scales to thousands of cores and gigabyte long strings. The most important issue for good scalability is to achieve load balance by having a good decomposition of the search space. Our approach scales to 16,386 cores and can support gigabyte long strings. We also propose an algorithm to extract supermaximal motifs with minimal overhead. Straight-forward calculation would require keeping track of all maximal motifs; this is too expensive, therefore no previous approach supports such motifs.

Motifs in time series

The notion of motifs exists in applications that use time series. As in the case of motifs in strings, most methods for extracting motifs in time series are approximate solutions [38, 39, 40]. Nevertheless, serial [41] and parallel [42] exact solutions exist, too. However, solutions for extracting motifs from time series cannot be used to extract motifs from strings for two main reasons. (*i*) The definition of motifs in strings is different that that in time series. In time series, motifs are pairs of similar time series subsequences. Without considering the difference in frequency requirement, motifs in time series have at least one exact match while motifs in strings may not appear exactly at all. In other words, the search space for motifs in time series is smaller than that of motifs in strings. (*ii*) The domain of time series is continuous whereas it is discrete for strings. If a time series is discretized, algorithms for extracting motifs in strings are not guaranteed to find the motifs in the original real-valued time series [43].

2.1.3 Automatic Tuning and Elasticity

Automatic tuning is needed for applications that handle big data and scale out on large infrastructures. The de facto application is data analytics using MapReduce or MPI-based systems. In MapReduce frameworks, data analytics is accomplished using multiple iterations. PREDIcT [44] introduced an experimental methodology for estimating the number of iterations and the time of each iteration in iterative analytics. PREDIcT does not estimate the number of machines required to meet user-specific constraints or suggest the best decomposition into a certain number of iterations. Both problems are challenging due to variance in iteration times. We address these problems for bag-of-tasks MPI-based applications, where the variance among tasks is even higher. Starfish [45] automatically tunes a Hadoop cluster to enhance its performance for data analytics. It uses dynamic instrumentation to profile jobs and a what-if engine to predict performance. Cumulon [46] is aimed specifically at statistical analysis on public clouds. In this thesis, we target different large-scale infrastructures, such as supercomputers, and suggest resources based on workload for general bag-of-tasks applications.

Resource allocation and scheduling are to provision resources and balance the load on them. Middleware solutions, such as Falkon [47], were used to manage resources for applications with many-tasks. Recent work provided middleware solutions with cost-efficient algorithms for tasks assignment across multiple clouds [48]. Middleware solutions are infrastructure and provider specific. Applications require code modification to utilize such services. To deal with trailing tasks, the number of workers can be shrunk at some point to keep utilization high [49]. BaTS [50] replicates trailing tasks on idle cores to increase the chances of completing them faster. Resource allocation is done by service providers and scheduling is done either by applications or middleware. We consider these problems from a user point of view. Our work is orthogonal to resource allocation and scheduling.

Runtime and scalability estimation are used to assist the decision making processes of schedulers and resource management modules [51]. One way of estimating task runtime is to analyze and profile code [52]. Compiler knowledge can be used to extract performance metrics [53]. Full code needs to be analyzed in order to create and validate scalability predictive models. The inputs for such models are low level system metrics, such as communication latencies, data type sizes, and memory contention. A more practical approach is to use statistical modeling to estimate runtime and scalability [54]. Bayesian and neural networks were used in grids to learn the execution times of running tasks [55]. Nevertheless, existing execution time estimators for large-scale applications were shown to be ineffective in clouds [56]. Our approach is less aggressive and requires no or minimal coding. We model the workload distribution of tasks using a sample run. Our model does not require code profiling and is not infrastructure specific.

Performance modeling is a useful tool to optimize existing systems and design future ones [57]. Works on performance prediction are mostly specific to certain applications and architectures requiring code inspection and instrumentation [58]. Regression techniques were used to estimate the performance of scientific workflows in heterogeneous environments [59]. Recently, a framework for predicting application performance on systems with hardware accelerators was introduced [60]. Our goal is to meet user constraints for a specific run of a bag-of-tasks application. We implicitly capture hardware performance in order to automatically tune the degree of parallelism and problem decomposition.

We are not concerned with allocation and scheduling strategies at the infrastructure level. We do not measure nor optimize performance of applications or computing infrastructures. In Chapter 6, we develop an automatic tuning process for the partitioning of the search space in order to scale efficiently to thousands of cores. For each query and dataset, we gather statistics and build an execution model. We then run simulations to decide the best partitioning that maximizes the efficient utilization of available CPUs. In conjunction with a cloud provider's pricing scheme, our auto-tuning method can also be used to minimize the financial cost of deployment on the cloud, while meeting the user constraints in terms of performance. We then generalize our approach to applications of many independent tasks to predict the appropriate degree of parallelism for a particular process based on user constraints with minimal overhead.

2.2 Notation

Let Σ be an input alphabet, a finite set of elements. The elements of Σ are called letters, characters, or symbols. A typical example of alphabets is the English alphabet of 26 letters. In biology, the DNA alphabet consists of the four characters $\{A, C, G, T\}$. A string $\mathtt{S}$ over Σ is a finite concatenation (ordered sequence) of symbols taken from Σ. The length, or size, of a string $\mathtt{S}{=}a_0a_1 \ldots a_{n-1}$, denoted by $|\mathtt{S}|$ is n, the number of its elements from the alphabet with repetitions. So, the length of $\mathtt{S} = ACAAT$ is 5. Extending these concepts to a collection of strings, the size is the number of strings in the collection, the length is the summation of their lengths, and the alphabet is the union of their alphabets.

The i-th element of a string $\mathtt{S}$ of length n is denoted by $\mathtt{S}[i]$ and i is its position in $\mathtt{S}$, where $0 \leq i < n$. We will assume that $\mathtt{S}[n]$ contains the special end character $\$ \notin \Sigma$, which cannot occur anywhere but in the end of a string. We denote by $\mathtt{S}[i, j]$ a substring of $\mathtt{S}$ defined as follows.

$$\mathtt{S}[i, j] = \begin{cases} \mathtt{S}[i]\mathtt{S}[i+1] \ldots \mathtt{S}[j] \text{ of length } j - i + 1 & j > i \\ \mathtt{S}[i] \text{ of length } 1 & j = i \\ \$ \text{ of length zero} & j < i \end{cases}$$

Given a string $\mathtt{S}$ and the three strings (possibly empty) $\mathtt{p}$, $\mathtt{s}$, and $\mathtt{t}$ such that $\mathtt{S} = \mathtt{pst}$; $\mathtt{p}$ is a *prefix* of $\mathtt{S}$, $\mathtt{s}$ is a *substring* of $\mathtt{S}$, and $\mathtt{t}$ is a *suffix* of $\mathtt{S}$. We say that the string $\mathtt{s}$ of length m occurs in $\mathtt{S}$ at position i, or that the position i is a match for $\mathtt{s}$ in $\mathtt{S}$, if $\mathtt{s} = \mathtt{S}[i, i{+}m{-}1]$, or $\mathtt{s}$ is similar to $\mathtt{S}[i, i{+}m{-}1]$ for some integer i given a similarity measure and threshold.

Chapter 3

A Unified Model and Query Language for Strings

The menu is not the meal.

Alan W. Watts
1915 — 1973 CE

We propose the use of a unified data model for strings. The model supports a wide range of core string operations, unifying the representation of their inputs and outputs. We define a declarative query language for strings where operators can be pipelined to easily form complex string queries. We demonstrate a string database system based on our unified model and query language. Because large-scale string analytics are too intricate to be solved on one machine, we implement a distributed system where parallel computation is used to augment the simplicity of our architecture.

3.1 A Unified Model for Strings

Operations on strings can have one or more strings as input and produce one or more strings as output. Therefore, a query language for strings will have to deal with sets of strings. Theoretically, a set is an ideal representation because it allows no duplicate entries and can have (i) no strings (the empty set), (ii) one string (a singleton), or (iii) many different strings.

We introduce StarDM, a string model that unifies different string operations. In StarDM, we consider strings as sequences of symbols, grouped into collections. A string of zero symbols is an empty string, and a collection of zero non-empty strings is an empty collection. Depending on how string collections are generated, they could consist of several long strings or millions of short strings. We aim to provide a declarative and abstract method for specifying strings and queries over them. We define operations that natively support string queries based on the set theory and that a string is a sequence of symbols. Operations in StarDM take one or two collections as input and output a collection. This unification of format, along with relevant meta-data allow for expressing and solving complex string queries. The simplicity of our model, StarDM, allows for efficient and scalable implementations, as we show in our demonstration.

3.1.1 String Operations

We collected and extended string algebra operations suggested by different works [61, 62]. Previous works focus on extending relational calculus to support strings [63, 64, 9]. This limits the applicability of string operations by the underlying relational model. Works that do not extend the relational model are constrained by either being application-specific [65] or by providing very basic operations that

Table 3.1: Set operations supported by StarDM, where A, B, and C are collections of strings.

Operator	**Procedural form**	**Algebraic form**
Intersection	C = intersection(A, B)	$A \cap B$
Union	C = union(A, B)	$A \cup B$
Difference	C = difference(A, B)	$A \setminus B$

make them theoretically sound but not practical [66]. The following operations are chosen to support most string queries required in different applications, such as matching and frequent patterns. They are not too specific, as genomic assembly, nor too generic, as simple concatenation. Queries can be composed using one or more of these operations. Nevertheless, the following set of operations is not exhaustive and is extensible.

Set operations

The set operations on string collections are identical to their counterparts in the set theory (see Table 3.1). Given A and B two sets of strings, the intersection operation outputs the set of strings common to A and B. Similarly, the union operation results in the set of strings in A or B and the difference operation filters B out of A. For example, if $A = \{s_1, s_2, s_3\}$ and $B = \{s_2, s_3, s_4\}$ then $A \cap B = \{s_2, s_3\}$; $A \cup B = \{s_1, s_2, s_3, s_4\}$; and $A \setminus B = \{s_1\}$.

Matching operations

Matching is the basic textual problem and one of the most used string operations. An exact matching operator may return a boolean value indicating the existence of a match or it may return the number of matches, if any. A more involved requirement is to return the positions of the matches in the searched string. Variations of this operator include approximate matching and evaluating regular expressions.

Table 3.2: Matching operations supported by StarDM, where A, B, and C are collections of strings; t is a binary for *all* or *any* and d is a distance metric threshold.

Operator	**Procedural form**	**Algebraic form**
Exact	C = exact(A, B, t)	A E B
Approximate	C = approx(A, B, t, d)	A $A_{all\|any,d}$ B
Regex	C = regex(A, B, t)	A $R_{all\|any}$ B

Table 3.3: Filtering operations supported by StarDM, where A, B, and C are collections of strings, and d is a distance metric threshold.

Operator	**Procedural form**	**Algebraic form**
Prefix Filter	C = is-prefix(A, B, d)	A P_d B
Suffix Filter	C = is-suffix(A, B, d)	A S_d B
Substring Filter	C = contains(A, B, d)	A C_d B

In Table 3.2, given A and B two sets of strings, the exact match operation outputs the set of substrings from A that match a string in B. Similarly, the approximate match operation finds matches of B in A given a distance metric threshold. For most flexible matching, regular expressions are used. In this case, B is a set of regular expressions and the operation outputs the set of substrings from A that match expressions in B. The *all* and *any* variants of the operators dectate the semantics over the two sets. In particular, an *all* match requires matching against every string in B, whereas an *any* match requires at least one string in B that matches. Specifically, matching over two string collections is logically an operation over the Cartesian product of these collections. For example, if A = {TEST, BEST, ESTATE} and B = {TT} then A E B = ∅; A $A_{ham(1)}$ B = {TE, ST, AT}; and A R {T*T} = {TEST, TAT}.

Table 3.4: Extraction operations supported by StarDM, where A and C are collections of strings, and $p_1 \leq p_2$ are integers; lengths in the case of prefixes and suffixes, and positions in the range case.

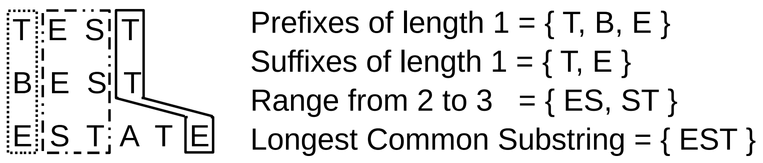
Operator	**Procedural form**	**Algebraic form**
Extract Prefixes	C = prefixes(A, p_1, p_2)	$\prod_{(START,[p_1,p_2])}$ A
Extract Suffixes	C = suffixes(A, p_1, p_2)	$\prod_{(END-[p_1,p_2],END)}$ A
Extract Range	C = range(A, p_1, p_2)	$\prod_{(p_1,p_2)}$ A
Extract LCS	C = lcs(A)	$\bigwedge$ A

Filtering operations

Filtering a collection of strings can be thought of as a form of set difference operation with a condition. In Table 3.3, given A and B two sets of strings, we filter the left operand according to conditions that involve the right operand. The prefix and suffix filter operations output the set of strings in A that have a prefix or a suffix from B within a distance. The substring filter is more generic, where A is filtered by matching strings from B regardless of position. Given two string collections, filters are applied on the Cartesian product of these collections. Moreover, sets can be filtered according to metadata properties, such as length, size, and alphabet. For example, if A = {TEST, BEST, ESTATE} and B = {TT} then A $P_{ham(1)}$ B = {TEST}; A $S_{ham(1)}$ B = {TEST, BEST, ESTATE}; and A $C_{ham(0)}$ B = $\emptyset$.

Extraction operations

Extraction operations allow for processing prefixes, suffixes, or ranges within a collection of strings as shown in Table 3.4. Given a set of strings A, we allow extracting prefixes with a range of length. If a string is shorter than p_1, it is not considered a valid prefix and thus eliminated altogether. Suffixes are handled in the same manner without the need to know the length of strings beforehand. The range extraction operation takes absolute positions and extracts the substrings. If


T E S T
B E S T
E S T A T E
Prefixes of length 1 = { T, B, E }
Suffixes of length 1 = { T, E }
Range from 2 to 3 = { ES, ST }
Longest Common Substring = { EST }

(a) Strings collection (b) Extraction operations results

Figure 3.1: Extraction operations applied to an example collection of strings and the StarDM representation of the extracted suffixes showing counts.

Table 3.5: Generation operations supported by StarDM, where A and C are collections of strings, $l_1 \leq l_2$ are lengths, f is a frequency threshold, and d is a distance metrics threshold.

Operator	**Procedural form**	**Algebraic form**
Common Motifs	C = cmotifs(A, l_1, l_2, f, d)	$\bigotimes_{l_1,l_2,f,d}$ A
Repeated Motifs	C = rmotifs(A, l_1, l_2, f, d)	$\bigoplus_{l_1,l_2,f,d}$ A
K-mers	C = kmers(A, l_1)	$\bigodot_{l_1}$ A

a string is shorter than p_2, no substring would be extracted. The longest common substring operation (LCS) outputs a collection of a single string, the longest common substring within the input collection.

Figure 3.1 shows an example of extraction operations over a sample collection. Notice that if we were to extract prefixes (or suffixes) of length 5 from the example strings then there would be only one. Moreover, in the example, there are 2 unique suffixes of length 1, but StarDM keeps track of the total number of suffixes. This will be detailed when the data structures are discussed in Section 3.3.2. In this case, the “T” will have a count of 2 because there were 2 “T” suffixes. The range example has a similar case.

Generation operations

The common motifs problem is to find frequent patterns that appear in a number of strings. A similar but more challenging problem is the repeated motifs problem,

Table 3.6: Aggregation operations supported by StarDM, where A is a collection of strings and I is an integer result.

Operator	**Procedural form**	**Algebraic form**
Length	I = length(A)	$\coprod$ A
Size	I = size(A)	$\|A\|$

where a pattern is frequent if it appears a certain number of times in a single string. In Table 3.5, we defined operators for common and repeated motifs. Given A a set of strings, the common motifs operation outputs the set of strings of length l_1 to l_2 that have matches in at least f strings from A. For the repeated motifs operation, the output is the union of repeated motifs in each string in A. Another operation used in many applications is generating the set of k-mers, substrings of length k. Our k-mer operation finds the set of strings of a certain length that appear exactly in at least one string from A.

Aggregation operations

For strings, it is important to know the size of a string collection and the length of strings in a collection. Table 3.6 shows the two aggregate string operations. Aggregate operations are deterministic and result in a single value. The length of a collection is the summation of the its member string lengths. The size is the number of these nonempty strings. Operations such as average, sum, maximum, and minimum can operate on outputs of length and size.

3.1.2 Extensibility

Given StarDM, any operator that takes one or more collections as input and outputs a collection is a valid operator. The list of operations discussed in Section 3.1.1 is not exhaustive. The four categories of string operations can be extended with new operations that have application specific logic. For example, in bioinformatics, genome assembly is an operation that takes a collection of short reads and outputs a collection of a single long sequence. Assembly can be added to the existing string generation operations. User-defined functions are incorporated through dynamic loading (e.g., by linking a shared library).

Another way to extend the functionality of StarDM operations is to define functions that are used within existing operators to embed application logic. For example, Table 3.2 shows that matching operations take as input a distance metric function. This function takes as input two strings and returns a scalar value indicating the disimilarity between the input strings. Typical distance functions include the Hamming and Edit distance metrics. However, in bioinformatics, users may require the use of application-specific metrics, such as weighted matrices for DNA similarity scoring.

3.2 A Declarative String Query Language

We introduce StarQL, a declarative query language based on our strings data model. Queries in StarQL are categorized according to their applicability and results to administrative and analytic queries. Administrative queries are used to manage the database and its string collections. Analytic queries are used to extract information. Next, we describe the syntax and semantics of StarQL and its implementation over StarDM.

3.2.1 Syntax and Semantics

In an effort to make StarQL easy to understand with a short learning curve, a declarative SQL-like approach is used. Figure 3.2 shows an abstract BNF of some

```
<query>      := <select> | <import> | <delete> | <update>
<extractor> := PREFIXES | SUFFIXES | RANGE
<generator> := RMOTIFS | CMOTIFS | KMERS
<identifier>:= <path> | <name> | <import>
<filter>    := PREFIX | SUFFIX | SUBSTRING | <metadata>
<match>     := EXACT | APPROXIMATE | REGEX
<import>    := IMPORT <path> | <select> | <identifier> AS <identifier>
<delete>    := DELETE <identifier>
<select>    := SELECT <extractor> | <generator> | *
               FROM <identifier> | (<import>) | (<select>)
               [ WHERE <filter> | <match> | <metadata> ]
               [ <aggregate-ops> | <limit> | <order-by>]
```

Figure 3.2: Abstract BNF for StarQL.

of the StarQL constructs. Complex string analytics can be performed by easily forming queries and nesting different constructs. Next, we discuss StarQL constructs, grouped according to functionality, and give examples of their usage.

Administration

The `IMPORT` utility provides a simple way for users to insert and index strings from the file system to the database management system. The input parameters of an import command are: the file system directory (path) and the name of the new collection that will be created. The original path is not required for further operations on the imported strings but a pointer to the original path is kept for reference. The newly created collection is named and given a unique ID. The number of strings and the total length of all the strings are saved as collection properties. The `DELETE` utility removes a previously existing collection from the database system. Any indexes and meta-data can be purged but the origin of a deleted collection (path or another collection) is not affected.

For example, a user imports a dataset of human DNA shotgun reads by running the following query: `IMPORT ‘‘/datasets/shotgun/human’’ AS hdna;` . The

IMPORT construct also allows for duplicating an existing collection or saving the results of a query as a new collection.

Matching

The EXACT matching command finds exact matches of a query pattern in a collection. The output of an exact match operation is either a collection of one string, the matched pattern, or an empty collection if no matches are found.

The APPROXIMATE matching command finds matches within a certain distance threshold from the query pattern. The distance function can be hamming distance or edit distance, for example. The search process is similar to that of exact matching except that we allow differences between matches and patterns within a user-defined threshold. The result is a collection of as many unique substrings that match with their respective counts in the original collection.

The REGEX matching command finds substrings that match a regular expression pattern. The capabilities of a regular expression matching command are vital for several string applications. Regular expressions can be evaluated using deterministic finite automaton (DFA) over the strings according to Thompson's algorithm. The result is a collection of as many unique substrings that match with their respective counts in the original collection.

For example, assume a user needs to find matches of the regex expression "AC..CA" in the previously imported collection hdna. Notice that a collection of one element can be supplied inline for convenience. The query is written as: `SELECT * FROM hdna WHERE REGEX(''AC..CA'');`. To save the results for later processing, an import is needed: `IMPORT (SELECT * FROM hdna WHERE REGEX(''AC..CA'')) AS re;`.

Extraction

The `PREFIXES` extraction command extracts all the unique prefixes in a collection of strings. The input is a collection along with the desired prefix length range. The output is a collection of all unique prefixes within required length range. The `SUFFIXES` extraction command extracts all the unique suffixes in a collection of strings. The input is a collection along with the desired suffix length range. The output is a collection of all unique suffixes within required length range. The `RANGE` extraction command extracts all the unique substrings that exist at a specific position in a collection of strings. The input is a collection along with the desired start and end positions. The output is a collection of all unique substrings that exist at the specified positions from all the strings.

In our running example, a user may be interested in the different substrings of length 2 that exist between a pair of "`AC`". In this case, a range extraction query could be used as follows. `SELECT RANGE FROM re WHERE POS>1 AND POS<4;`. If `re` consisted of {`ACCCAC, ACGTAC, ACTTAC, ACATAC`}, then the output would be {`CC, GT, TT, AT`}.

Generation

The `RMOTIFS` generation command finds all the repeated motifs supported by at least one string in the collection operated on. The input is a collection along with the desired motif properties; namely, minimum length, maximum length, frequency threshold. In addition, motifs can be exact or approximate. The output is a collection of repeated motifs. The `CMOTIFS` generation command finds all the common motifs supported by a user specified number of strings in a certain collection. The input is a collection along with the desired motif properties; namely, minimum length, maximum length, frequency threshold. In addition, motifs can be exact or approximate. The output is a collection of common motifs. The `K-MERS` generation command finds all the unique k-mers in a collection of strings. The input is a collection along with the desired k value (substring length). The output is a collection of the unique k-mers from all the strings in the collection.

For example, to generate the k-mers of length 3 from the previously saved collection `re`, the query is: `SELECT KMERS FROM er WHERE LEN=3;`. The results consist of {`AAC, CCC, CCA, CAC, ACG, CGT, GTA, TAC, ACT, CTT, TTA, ACA, CAT, ATA`}.

Filtering

The `PREFIX` filtering command finds the strings that share a certain prefix; either exactly or approximately. The input is a collection of strings, a prefix pattern, and a distance function and threshold. The output is a collection of strings with matching prefixes. The `SUFFIX` filtering command finds the strings that share a certain suffix; either exactly or approximately. The input is a collection of strings, a suffix pattern, and a distance function and threshold. The output is a collection of strings with matching suffixes. The `SUBSTRING` filtering command finds the strings that share a certain substring; either exactly or approximately. The input is a collection of strings, a substring pattern, and a distance function and threshold. The output is a collection of strings with matching substrings. The `LENGTH` filtering command finds strings of a certain length range. The input is a collection of strings and a length range. The output is a collection of strings of the specified range.

Continuing our running example, assume the user is interested in the k-mers of length 3 that include the 2 characters between the pair of "`AC`" in the regular expression matches, `r =` {`CC, GT, TT, AT`}. This is accomplished using the query: `SELECT er.KMERS AS k FROM er, r WHERE k.LEN=3 AND k.SUBSTRING(r);`. The resulting filtered k-mers are {`CCC, CCA, CGT, GTA, CTT, TTA, CAT, ATA`}.

Meta-data

The `ID` is a unique identifier of a collection, a string, or an occurrence. The `NAME` is a textual description of a collection. The `SIZE` is the number of strings in a collection. The `LENGTH` is the total length of all the strings in a collection or the length of a string. The `ALPHABET` of a collection is the set of symbols used in all its strings. For example, if a collection has two string, {aba, cba}, then the alphabet of the collection is “abc”. The `ORIGIN` of a collection is the path or query that resulted in the creation of the collection. The transformations of strings from one collection to another can be tracked using this attribute.

3.2.2 Execution Plans

Complex StarQL queries can be formed easily. A query that involves multiple string operations can be optimized by ordering the execution of the underlying operations. For example, if a query involves multiple matching operations, one of which is against the longest common substring, then finding the longest common substring first reduces intermediate results and the search space for subsequent matching operations. However, it is not always straightforward to optimize query plans as the optimizer needs to verify semantics and keep them intact.

Consider two collections of strings, `hom` for human DNA and `mus` for mouse DNA. A biologist finds the sequences that are similar in both collections by using the approximate matching operation. For simplicity, we assume a hamming distance of 3 is acceptable. The following StarQL query results in all the subsequences in `hom` that have approximate matches in `mus`.

```
SELECT APPROXIMATE FROM hom, mus WHERE HAMM<=3;
```

Validating the correctness of string query plans is challenging. This is because the order of executing sub-queries could change the final results. For example, consider a user wants to find all common motifs in a group of strings that are of edit distance 2 or less from the word "test". The following plans will produce different results. (*i*) Extract all common motifs from all strings, then filter the motifs of edit distance 2 or less from "test". (*ii*) Filter strings of edit distance 2 or less from "test", then extract all common motifs from them. To ensure correctness, we use the syntax of StarQL to avoid such scenarios. In particular, the final output is always a subset of the operation after a `SELECT` keyword. Conditions after a `WHERE` keyword are interpreted from left to right, but not necessarily executed in this order. Using this convention, how StarQL interprets the query is clear to users. Only valid plans are compared internally to optimize efficiency.

Given the StarQL syntax and semantics, users easily form queries that represent their intentions and systems interpret them the same way. Hence, queries (*i*) `SELECT CMOTIFS FROM collection WHERE APPROXIMATE(‘‘test’’, 2);` and (*ii*) `SELECT * FROM collection WHERE CMOTIFS AND APPROXIMATE(‘‘test’’, 2);` are clearly different. The following nested query extracts the suffixes of length 3 from the Wikipedia repeated motifs.

```
SELECT SUFFIXES FROM (SELECT RMOTIFS FROM wiki
                               WHERE LEN>4 AND EDIT<3)
  WHERE LEN=3;
```

Alternatively, the motifs query could be saved as a collection for further processing. This allows users to analyze intermediate results and execute different queries without having to start from scratch. For example, after extracting suffixes of length 4 from the motifs, the user may want to explore prefixes or suffixes of different lengths. In this case, first the motifs are saved as a collection then used in subsequent queries as shown next. Motifs of length 5 will not contribute to the results of the prefixes query below.

```
IMPORT ( SELECT RMOTIFS FROM wiki WHERE LEN>4 AND EDIT<3 ) AS mot;
SELECT SUFFIXES FROM mot WHERE LEN=10;
SELECT PREFIXES FROM mot WHERE LEN>5 AND LEN<10;
```

3.3 A DBMS for Large-Scale String Analytics

In this section, we introduce StarDB, our proof-of-concept implementation. We demonstrate the use of our unified model and query language in a distributed system. We utilize parallel computation to augment the simplicity of our architecture.

3.3.1 System Architecture

Generally, the workloads of string algorithms scale super-linearly with data size [67]. A string database system needs to be distributed in order to handle string analytics in a timely manner. StarDB uses the master-worker parallel programming paradigm. Complex queries are solved in parallel whereas simpler ones are run in serial. The abstract architecture of StarDB is shown in Figure 3.3.

The supported StarDM operations are the interface to StarDB and are accessed using StarQL. Some operations are primitive and require less effort to execute, such as administrative tasks. Other operations require optimization and planning, such as matching. Naïvely, string operations require multiple scans to strings and mostly involves a much wider search space than the strings themselves. The complexity of a string operation depends on the algorithm, the string size, and the used parameters. We attack these challenges at several fronts by implementing the operators so they make use of the available system components and abstractions. Particularly, we utilize novel data structures and indexes to enhance performance

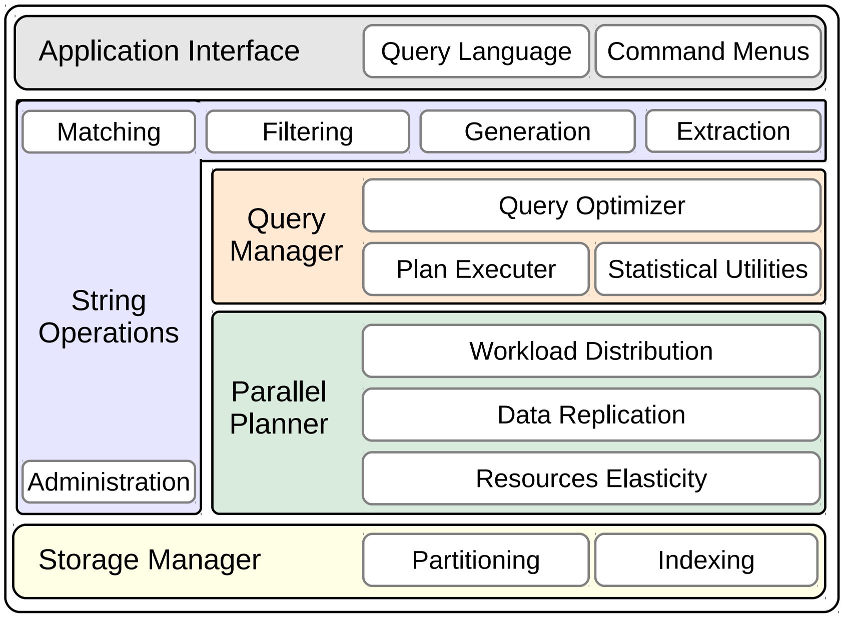


Figure 3.3: Architecture of our large-scale string database system.

and introduce optimizations at different system layers.

The analysis of a string is done by executing several queries. Users can utilize the command line interface or form and submit StarQL queries. The query manager consists of three main components. (i) The optimizer is needed to create efficient plans for queries. Adhering to the StarQL conventions, a query is optimized by pushing less complex operators down the execution tree, as long as correctness is maintained. This minimizes intermediate results and limits the domain of the more complex operators involved in the query plan. (ii) The execution of a query plan in a distributed system running over a large-scale infrastructure is not straightforward. The plan executer considers the underlying infrastructure and the current state of the system to find low cost means for solving the query. For example, less cores may be used in a multicore system to avoid contention or a fine grained task decomposition is done to better balance the workload in a large cluster. (iii) Statistical utilities are used to predict the complexity of an operator over a specific collection of strings and to guide the decisions of the optimizer and plan executer.

Resource utilization is a great challenge as balancing the workload and avoiding parallel execution overheads are essential tasks. The cloud manager is responsible for workload distribution, data replication, and system elasticity according to actual workload and availability of resources. String operations and the query manager use the cloud manager to interact with the hardware or work within its context. For example, load balancing is achieved by automatically finding the best probelm decomposition. Large-scale systems require resource pooling and elasticity in order to be cost effective. Adding and removing resources should be transparent to user applications. The elasticity manager estimates the required resources and updates the query plan executer. The effective orchestration of resources in a large-scale system is used to hide I/O cost and enhance the overall performance.

In a cloud environment that utilizes a large-scale infrastructure, the collective memory and disk of each machine form the storage subsystem. Issues that are simple for small strings are problematic for large strings. For example, reading a few gigabytes string takes more time than indexing a few megabytes string. Involved challenges include partitioning large string collections and prefetching to deal with I/O and network overheads. The storage manager provides abstractions that guarantee data independence. Indexes are used to minimize string access while keeping storage cost at a minimum. Under the hood, the storage manager creates decomposed indexes so parts of them can exist in local memory while other parts are in the memory of other nodes or on disk. In our implementation, we keep the strings in the file system as plain files and serialize their in-memory indexes to disk for persistence. Since the metadata of our collections is structures, we make use of an embedded relational database [68] to keep and access metadata.

3.3.2 Data Structures

The de facto data structure for a collection of strings is the generalized suffix tree (GST), a powerful full-text index. However, a GST is large in size, not scalable in terms of number of strings, and challenging to parallelize. A hefty constant is hidden when describing the space requirement of a GST as linear. In average, the number of nodes in a suffix tree is 2n, where n is the length of the indexed string. However, each node stores at least 3 integers to identify its path label (a string identifier, the starting position, and the length). A GST is not scalable because it requires as many distinct terminating symbols as the number of indexed strings. This is infeasible considering the number of strings in a set can be in the order of millions. Therefore, the alphabet size will be huge. Alternatively, GST leaf nodes will have to store a list of strings that share their path labels; adding to the space requirement. Moreover, parallel search in a GST would require extensive communication or excessive replication in the case of distributed processing.

With our goal of supporting generic string operations for large-scale analytics, we opt for a non-orthodox methodology based on lightweight indexing and parallel computation. Previously, generalized suffix trees were used to save space and provide information with minimal computation. However, at the targeted scale, random storage access to retrieve path labels is an unacceptable I/O burden and node size explodes in order to store string identifiers. We argue that parallel computation should be used with more basic data structures to support scalable and efficient string operations. We utilize a novel suffix trie index. The suffix trie indexes all suffixes of all strings and retains the frequency of every path label by storing a single integer in every node. Because we are targeting large collections of strings (e.g. thousands of motifs or regular expression matches), each node stores a single character, avoiding the need to reference strings to retrieve path labels. The space gain from compacting path labels is not significant in this case while the gain from avoiding excessive I/O is. We eliminate the need for different terminating symbols or maintaining string identifiers and compacting path labels. Some information that would have been readily available in a GST now requires extra computation. However, this is done in parallel in a distributed fashion.

Although the space requirements of suffix trees and tries are asymptotically the same, it is accepted that a suffix tree is space-efficient because it is a compressed trie. Nevertheless, long common labels are less expected in large collections of strings as the probability of having different combinations from a fixed alphabet increase with string lengths and collection size. Consequently, the construction complexity added for compacting path labels is unjustifiable given the expected space saving. In cases where most suffix tree labels are single characters, a trie is superior in both space requirement and access time. A trie node is more compact than a suffix tree node as no start position and label length need to be kept. In addition, retrieving a path label from a trie does not require accessing the indexed strings multiple times.

We assume that collections imported by users consist of long strings (e.g., genomic sequences) while collections that result from user queries consist of relatively short strings (e.g., matches and k-mers). This assumption allows us to be space-efficient in the case of long strings, where strings are stored and indexed separately using suffix trees. We still avoid a GST even for long strings because a huge single tree is challenging to manage. Operations over long strings avoid the search algorithms since their performance degrades as strings get longer. At the same time, multiple suffix trees fit the distributed nature of large-scale string analytics, where

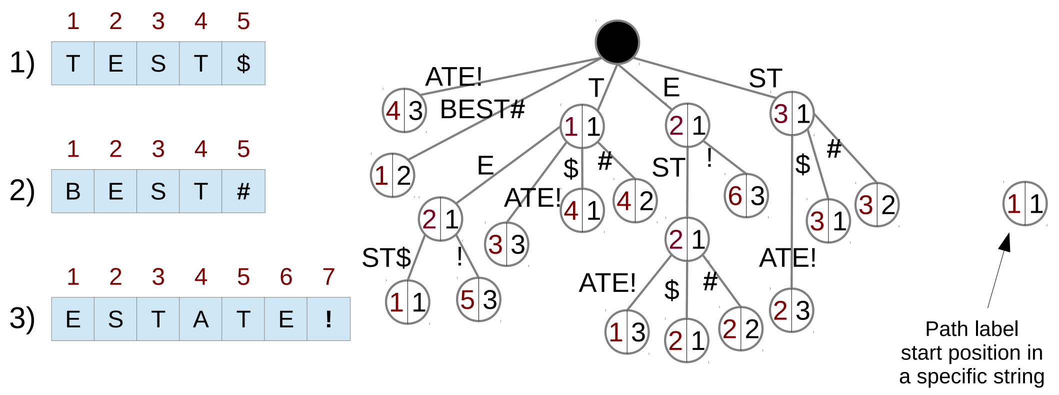


Figure 3.4: Example generalized suffix tree for a collection of 3 strings.

work on a collection could be partitioned among a number of workers. Suffix trees are well studied; we next focus on our suffix tries and parallel computation model.

Using our data structures, substrings in the index are not mapped to original strings in the collection. When exact positions or counts within each string are required, we utilize well-known string algorithms to efficiently extract this information in parallel. We strike a balance between preprocessing time, index size, and execution efficiency. For instance, Boyer-Moore search algorithm is run in parallel to find original strings that satisfy a certain filter query after pruning the search space using the suffix trie and an External R-Way merge sort algorithm is used to eliminate duplicate results after a pattern extraction query is executed. For example, consider the set of strings S = { TEST, BEST, ESTATE }. Figure 3.4 shows how S is represented using a GST.

We will next compare the GST with StarDM using the same example collection. The StarDM representation is shown in Figure 3.5. As a simple form of a suffix tree, StarDM is constructed in linear time by traversing the trie from the root using the suffixes. When a suffix exists, the counts of the nodes are incremented. Otherwise, new nodes are created for the newly added suffix and the counts are initialized to 1.

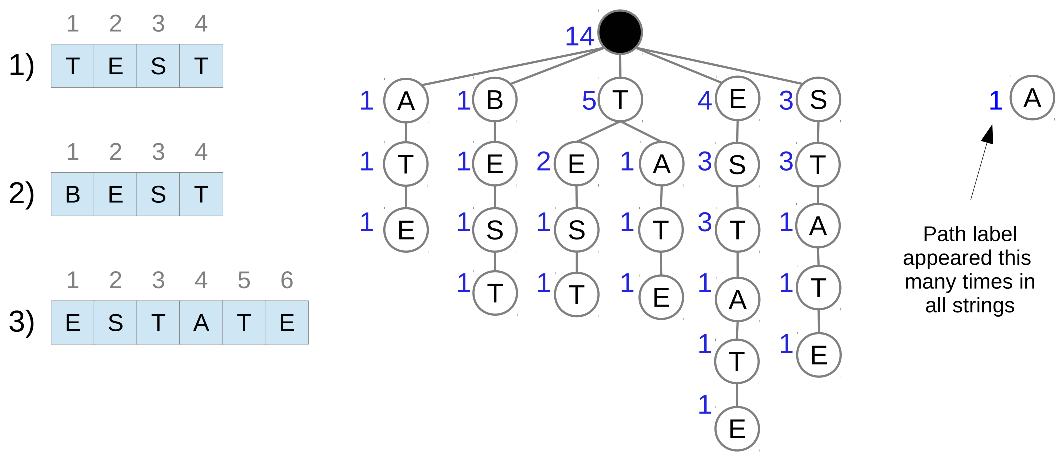


Figure 3.5: StarDM representation of strings in Figure 3.4.

In our example, finding the number of times "EST" appear in S requires traversing the GST to the node whose path label is "EST" then counting the leaf nodes below it, i.e., exhaustively traverse the sub-tree to find three leaf nodes indicating three appearances. In StarDM, we traverse the suffix trie following nodes "E", "S", then "T" to find that "EST" appeared three times. In StarDM, the total count of appearances in available by construction.

Next we show by example how we utilize parallel computation to extract information that is not stored in our novel data structure. Assume a user was interested in finding the positions where "EST" appears in S. Using the GST, instead of counting leaf nodes in the sub-tree rooted by "E" then "ST", we would find in each node the string identifier and the starting position of "EST" in that string. In StarDM, we would know from traversing the trie that "EST" appears three times. To find the exact positions, a parallel search is executed where the strings in S are distributed among workers to search for the three occurrences. This search is feasible because it is an exact search and the cost is distributed between workers.

To show how StarDM supports our string primitives we describe matching, filtering, extraction, and generation operations. Given the collection S from the

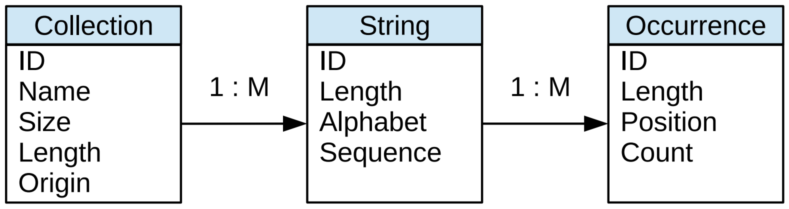


Figure 3.6: The meta-data of StarDM.

previous example, let T be another collection of one string, T = {WEST}. To execute match(S, T, edit(1)), the suffix trie of S is traversed to match "WEST" allowing one edit. The output is the set of matches {BEST, TEST, EST}. Now assume T = {TA}, to filter strings in S with a suffix "TA" allowing one mismatch, we use is_suffix(S, B, hamming(1)). The suffix trie for S is traversed to level 2 keeping the path labels that are within hamming distance 1 from "TA". To ensure a path label is a suffix, it must be a leaf or have its count is more than the sum counts of its children. In this case, the output is {ESTATE}. To extract the longest common substring in S, we run lcs(S) which first extracts the longest substrings that appear at least $|S| = 3$ times then verify that they exist in every string at least once. In the first step, the candidate solutions are ordered according to their length, {EST, ST, E, T}. In order to stop short, if possible, verification starts from the longest candidate. An exact parallel search is used to verify that "EST" appears in every string, which is the case in this example and the result is {EST}. Finally, to generate 3-mers of S, kmers(S, 3), the suffix trie branches of length three are simply spelled out {ATE, BES, TES, TAT, EST, STA}.

Our string primitives that find positions or report statistics (e.g. length and size) utilize the meta-data of StarDM. Depending on the required information, basic meta-data is used to answer some queries without accessing strings or indexes. Basically, meta-data includes statistics about the number of strings in a collection and the number of occurrences of substrings. Collections and strings are realized entities while occurrences are generated on-the-fly when needed. Figure 3.6 shows a representation of our meta-data. For instance, the previously used example set of strings S size is 3 (the number of strings) and length is 14 (the number of characters). Meta-data is used to filter collections, optimize the execution of some operations, and answer certain queries.

Example: Consider a user needs to find text that appears frequently in Wikipedia. The user has to work around spelling mistake and simple differences such as noun plurals and verb tenses. First, the Wikipedia archive is imported into StarDB using an administrative operation. StarDB indexes the dataset and may partition or replicate indexes depending on size and available resources. Motifs are patterns that appear frequently but not necessarily exactly. StarDB supports edit distance or hamming distance for approximate matching. To find all frequent patterns, a query to generate motifs is used. Running on 480 cores, the motifs search space (a combinatorial tree over English alphabet) is partitioned to 17,576 tasks (26^3 sub-trees). The workload is balanced by dynamically assigning tasks and the results are gathered and returned to the user.

The user may decide to filter out motifs of length 4 or less as they correspond to common short words, such as articles and prepositions. The user allows an edit distance of 2 characters so words like “fishes” and “fishy” count as occurrences for the motif “fish”. The length and approximate matching parameters are readily available in StarQL. The user in our example may form and submit the following StarQL query.

```
SELECT RMOTIFS FROM wiki WHERE LEN>4 AND EDIT<=2 AND FREQ>1000;
```

The Wikipedia archive is represented logically by one collection but indexed using suffix tries that fit in memory. StarDB first executes the repeated motifs operation in parallel. Since the motifs search space is a combinatorial tree, it is logically partitioned into many sub-trees. On a supercomputer, the parallel planner finds that 2,048 cores can be fully utilized given the query workload. The original archive is not accessed because suffix tries are annotated with counts. The resulting repeated motifs are also in the form of a suffix trie. Therefore, the common suffixes are easily extracted by a simple index traversal.

3.3.3 User-Defined Functions

User-defined functions extend StarDB with new operations or introduce application-specific logic using routines executed by other functions. For exmaple, one of the main routines used in string operations is the distance function. A distance function accepts two strings as input and outputs a scalar value indicating the dissimilarity of these strings. Maximizing a similarity is fundamentally the same as minimizing a distance. However, it is difficult to include every possible distance function. Therefore, user-defined functions are used to augment StarDB distance functions in order to support new metrics.

In bioinformatics, weighted matrices are used to measure the similarity of DNA sequences. The weights are based on a particular theory of evolution, where certain symbols are more likely to mutate into other symbols. Conventionally, rows and columns in a weighted matrix are associated with alphabet symbols. For instance, using the example matrix below, the distance between `AACT` and `GAAT` is $2 + 0 + 1 + 0 = 3$, whereas using Hamming distance it is 2.

$$W = \begin{array}{c|cccc} & \mathtt{A} & \mathtt{C} & \mathtt{G} & \mathtt{T} \\ \hline \mathtt{A} & 0 & 1 & 2 & 1 \\ \mathtt{C} & 1 & 0 & 3 & 2 \\ \mathtt{G} & 2 & 3 & 0 & 3 \\ \mathtt{T} & 1 & 2 & 3 & 0 \end{array}$$

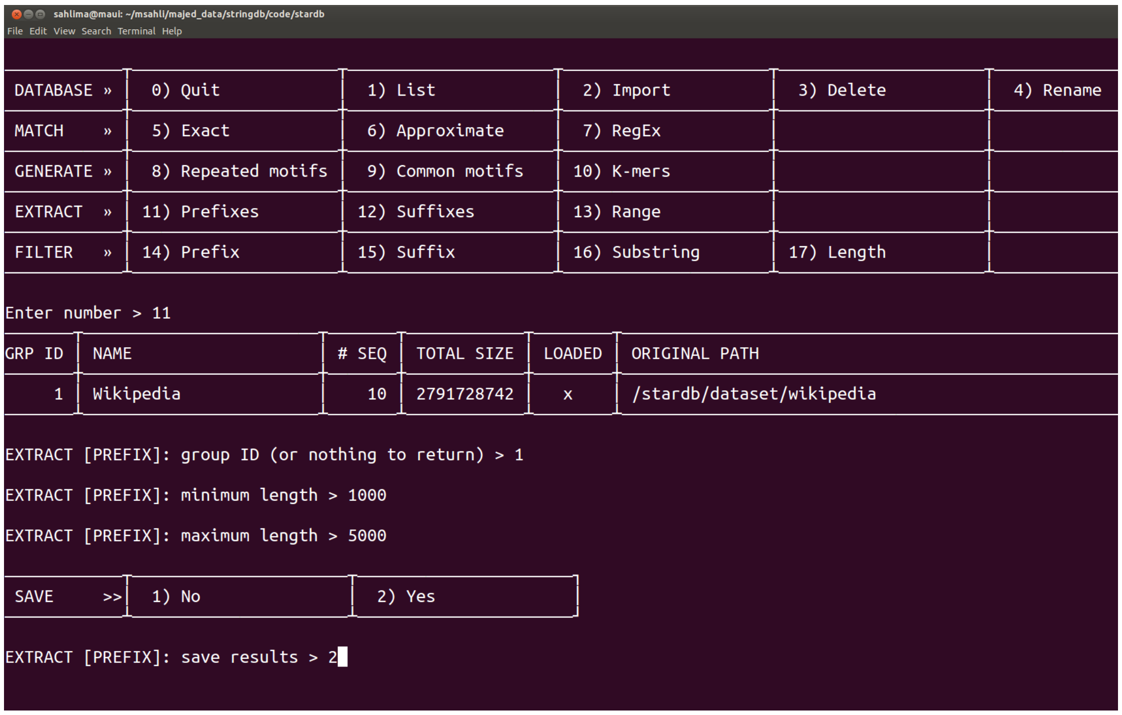
3.3.4 Implementation

Given our proposed architecture for a large-scale string database, we implemented StarDB using C/C++ and MPI. Key features include the following:

1. Scalability to large infrastructures.
2. Automatic tuning for execution.
3. Perform online analytical tasks.
4. Handle real string datasets.
5. Express complex string queries.
6. Extend matching with DNA weighted matrices.

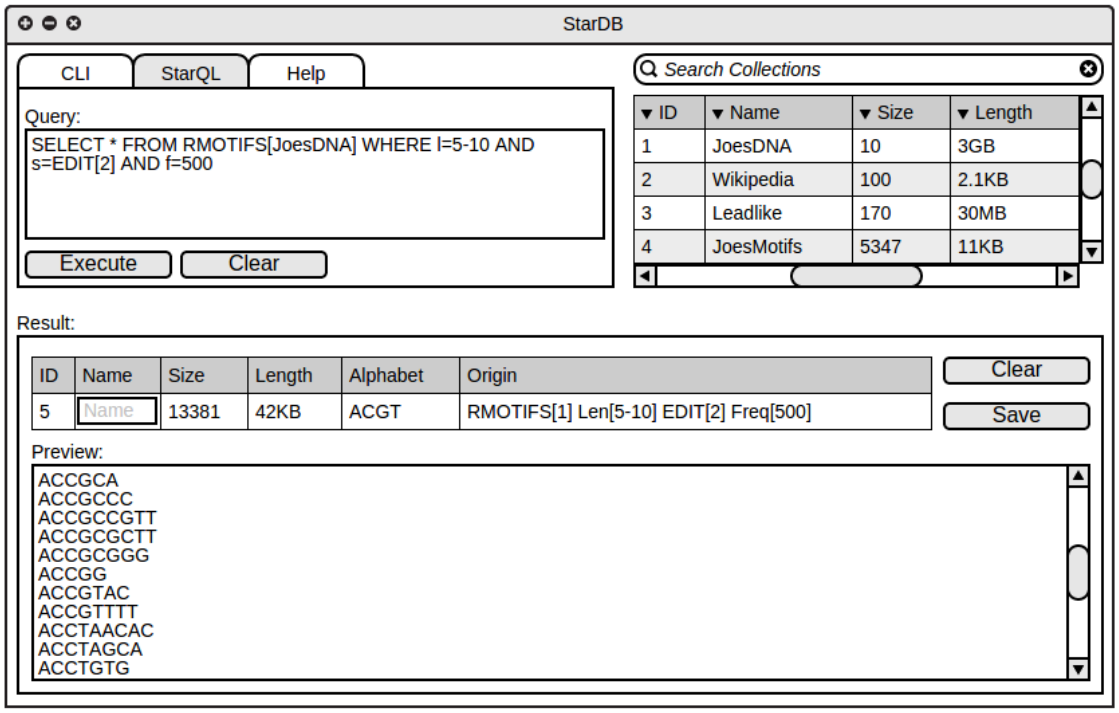
The application interface is handled by the master process. StarDB is accessed by a command line interface (CLI) or by a graphical user interface (GUI). The CLI is ideal to run over secure network connections when using remote servers and provides a command line menu to form queries and manage datasets. On workstations and local clusters, the StarDB GUI provides the same functionality of the CLI with a more user-friendly experience. Figure 3.7 shows sample screens.

We use real datasets from bioinformatics and the English language. We have tested StarDB using real datasets in the order of gigabytes (e.g., complete human genome) and synthetic datasets of large alphabets (i.e., upto 100 symbols). The used datasets are one to two orders of magnitude larger than previously reported results in string databases and parallel procedural string applications. The chosen datasets exhibit a variety of alphabets, strings of different lengths, and collections of different sizes. Moreover, these datasets show the difference in query workloads given different character repetition distributions.


sahlima@maui: ~/msahli/majed_data/stringdb/code/stardb
File Edit View Search Terminal Help

```
DATABASE »  |  0) Quit             |  1) List           |  2) Import       |  3) Delete     |  4) Rename
MATCH    »  |  5) Exact            |  6) Approximate    |  7) RegEx        |                |
GENERATE »  |  8) Repeated motifs  |  9) Common motifs  | 10) K-mers       |                |
EXTRACT  »  | 11) Prefixes         | 12) Suffixes       | 13) Range        |                |
FILTER   »  | 14) Prefix           | 15) Suffix         | 16) Substring    | 17) Length     |

Enter number > 11
GRP ID | NAME                        | # SEQ | TOTAL SIZE | LOADED | ORIGINAL PATH
     1 | Wikipedia                   |    10 | 2791728742 |   x    | /stardb/dataset/wikipedia

EXTRACT [PREFIX]: group ID (or nothing to return) > 1

EXTRACT [PREFIX]: minimum length > 1000

EXTRACT [PREFIX]: maximum length > 5000

SAVE     >> |  1) No                |  2) Yes              |

EXTRACT [PREFIX]: save results > 2
```

(a) Command line interface (CLI)



Query:

```
SELECT * FROM RMOTIFS[JoesDNA] WHERE l=5-10 AND
s=EDIT[2] AND f=500
```

ID	Name	Size	Length
1	JoesDNA	10	3GB
2	Wikipedia	100	2.1KB
3	Leadlike	170	30MB
4	JoesMotifs	5347	11KB

Result:

ID	Name	Size	Length	Alphabet	Origin
5	Name	13381	42KB	ACGT	RMOTIFS[1] Len[5-10] EDIT[2] Freq[500]

Preview:

```
ACCGCA
ACCGCCC
ACCGCCGTT
ACCGCGCTT
ACCGCGGG
ACCGG
ACCGTAC
ACCGTTTT
ACCTAACAC
ACCTAGCA
ACCTGTG
```

(b) Graphical user interface (GUI)

Figure 3.7: Sample StarDB interface screens.

StarDB fully utilizes the resource of an IBM Blue Gene/P supercomputer (16,384 CPUs) at over 90% speedup efficiency [19]. We also used a dedicated high-end 480-core Linux cluster. This cluster consists of 20 machines; each machine is equipped with 148GB RAM shared by 24 cores. StarDB was also tested on 40 Amazon EC2 instances.

3.3.5 Scenarios and Workloads

We consider scenarios of users performing analytical tasks on strings; such as finding frequent patterns, matching, counting, and generating k-mers. Queries can be executed in different orders, nested, saved, and further processed.

Bioinformatics scenario: Given the DNA and protein datasets, a biologist needs to find the patterns that are frequent within every genomic sequence and at the same time common between sequences. Such patterns have potential functional importance and can be used to draw conclusions across species or to find mutations within a species. The workload of such task is high due to the exponential search space size with respect to the string's alphabet size. The task translates to generating repeated motifs, generating common motifs, then finding the motifs that are both repeated and common. Another way of executing this task is to generate repeated (or common) motifs, then count the matches to check if they are also common (or repeated). The total search space is smaller in this case. For matching, we extend StarDB with a user-defined function that uses a weighted distance matrix for DNA sequences. The following StarQL query is an example for this scenario.

```
SELECT CMOTIFS(dna) AS c FROM dna,(SELECT RMOTIFS as r FROM dna
    WHERE r.LEN>10 AND r.USERDIST<2 AND r.FREQ>10000) AS rep
  WHERE c.LEN>10 AND c.USERDIST<2 AND c.FREQ=10 AND c.EXACT(rep);
```

First, the inner sub-query is evaluated and its results are saved as a collection (`rep`). This evaluation is done in parallel by executing the repeated motifs operator. The collection `rep` is only available for the current query because it is not imported. Then, the outer sub-query is evaluated by running the common motifs operator resulting in another temporary collection (`c`). The exact filter (`c.EXACT(rep)`) is used to find the intersection between the repeated and common motifs. After returning the final results, the temporary collections and their indexes are dropped.

Literary scenario: Given the English text of Wikipedia, a librarian is curious about how writers start articles. She wants to explore the letters that frequently appear consecutively in the prefaces of Wikipedia pages. In StarDB, the k-mers operation finds symbols that appear consecutively and reports their frequencies. To find k-mers from the beginnings of articles, the k-mers are generated from the prefixes of the texts or are checked to exist in a specific range. To explore, the librarian may need to increase k-mers lengths as long as frequency is high. Given StarDB indexing, these queries are of low workloads. The following two StarQL queries find the top 20 k-mers of length 4, 8, or 16, that appear in the first 100 characters more than 200 times across the dataset.

```
IMPORT (SELECT PREFIXES AS p FROM wiki WHERE p.LEN=100) AS pref;
SELECT KMERS AS k FROM pref WHERE k.LEN=4 OR k.LEN=8 OR k.LEN=16 AND
  COUNT(k)>200 ORDER BY COUNT(k) DESC LIMIT 20;
```

The first query translates to creating a new collection in the database from the results of the prefix extractor. The second query is run by executing the k-mers operation with length and frequency filters to keep the top 20 results only.

3.4 Discussion

Considering the complexity of a distributed database system for large-scale string analytics, it is best to deal with the challenges one at a time. First, we choose a string operation that faces the challenges of the larger system. We focus on repeated motifs extraction, a core string problem used in many applications. For example, they are used to identify biological functionality in genomic sequences, periodicity in time series, or user activity trends in web logs. A repeated motif is a pattern that approximately appears many times in a string. Motifs extraction is a time consuming task with random data access patterns and an exponential search space. Efficiency is key to perform string analytics. Moreover, scaling to larger strings and supporting more interesting motif types of higher workloads, such as supermaximal ones, require scaling out. Merely using more cores does not guarantee speedup efficiency, because of parallel overheads, such as duplicate work and communication. Minimizing user tuned parameters frees the system from the user assumptions and previous (possibly limiting) background. We require automatic tuning and elasticity features in order to be practical at scale.

Chapter 4

Cache Efficiency Applied to Motif Extraction

Obviously, the highest type of efficiency is that which can utilize existing material to the best advantage.

Jawaharlal Nehru
1889 — 1964 CE

We introduced CAST, a cache-aware method for solving the combinatorial repeated motifs problem. CAST arranges the suffix tree in continuous memory blocks, and accesses it in a way that exhibits spatial and temporal locality, thus minimizing cache misses. As a result, CAST improves performance of serial execution by an order of magnitude.

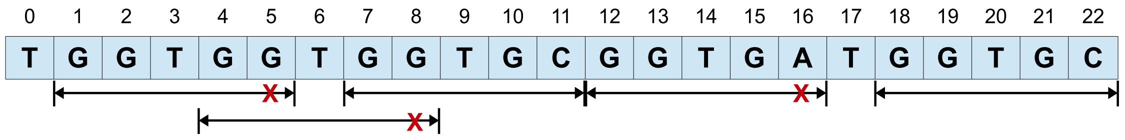


Figure 4.1: Example string S over DNA alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Occurrences of motif candidate $m = \mathtt{GGTGC}$ are indicated, assuming distance threshold $d = 1$. X refers to a mismatch between m and the occurrence. Occurrences may overlap.

4.1 Background

This section introduces necessary definitions and defines the repeated motifs problem. The problem search space and index used in the solution are then discussed.

4.1.1 Motifs

A string S over an alphabet Σ is an ordered and finite list of symbols from Σ. $S[i]$ is the ith symbol in S, where $0 \leq i < |S|$. A substring of S that starts at position i and ends at position j is denoted by $S[i, j]$ or simply by its position pair (i, j). For example, (7,11) represents `GGTGC` in Figure 4.1. Let $\mathcal{D}$ be a function that measures similarity between two strings. Following the previous work [69, 13], in this chapter we assume $\mathcal{D}$ is the Hamming distance (i.e., number of mismatches). A *motif candidate* m of length n is an n-tuple of symbols from Σ. A substring $S[i, j]$ is an *occurrence* of m in S, if the distance between $S[i, j]$ and m is at most d, where d is a user-defined distance threshold. The set of all occurrences of m in S is denoted by $\mathcal{L}(m)$. Formally: $\mathcal{L}(m) = \{(i, j) | \mathcal{D}(S[i, j], m) \leq d\}$.

Definition 1 (Motif)**.** *Let S be a string, $\sigma \geq 2$ be a frequency threshold, and $d \geq 0$ be a distance threshold. A candidate m is a* motif *if and only if there are at least σ occurrences of m in S. Formally: $|\mathcal{L}(m)| \geq \sigma$.*

Definition 2 (Maximal motif). *A motif m is maximal if and only if it cannot be extended to the right nor to the left without changing its occurrences set. Formally, $\mathcal{L}(m) \neq \mathcal{L}(\alpha m) \neq \mathcal{L}(m\beta)$, where α and $\beta \in \Sigma$.*

A maximal motif must be *right* and *left maximal* [69]. A motif m is right maximal if $\mathcal{L}(m\beta)$ has less occurrences or more mismatches than $\mathcal{L}(m)$. Similarly, a motif m is left maximal if extending m to the left causes a loss in occurrences or introduces new mismatches. There is excessive overlapping among maximal motifs. Users are typically interested in longer motifs [69], such as the right-supermaximal ones, denoted by *rs-motifs* in the rest of this chapter.

Definition 3 (Right-Supermaximal Motif). *Let M be the set of maximal motifs from Definition 2 and let $\widehat{m} \in M$. $\widehat{m}$ is a* rs-*motif, if $\widehat{m}$ is not a prefix of any other motif in M. We call M_{rs} the set of all rs-motifs. Formally: $M_{rs} = \{\widehat{m} | \widehat{m} \in M, \widehat{m}\alpha \notin M, \alpha \in \Sigma\}$.*

The number of possible motif candidates for a certain σ value is $\sum_{i=1}^{|S|-\sigma+1} |\Sigma|^i$, where $|\Sigma|$ is the alphabet size. To restrict the number of motif candidates, previous works have imposed minimum (l_{min}) and maximum (l_{max}) length constraints.

Problem. *Given string S, frequency threshold $\sigma \geq 2$, distance threshold $d \geq 0$, minimum length $l_{min} \geq 2$, and maximum length $l_{max} \geq l_{min}$; find all rs-motifs.*

The most interesting case is when $l_{max} = \infty$. Obviously, this is also the most computationally expensive case since the length cannot be used as an upper bound.

4.1.2 Trie-based Search Space and Suffix Trees

The search space of a motif extraction query is the set of motif candidates for that query. The size of such search space is astronomical even for a short input string

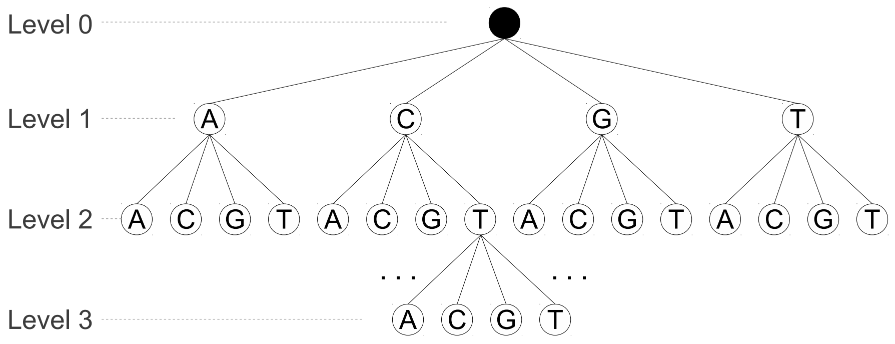


Figure 4.2: Two levels of the search space trie for DNA motifs, alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.

and a small alphabet. A combinatorial trie (see Figure 4.2) is used as a compact representation of the search space. Every path label formed by traversing the trie from the root to a node is a motif candidate. Finding the occurrences of each motif candidate and verifying maximality conditions require a large number of expensive searches in the input string S. To minimize this cost, a suffix tree index is typically used. A suffix trees is a full-text index where the path labels from the root to the leaves correspond to the suffixes of the indexed string [12]. It is built in linear time and space as long as the string and the tree fit in memory [70].

The properties of the suffix tree facilitate the verification of right and left maximality as discussed by Federico and Pisanti [69]. For the sake of completeness, we highlight the following essential properties. (i) A suffix tree node is *left-diverse* if at least two of its descendant leaves have different left symbols in S. Based on the suffix tree, a motif m is left maximal if one of its occurrences is a left-diverse suffix tree node. (ii) The labels of the children of an internal suffix tree node start with different symbols. Hence, if a motif has an occurrence that consumes the complete label of an internal node, then it is right maximal.

We annotate the suffix tree by traversing it once and storing in every node whether it is left-diverse or not and the number of leaves reachable through it.

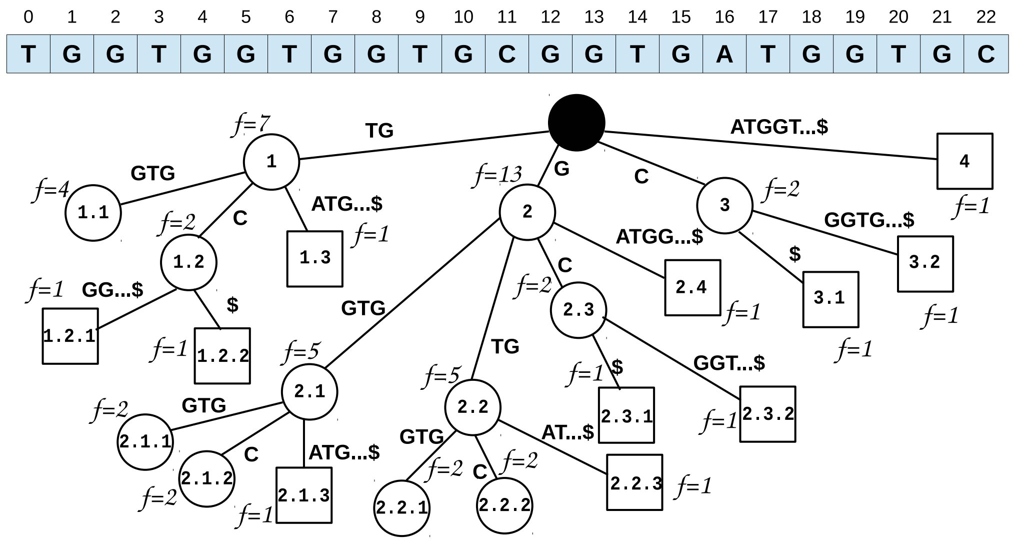


Figure 4.3: Example sequence over $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and its suffix tree. Suffix tree nodes are annotated with the frequency of their path labels and are numbered for referencing in the text.

This number is the frequency of a node's path label. For example, the path label for node `1.2` in Figure 4.3 is `TGC` and it is not a left-diverse node as `TGC` is always preceded by `G` in S. Node `1.2` is annotated with $f = 2$ because `TGC` appears in S at $\{(9, 11), (20, 22)\}$. For simplicity, Figure 4.3 does not show the left-diversity annotation. In case of exact motifs, where $d = 0$, the search space is reduced to the suffix tree [71]. For the general case, where $d > 0$, occurrences of a candidate motif are found at different suffix tree nodes. The frequency of the motif is calculated by summing the annotations from all these nodes.

Example. Let us start a depth first traversal of the search space trie in Figure 4.2 to extract motifs from the example sequence and its suffix tree in Figure 4.3. Assume $d = 1$ and $\sigma = 10$. The trie traversal starts at node `A` in the first level. By traversing the suffix tree, we find that the first symbol from every branch starting at the root is an occurrence of `A` within distance 1. Therefore, the occurrences set contains the following suffix tree nodes: $\mathcal{L}(\mathtt{A}) = \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$ and a total frequency of $7 + 13 + 2 + 1 = 23$. Search space traversal continues to the first child of `A`, representing the motif candidate `AA`. The occurrences set of `A` is used to create `AA`'s. The label of suffix tree node `1` is `TG`. It already has distance 1 from `A` in the first level. Extending the occurrence by one symbol introduces another mismatch for `AA` so it is discarded. To extend the label for the second occurrence we need to check all branches from suffix tree node `2`. The first three children `2.1` to `2.3` of node `2` are pruned for exceeding the allowed distance. Node `2.4` is added to the occurrences set of `AA` since its path label is `GA`, which has distance 1 from `AA`. The rest of the occurrences of `A` are extended and validated in the same manner. The occurrences set of `AA` is $\mathcal{L}(\texttt{AA}) = \{\texttt{2.4}, \texttt{4}\}$ with a total frequency of $1 + 1 = 2$. `AA` is not frequent enough and the search space is pruned by backtracking to node `A` in Figure 4.2. Then, `AC` is processed in the same way.

4.2 Cache-optimized Motif Extraction (CAST)

We maintain the search space of motif extraction using our cache-optimized mechanism, called CAST[1]. CAST is implemented as part of our exact motif extractor, ACME.

4.2.1 Spatial and Temporal Memory Locality

Existing motif extraction methods realize the search space trie as a set of nodes, where each node has a one character label, pointers to its parent and children, and its occurrences set. These nodes are dynamically allocated and deallocated. The maximum number of nodes to be created and then deleted from main memory is $\sum_{i=1}^{l_{max}} |\Sigma|^i$. For example, when $l_{max} = 15$ and $|\Sigma| = 4$, the maximum number of

[1]CAST stands for Cache Aware Search space Traversal

nodes is 1,431,655,764. These nodes are scattered in main memory and visited back and forth to traverse all motif candidates. Consequently, existing methods suffer dramatically from cache misses, plus the overhead of memory allocation and deallocation.

A branch of trie nodes represents a motif candidate as a sequence of symbols. These symbols are conceptually adjacent with preserved order; allowing for spatial locality. Moreover, maintaining occurrences sets is a pipelined process; for instance, the occurrences set of `AA` is used to build the occurrences set of `AAA`. This leads to temporal locality. Existing approaches overlooked these important properties.

We propose CAST, a representation of the search space trie together with access methods, that is specifically designed to be cache efficient by exhibiting spatial and temporal locality. For spatial locality, CAST utilizes an array of symbols to recover all branches sharing the same prefix. The size of this array is proportional to the length of the longest motif to be extracted. For instance, a motif of 1K symbols requires roughly a 9KB array. In practice, motifs are shorter. We experimented with DNA, protein, and English sequences of gigabyte sizes, where the longest frequent motif lengths are 28, 95 and 42 symbols respectively. Moreover, the occurrences set is also realized as an array. A cache of a modern CPU can easily fit a sub-trie branch and, in most cases, its occurrences array. For temporal locality, once we construct the occurrences array $\mathcal{L}(v_i)$ of branch node v_i, we revisit each occurrence to generate $\mathcal{L}(v_{i+1})$. We take advantage of the fact that the total frequency of $\mathcal{L}(v_{i+1})$ is bounded by that of $\mathcal{L}(v_i)$. Therefore, with high probability all data necessary for the traversal and validation are already in the cache.

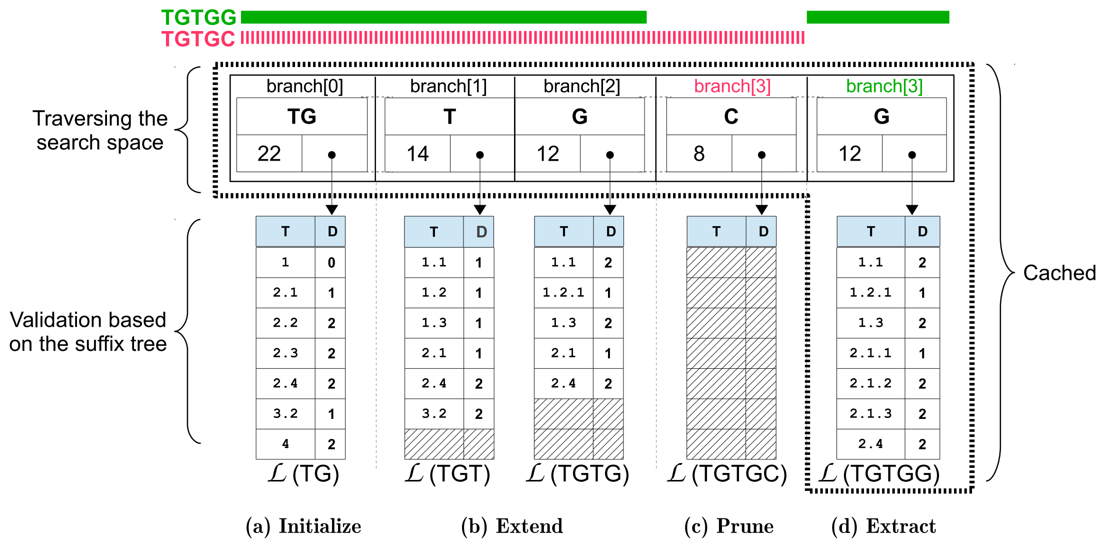


Figure 4.4: Snapshot of CAST processing for $Q'(|S| = 23, \sigma = 12, l_{min} = 5, l_{max} = 5, d{=}2)$ over the sequence of Figure 4.3. Prefix `TG` is extended one symbol at a time to maintain `TGTGC` and `TGTGG` branches. A branch is traversed from ancestor to descendant by moving from left to right. CAST array (*branch*) and the occurrences array of the deepest descendant are easily cached, since both fit into small contiguous memory blocks.

4.2.2 CAST Algorithm

CAST extracts valid motifs as follows: (*i*) initialize the sub-trie prefix; then (*ii*) extend the prefix as long as it leads to valid motif candidates; otherwise (*iii*) prune the extension. In the rest of this section we consider sequence S from Figure 4.3 and use an example query Q with $\sigma = 12$, $l_{min} = l_{max} = 5$ and $d = 2$.

Algorithm 1 shows the details. Let *branch* be the sub-trie branch array. An element $branch[i]$ contains a symbol c, an integer F, and a pointer, as shown in Figure 4.4. Each sub-trie has a prefix p that is extended to recover all motif candidates sharing p. $branch[i]$ represents motif candidate $m_i = pc_1 \ldots c_i$, where c_i is a symbol from level i of the sub-trie (see Figure 4.2). F_i is the total frequency of m_i and the pointer refers to $\mathcal{L}(m_i)$. Each occurrence in $\mathcal{L}(m_i)$ is a pair $\langle T, D \rangle$, where T is a pointer to a suffix tree node whose path label matches motif candidate

```
Input: $l_{min}$, $l_{max}$, prefix $p$
Output: Valid motifs with prefix $p$
1  Let branch be an array of size $l_{max} - |p| + 1$
2  $branch[0].L \leftarrow$ getOccurrences($p$)
3  $branch[0].F \leftarrow$ getTotalFreq($branch[0].L$)
4  $i \leftarrow 1$
5  $next \leftarrow$ DepthFirstTraverse($i$)
6  while $next \neq$ END do
7      $branch[i].C \leftarrow next$
8      $branch[i].F \leftarrow branch[i-1].F$
9      foreach occur in $branch[i-1].L$ do
10         if occur is a full suffix tree path label then
               // check child nodes in suffix tree
11             foreach child of occur.T do
12                 if first symbol in child label $\neq$ next then
13                     $child.D = occur.D + 1$
14                     if $child.D > d$ then
15                         Discard(child)
16                         if $branch[i].F < \sigma$ then
17                             Prune($branch[i]$)
18                 else
19                     add child to $branch[i].L$
20         else
               // extend within label in suffix tree
21             if next symbol in occur.T label $\neq$ next then
22                 increment occur.D
23                 if $occur.D > d$ then
24                     Discard(occur)
25                     if $branch[i].F < f$ then
26                         Prune($branch[i]$)
27             else
28                 add occur to $branch[i].L$
29     if isValid($branch[i]$) then Output($branch[i]$)
30     $i{+}{+}$
31     $next \leftarrow$ DepthFirstTraverse($i$)
```

Algorithm 1: Cache efficient motif extraction (CAST)

m_i with D mismatches. $branch[0]$ represents the fixed-length prefix of the sub-trie. F_0 is a summation of the frequency annotation from each suffix tree node in $\mathcal{L}(p)$.

Prefix Initialization

Algorithm 1 starts by creating the occurrences array of the given fixed-length prefix before recovering motif candidates. CAST commences the occurrences array maintenance for a prefix by fetching all suffix tree nodes at depth one. The maximum size of the occurrences array at this step is $|\Sigma|$. The distance is maintained for the first symbol of the prefix. Then the nodes, whose distances are less than or equal to d, are navigated to incrementally maintain the entire prefix. The number of phases to maintain the occurrences array of prefix p is at most $|p|$.

For example, the sub-trie with prefix `TG` is initialized by CAST in two phases using the suffix tree in Figure 4.3. Figure 4.4(a) shows the final set $\mathcal{L}(\texttt{TG})$ of occurrences in S. The first element in $\mathcal{L}(\texttt{TG})$ is $\langle \texttt{1}, 0\rangle$ because the path label of suffix tree node `1` is `TG` with no mismatches from our prefix. The second element in $\mathcal{L}(\texttt{TG})$ is $\langle \texttt{2.1}, 1\rangle$ because the first two symbols from the path label of suffix tree node `2.1` are `GG` with one mismatch from our prefix. The total frequency of `TG` at $branch[0]$ is the sum of frequencies of the suffix tree nodes in $\mathcal{L}(\texttt{TG})$: $7 + 5 + 5 + 2 + 1 + 1 + 1 = 22$.

Extension, Validation and Pruning

Since `TG` is frequent enough, it is extended by traversing its search space sub-trie. The depth-first traversal (DFT) of the sub-trie starts at line 5 in Algorithm 1 to extend $branch[0]$; it considers all symbols of Σ at each level of the DFT. At level i, DepthFirstTraverse returns c_i to extend $branch[i-1]$. Figure 4.4(b) demonstrates the extension of $branch[0]$ with symbols `T` then `G`.

The maintenance of the occurrences set is a pipelined function, where $\mathcal{L}(branch[i+1])$ is constructed from its parent's, $\mathcal{L}(branch[i])$. This process is done in the loop starting at line 9. For example, $\mathcal{L}$(TGT) is created by navigating each element in $\mathcal{L}$(TG). The first element of $\mathcal{L}$(TG) adds suffix tree nodes 1.1, 1.2, and 1.3 to $\mathcal{L}$(TGT) with distance 1 since their labels do not start with T. The second element of $\mathcal{L}$(TG) is added to $\mathcal{L}$(TGT) since its label was not fully consumed. In node 2.2, the next symbol of its label introduces the third mismatch. Thus, the third element of $\mathcal{L}$(TG) is discarded. The rest of $\mathcal{L}$(TG) is processed in the same way. The total frequency at $branch[1]$ drops to 14. Similarly, $\mathcal{L}$(TGTG), $\mathcal{L}$(TGTGC) and $\mathcal{L}$(TGTGG) are created in Figure 4.4(b), Figure 4.4(c), and Figure 4.4(d), respectively.

A node at $branch[i]$ can be skipped by moving back to its parent at $branch[i-1]$, which is physically adjacent. Therefore, our pruning process has good spatial locality, where backtrack means move to the left. For example in Figure 4.4(c), the total frequency of TGTGC drops below the frequency threshold $\sigma = 10$ after discarding node 1.1 of frequency 4 from $\mathcal{L}$(TGTG), i.e. $12 - 4 < \sigma$. Since TGTGC has frequency less than σ, we do not need to check the rest of the occurrences and the branch is pruned (see line 16 in Algorithm 1).

After pruning TGTGC, CAST backtracks to $branch[2]$ which will now be extended using G. All occurrences from $branch[2]$ are also valid for TGTGG at $branch[3]$ with no change in total frequency. The IF statement in line 29 returns true since the branch represents a valid motif of length 5 and function OUTPUT is called. The next call to DEPTHFIRSTTRAVERSE finds that $i > l_{max}$ so it decrements i until the level where an extension is possible or the sub-trie is exhausted.

CAST supports exact-length motifs, maximal motifs, and supermaximal motifs. Function ISVALID in line 29 determines whether a branch represents a valid motif or not as discussed in Section 4.1. For exact-length motifs, only branches of that length are valid. For maximal motifs IsValid returns false if: (i) $branch[i]$ could be extended without changing its occurrences list (i.e., not right maximal); or (ii) none of its occurrences is a left-diverse node (i.e., not left maximal). For supermaximal motifs, IsValid passes the right-supermaximal motifs to a combiner that implements Algorithm 3 as discussed in Section 5.3.

4.3 Evaluation

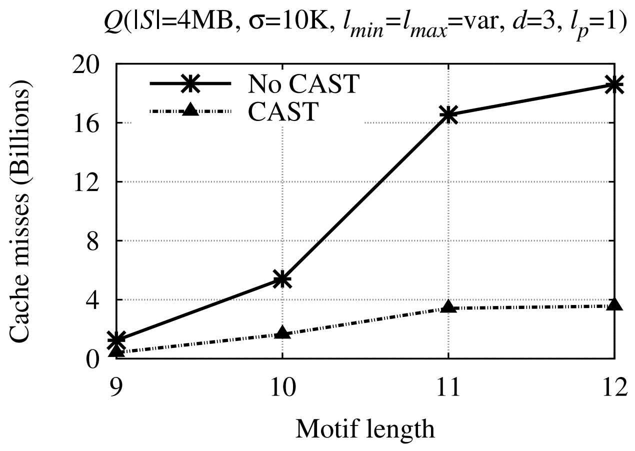
Table 4.1 shows the specifications of the machines used to evaluate CAST. The cache efficiency experiments were run on *System#1* and the comparison experiments between ACME and the other systems were run on *System#2*. The sequences used in these experiments are from the human genome DNA sequence and the size is noted in each experiment along with the other parameters, namely, frequency, length, and distance.

Table 4.1: Specifications of the systems used to evaluate CAST.

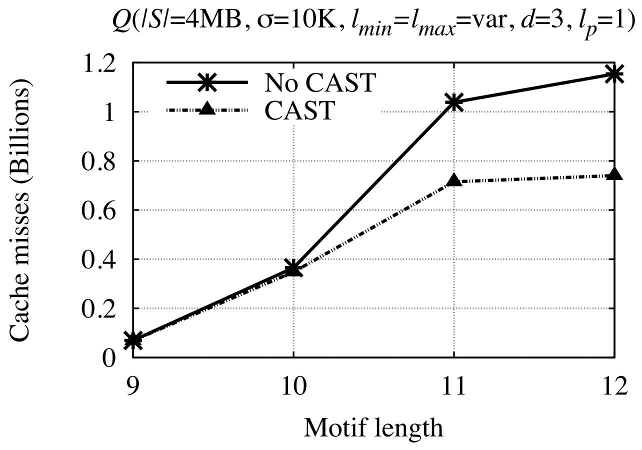
#	Architecture	Cores	RAM	Cache		
				L1	L2	L3
1	32-bit Linux	2 @2.16GHz	2GB	64KB	1MB	-
2	64-bit Linux	12 @2.67GHz	192GB	64KB	256KB	12MB

4.3.1 Cache efficiency

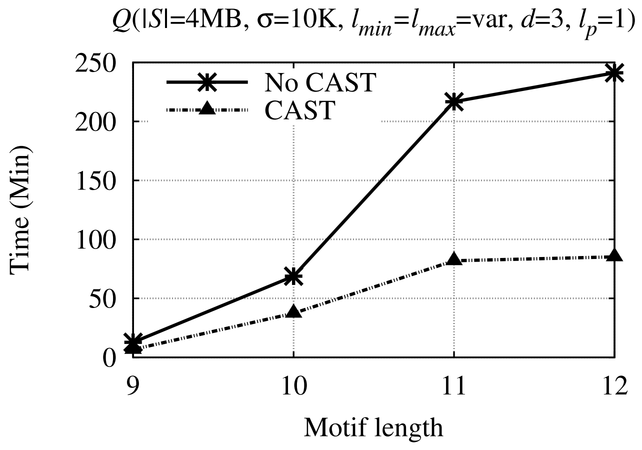
Existing motif extraction methods incur a lot of cache misses while traversing the search space. ACME uses our CAST approach to represent the search space in contiguous memory blocks. The goal of this experiment is to demonstrate the cache efficiency of CAST. We implemented the most common traversing mechanism utilized in the recent motif extraction methods, such as FLAME and MADMX, as discussed in Section 4.1. We refer to this mechanism, as *NoCAST*.



(a) L1 cache misses



(b) L2 cache misses



(c) Time performance

Figure 4.5: Correlation between caches misses and motif extraction time; DNA dataset.

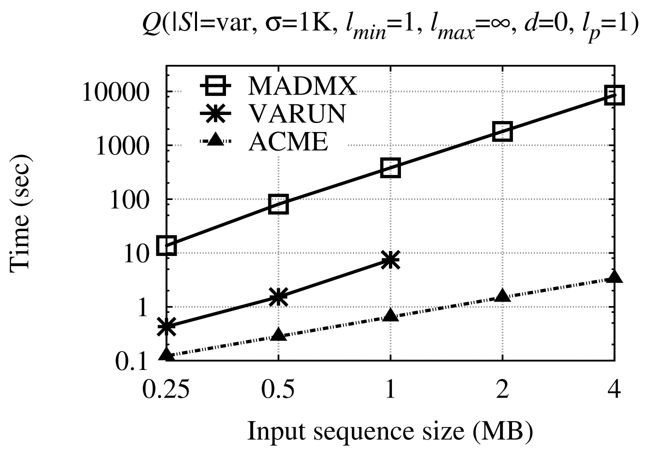


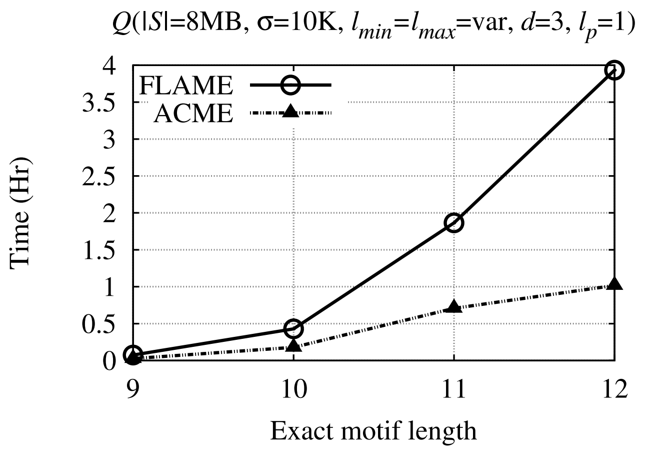
Figure 4.6: Serial execution of ACME extracting maximal motifs vs MADMX and VARUN.

We used the *perf* Linux profiling tool to measure the L1 and L2 cache misses. CAST significantly outperforms NoCAST in terms of cache misses and execution time especially when the motif length, and consequently the workload, is increased, as shown in Figure 4.5. The difference between CAST and NoCAST shows earlier in L1 cache. However, the difference in L2 cache misses starts to show as the motif length is increased and has the same pattern. The correlation between cache efficiency (Figure 4.5(a) and Figure 4.5(b)) and serial execution time (Figure 4.5(c)) is clear.

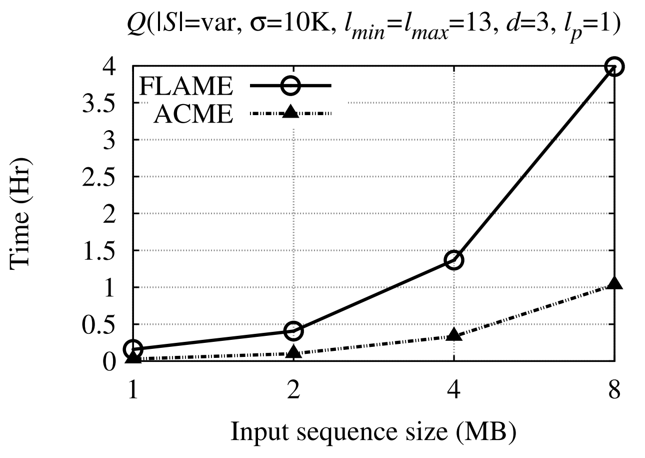
4.3.2 Comparison against state-of-the-art

We compared ACME against FLAME, MADMX, and VARUN. Since the source code for FLAME was not available, we implemented it using C++. MADMX[2] and VARUN[3] are available from their authors' web sites. These systems do not support parallel execution and are restricted to particular motif types. The following experiments were executed on *System#2*. Since our competitors run serially,

[2]http://www.dei.unipd.it/wdyn/?IDsezione=6376
[3]http://researcher.ibm.com/files/us-parida/varun.zip

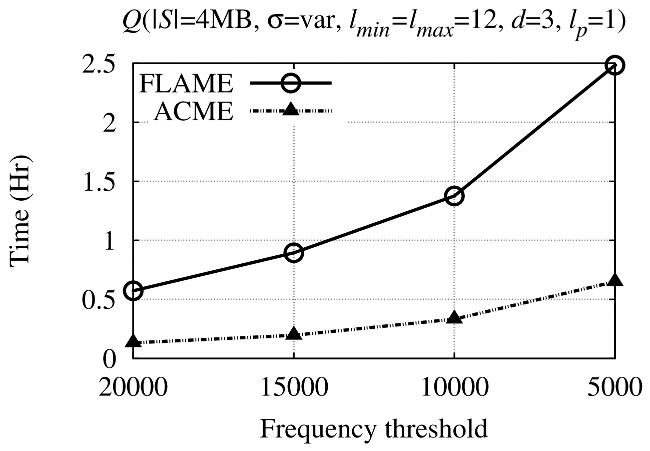


(a) Variable motif length

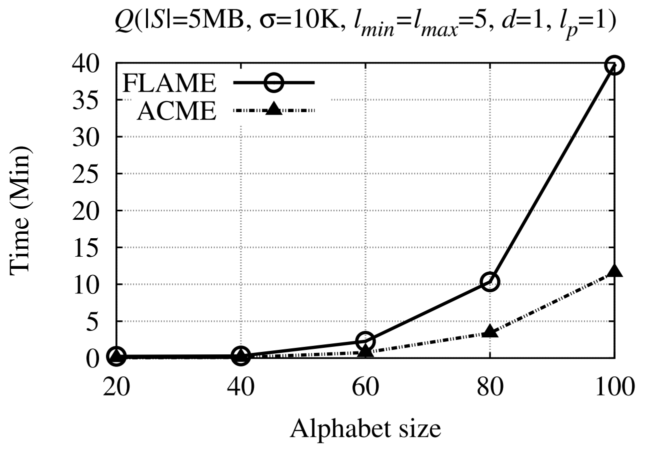


(b) Variable sequence size

Figure 4.7: Serial execution of ACME extracting exact length motifs vs FLAME. ACME is superior as the workload is increased using different factors.



(a) Variable frequency



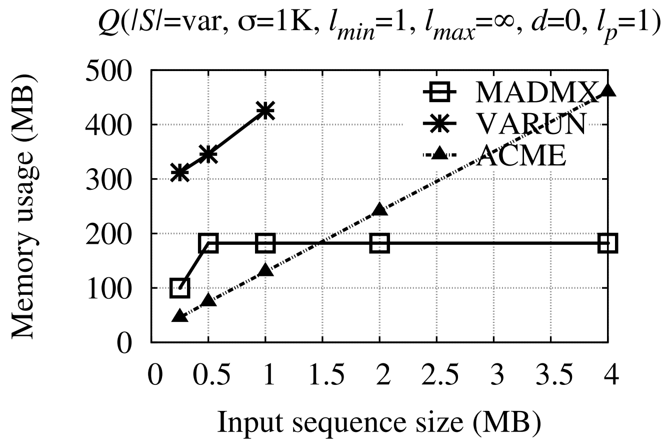
(b) Variable alphabet size

Figure 4.8: Serial execution of ACME extracting exact length motifs vs FLAME. ACME is superior as the workload is increased using different factors.

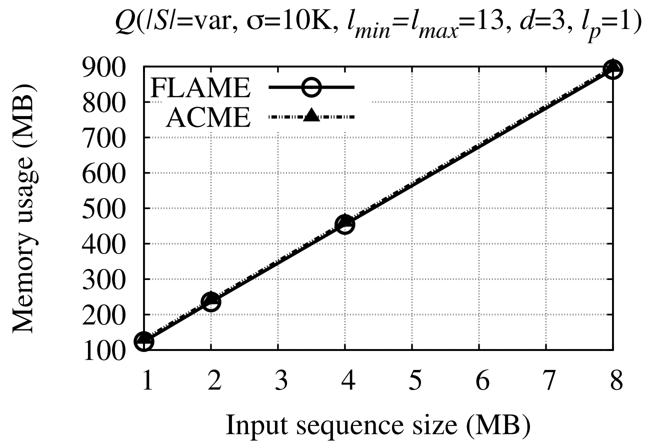
for fairness ACME uses only one core. The reported time includes the suffix tree construction and motif extraction time; the former is negligible compared to the extraction time. Note that we use small datasets (i.e., up to 8MB from DNA), because our competitors cannot handle larger inputs.

ACME is evaluated against MADMX and VARUN when extracting maximal motifs. Different similarity measures are utilized by ACME, MADMX and VARUN. Therefore, this experiment does not allow mismatches (i.e., $d = 0$) in order to produce the same results. Since the workload increases proportionally to the distance threshold, this experiment is relatively of light workload. Figure 4.6 shows that ACME is at least one order of magnitude faster than VARUN and two orders of magnitude faster than MADMX. Surprisingly, VARUN breaks while handling sequences longer than 1MB for this query, despite the fact that the machine has plenty of RAM (i.e., 192GB). We were not able to test the scalability of VARUN and MADMX in terms of alphabet size because they support DNA sequences only.

FLAME and ACME produce identical exact-length motifs. The serial execution of ACME significantly outperforms FLAME with increasing workload, as illustrated in Figure 4.8. We vary the workload by increasing motif length (Figure 4.7(a)), sequence size (Figure 4.7(b)), frequency threshold (Figure 4.8(a)), and alphabet size (Figure 4.8(b)). The impressive performance of our system is a result of ACME's cache efficiency. Note that, if we were to allow ACME to utilize all cores, then it would be one more order of magnitude faster. For example, we tested the query of Figure 4.7(a) when motif length is 12: FLAME needs 4 hours, whereas parallel ACME finishes in 7 minutes. The alphabet size experiment was run using synthetic datasets, generated with random distribution to guarantee comparative workloads between sequences of different alphabets.



(a) ACME vs MADMX and VARUN



(b) ACME vs FLAME

Figure 4.9: Memory usage of ACME compared to VARUN and MADMX (for query in Figure 4.6) and FLAME (for query in Figure 4.7(b)).

Memory usage

The memory usage of ACME grows linearly with respect to the sequence size. This is because the main factor is the suffix tree index; its implementation is not optimized since it is not a contribution of ACME. Figure 4.9(a) shows that the memory footprint of VARUN is three times higher than ACME's. MADMX has a low and constant memory footprint for sequences over 0.5MB but at the expense of 2 orders of magnitude higher runtime (see Figure 4.6). Figure 4.9(b) shows that FLAME and ACME have the same memory footprint because they share the same suffix tree implementation. Yet, ACME performs better than FLAME because of our cache-efficient approach, CAST (see Figure 4.7(b)).

Chapter 5

High Degree of Parallelism

We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not.

Tim Mattson
1967 CE — PRESENT

This section presents FAST[1], our efficient parallel space traversal approach that scales to thousands of CPUs[2]. FAST achieves high degree of concurrency by partitioning the search space horizontally and balancing the workload among CPUs with minimal communication overhead. Given this improvement, we are able to support *supermaximal* motifs with minimal overhead. Previously, motif extractors that support supermaximal motifs did not exist because the straightforward calculation would require keeping track of all maximal motifs.

[1]FAST stands for Fine-grained Adaptive Sub-Tasks

[2]For simplicity, the discussion assumes that each CPU executes a single process. In practice, our implementation executes one process per *core*.

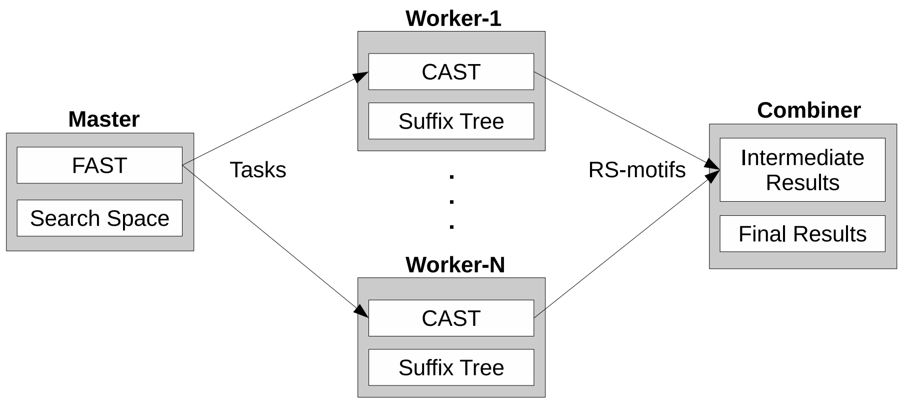


Figure 5.1: Architecture of parallel ACME.

5.1 Background

5.1.1 Parallel Computation Paradigm

We adopt the master-worker architecture shown in Figure 5.1. Given $\mathcal{C}$ CPUs, one master generates tasks; $\mathcal{C} - 2$ workers request tasks from the master and generate right-supermaximal motifs; and one combiner receives the intermediate results from the workers and extracts supermaximal motifs. The details are explained below.

Master. First the master reads the sequence from the disk and sends it to the workers. Then it decomposes the search space and starts generating tasks. The decomposition of the search space is based on horizontal partitioning of the search space to independent tasks. The goal is to utilize efficiently the available CPUs. Given a decomposition, the master uses our FAST technique, discussed in Section 5.2.2, to generate tasks. Tasks are scheduled dynamically using pull requests from workers.

Worker. Each worker receives the input sequence and constructs the annotated suffix tree (see Figure 4.3). Every worker needs access to the entire suffix tree, because occurrences of a motif candidate can occur at different branches. Once the index is ready, the worker requests a task from the master. Tasks are processed using our CAST technique to find right-supermaximal motifs; refer to Section 4.2 for details. The right-supermaximal motifs from each task are sent to the combiner in batches. Results within a batch share the same prefix; therefore, the prefix is stripped to better utilize the limited buffer space and minimize communication cost. When a worker is free it requests a new task from the master. This simple scheduling scheme allows workers to enter or leave the system anytime.

Combiner. The combiner implements Algorithm 3: it receives right-supermaximal motifs from all workers and uses a trie (see Figure 5.4(c)) to generate the final result, that is the set of supermaximal motifs. We will show in the experimental evaluation that the workload of the combiner is minimal, compared to the workers'. Therefore, the combiner is not a bottleneck and does not affect the scalability of the system.

5.2 Horizontal Search Space Partitioning

The search space depicted in Figure 4.2 can be split into independent sub-tries. Parallelizing the trie traversal is easy in this sense. However, the motif extraction search space is pruned at different levels during the traversal and validation process. Therefore, the workload of each sub-trie is not known in advance. The absence of such knowledge makes load balancing challenging to achieve. Imbalanced workload affects the efficiency of parallel systems due to underutilized resources.

FAST decomposes the search space into a large number of independent sub-tries. Our target is to provide enough sub-tries per CPU to utilize all computing resources with minimal idle time. We partition horizontally the search space at a

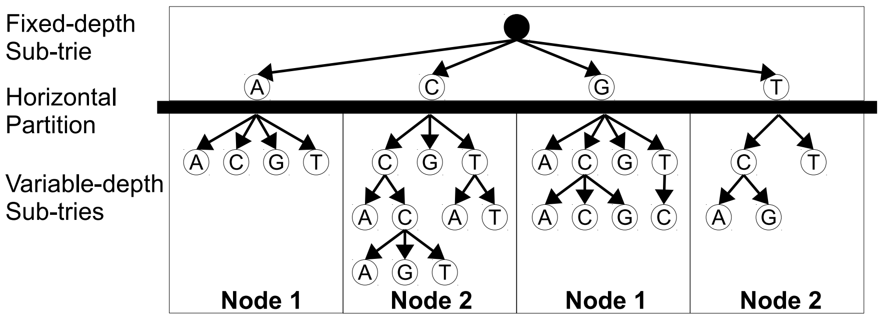


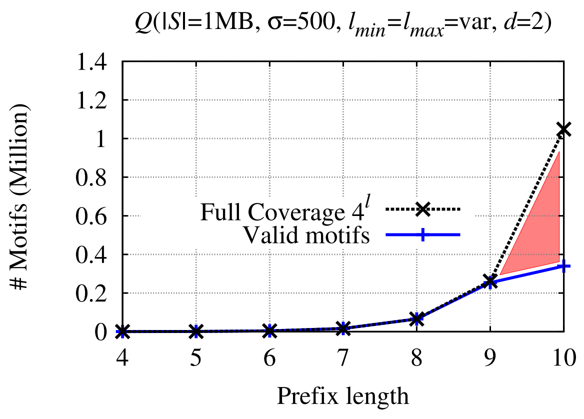
Figure 5.2: Combinatorial trie partitioned at depth $l_p = 1$ into a fixed-depth sub-trie leading to four variable-depth sub-tries, which are traversed simultaneously by two compute nodes.

certain depth l_p, into a fixed-depth sub-trie and a set of variable-depth sub-tries, as shown in Figure 5.2; observe that l_p corresponds to the *prefix length*. Since the search space is a combinatorial trie, there are $|\Sigma|^{l_p}$ sub-tries. The variable-depth sub-tries are of arbitrary size and shape due to the pruning of motif candidates at different levels.

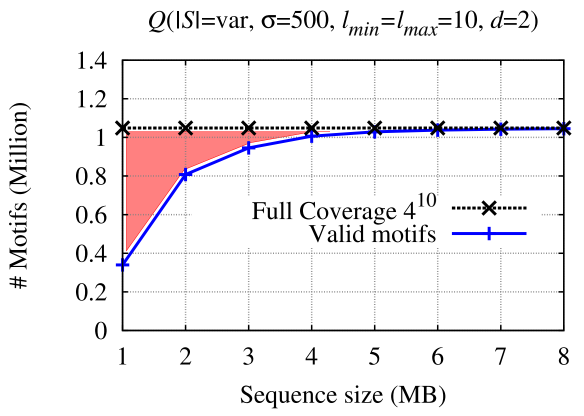
Example. Consider the search space for extracting motifs of length exactly 15 from a DNA sequence ($|\Sigma| = 4$). The search space trie consists of 4^{15} different branches, where each branch is a motif candidate of length 15. If we choose to set our horizontal partition at depth 2, our prefixes will be of length 2 and there are 4^2=16 large variable-depth sub-tries. Each sub-trie consists of 4^{13} branches (more than 67 million). If the horizontal partition cuts at depth 8, then there are 4^8=65,536 independent and small variable-depth sub-tries with 16 thousand branches each. □

5.2.1 Prefix length trade-off

The fixed-depth sub-trie indexes a set of fixed-length prefixes. Each prefix is extended independently to recover a set of motif candidates sharing this prefix. A



(a) Variable motif length



(b) Variable sequence size

Figure 5.3: Search space coverage in a DNA sequence. The shaded regions emphasize false positive prefixes, which increase by increasing the prefix length and decrease by increasing the input sequence size.

false positive prefix is a prefix of a set of false positive candidates, which would be pruned if a shorter prefix was used. For example, let $|\Sigma| = 4$ and let `AA` be a prefix that leads to no valid motifs. Using a prefix length of 5 (i.e., horizontal partitioning at depth 5) introduces 64 false positive prefixes that start with `AA`. Therefore, although the longer prefix increases the number of tasks (i.e., increases the degree of concurrency), the resulting false positive prefixes introduce useless overhead and, consequently, suboptimal utilization of resources.

Observation 1. *Given distance threshold d, all prefixes of length d are valid (i.e., cannot be pruned earlier).*

Let S be the input sequence. Any subsequence of S of length l will not exceed the distance threshold d for all search space branches of length l as long as $l \leq d$. For example, if a user allows up to 4 mismatches between a motif candidate and its occurrences, then any subsequence of length 4 from the input sequence is a valid occurrence of any prefix of length 4 in the search space. Observation 1 means that no pruning can be done until depth d of the search space, assuming the frequency threshold is met. We say that the search space is *fully covered* at depth d. Figure 5.3(a) shows an experiment where prefixes of length up to 9 symbols are fully covered although the sequence size is only 1MB. In this experiment, the prefix of length 10 leads to more than 0.5M false positive prefixes, that is, useless tasks that will be processed.

Observation 2. *As the input sequence size increases, the depth of the search space with full coverage increases.*

A longer sequence over a certain alphabet Σ means more repetitions of subsequences. Therefore, the probability of finding occurrences for motif candidates increases. Our experiments show that, even for a relatively small input sequence, the search space can be fully covered to depths beyond the distance threshold. Figure 5.3(b) shows an experiment where the number of false positive prefixes generated at $l_p = 10$ in the 1MB sequence decreases by increasing the sequence size.

Observation 3. *If the search space is horizontally partitioned at depth l_p, where the average number of sub-tries per CPU leads to high resource utilization, then a longer prefix is not desirable to avoid the overhead of false positives.*

5.2.2 FAST algorithm

FAST generates enough independent tasks per CPU, to maximize the utilization of CPUs. A task consists of one or more sub-tries and is transferred in a fixed-length compact form. The master horizontally partitions the search space and schedules tasks as shown in Algorithm 2. Function GetPrefixLength in line 1 calculates the near-optimal prefix length that will achieve the best load balance. At this point, we use Observations 1 and 3 to choose the prefix length that creates enough small tasks to occupy our cores. This function evaluates the equation $l_p = \lceil \log_{|\Sigma|}(\mathcal{K} \times \mathcal{C}) \rceil$, where $\mathcal{K}$ is empirically found to be 32. The exact-length prefixes are generated by depth first traversal of the fixed depth sub-trie. An iterator is used to recover these prefixes by a loop that goes over all combinations of length l_p from Σ. The master process is idle as long as all workers are busy. Algorithm 2 is lightweight compared to the extraction process carried out by workers. Hence, parallelizing the prefix generation does not lead to any significant speedup.

5.3 Supermaximal Motifs

Supermaximal motifs are those that are not contained in any other motif. They are very useful in practice, since they provide a compact and comprehensive representation of the set of all motifs. However, naïve methods that compute supermaximal motifs require to maintain the complete set of maximal ones [69]. The set of maximal motifs is prohibitively large for typical inputs and queries, and the verification process is computationally very expensive. For this reason, none of the existing systems supports supermaximal motifs.

Below, we propose a novel algorithm for extracting supermaximal motifs without storing the complete set of intermediate results. In the experimental section

Input: Alphabet Σ, Number of CPUs $\mathcal{C}$, Task size factor λ
Output: Generate and schedule tasks

```
   // Calculate optimal prefix length
1  $l_p \leftarrow$ GetPrefixLength($\Sigma$,$\mathcal{C}$)
2  $i \leftarrow 0$ // An iterator over all prefixes
   // Calculate task size
3  $t \leftarrow \lfloor \lambda |\Sigma|^{l_p} / \mathcal{C} \rfloor$ // one task contains $t$ prefixes
   // Assign tasks
4  while $i \neq$ prefixes end do
5      task $\leftarrow$ GetNextPrefix($i, t$)
6      WaitForWorkRequest( )
7      SendToRequester(task)
8      $i \leftarrow i + t$
   // Signal workers to end
9  while worker exist do
10     WaitForWorkRequest( )
11     SendToRequester(end)
```

Algorithm 2: Problem decomposition and scheduling (FAST)

we will show that our algorithm poses minimal overhead, compared to existing methods that only find maximal motifs. Our solution is based on the following observations:

Observation 4. *Let $\alpha m \beta$ be a supermaximal motif. The set of maximal motifs M may contain motifs $\{\alpha,\ \alpha m,\ \alpha m \beta,\ m,\ m\beta,\ \beta\}$.*

Observation 5. *The set of supermaximal motifs M_s does not contain prefixes $M_{pre} = \{\alpha, \alpha m\}$, or suffixes $M_{suf} = \{m\beta, \beta\}$, or subsequences $M_{sub} = \{m\}$.*

Example. Let the set of maximal motifs for a certain query be $M = \{\texttt{AGTT}, \texttt{GTT}, \texttt{TT}, \texttt{AGT}, \texttt{AG}, \texttt{GT}, \texttt{CTT}, \texttt{CT}\}$. To find the set of supermaximal motifs M_s, we have to eliminate maximal motifs that are subsequences of other ones. According to our observations, (*i*) $M_{pre} = \{\texttt{AGT}, \texttt{AG}, \texttt{CT}\}$, (*ii*) $M_{suf} = \{\texttt{GTT}, \texttt{TT}\}$, and (*iii*) $M_{sub} = \{\texttt{CT}, \texttt{GT}\}$. Therefore, $M_s = \{\texttt{AGTT}, \texttt{CTT}\}$. □

```
Input: Empty trie
Output: Supermaximal motifs
1 while Workers Exist do
2     buffer ← ReceiveFromWorker()
3     foreach motif in buffer do
4         reversed ← Reverse(motif)
5         InsertInTrie(reversed)
6 SpellTrieFromLeaves()
```

Algorithm 3: Extraction of supermaximal motifs

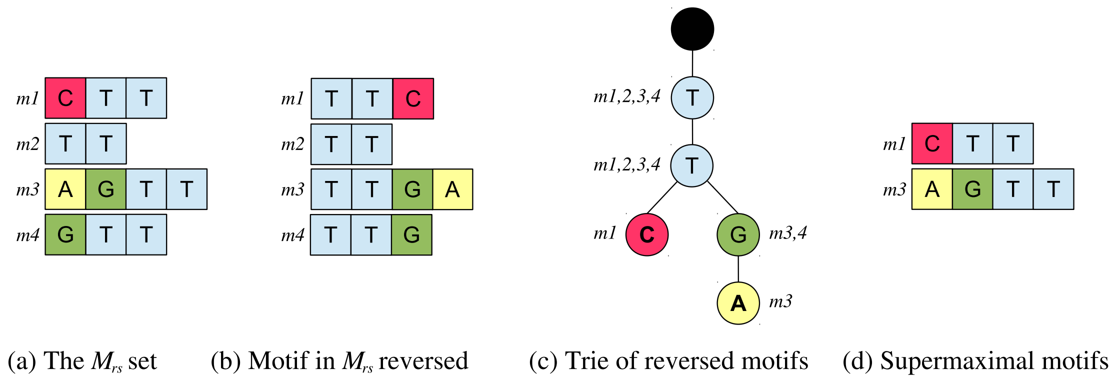


Figure 5.4: The steps for extracting the set of supermaximal motifs M_s from the set of intermediate results M_{rs}.

During the depth-first traversal of the search space, motifs that share the same prefix are grouped in the same sub-trie (see Figure 4.2); hence, we are able to easily filter motifs that are prefixes of other ones. The longest valid branches represent the set of maximal motifs that do not belong to M_{pre} or M_{sub}. We refer to this set as the right-supermaximal [18] set $M_{rs} = M_s \cup M_{suf}$. In our example, $M_{rs} = \{\texttt{AGTT}, \texttt{CTT}, \texttt{GTT}, \texttt{TT}\}$. Now, we can find the supermaximal motifs by discarding all proper suffixes from M_{rs}. However, computing M_{suf} is challenging, because motifs in M_{rs} belong to different parts of the search space as they start with different prefixes. A naïve solution would check all possible pairs in M_{rs}; the complexity of such a solution is $\mathcal{O}(|M_{rs}|^2)$.

We propose Algorithm 3, which in the average case removes redundant suffixes in $\mathcal{O}(|M_{rs}| \log_{|\Sigma|} |M_{rs}|)$ time. The algorithm reverses the contents of M_{rs}, effectively transforming the problem from suffix- to prefix-removal. As we mentioned earlier, the latter can be solved efficiently by utilizing a trie.

Figure 5.4(a) depicts the M_{rs} set for our running example; the conceptually reversed motifs are shown in Figure 5.4(b). The reversed motifs are inserted in a trie that is shown in Figure 5.4(c); observe that common prefixes are grouped together. In the trie, each path from the root to a leaf corresponds to a string that is not a prefix of any other. Our example trie has two leaves. After reversing back the corresponding paths, the final set $M_s = \{\texttt{AGTT}, \texttt{CTT}\}$ of supermaximal motifs is shown in Figure 5.4(d).

5.3.1 Proof for Supermaximal Motifs Algorithm

The set of supermaximal motifs from Algorithm 3 is correct and complete. Let SpellTrieFromLeaves be a function that takes as input a trie and returns the paths from the root to each leaf. This is easily achieved using a depth-first traversal of the trie.

Lemma 1. SpellTrieFromLeaves *returns all sequences in the input trie that are not proper prefixes of any other.*

Proof. By construction of the trie. □

We produce M_s in two steps: (i) we produce a right-supermaximal set of sequences M_{rs} by calling SpellTrieFromLeaves on the pruned search space, and (ii) we pass M_{rs} to Algorithm 3, whose job is to eliminate proper suffixes from M_{rs} to produce M_s. The outline of the proof is as follows: first we prove that SpellTrieFromLeaves eliminates proper prefixes from an input set of sequences, then we show that SpellTrieFromLeaves can be used to eliminate proper suffixes, after that we show that Algorithm 3, when given a right-supermaximal set of motifs M_{rs}, produces the supermaximal motifs set. We conclude our proof by showing that the input we give to Algorithm 3 is indeed right-supermaximal.

SpellTrieFromLeaves can be used to remove proper suffixes from a set of sequences. When sequences are reversed, proper suffixes become proper prefixes, so it follows from Lemma 1 that SpellTrieFromLeaves can be used to remove proper suffixes from a set of sequences when it is called on a trie construed with a set M_{rev} of reversed sequences, where $s \in M_{rev}$ if $s_{rev} \in M$.

When the input to Algorithm 3 is a right- supermaximal set of sequences M_{rs}, its output is a supermaximal set of sequences M_s. Algorithm 3 removes proper suffixes from a set of sequences using SpellTrieFromLeaves and the input set M_{rs} does not have sequences that are proper prefixes or proper subsequences of other sequences (proof in next paragraph), which means that the output set M_s does not have sequences that are proper prefixes, proper subsequences, or proper suffixes of other sequences in M_s, that is, M_s is a supermaximal set of motifs.

In this paragraph we show that the input to Algorithm 3 is a right-supermaximal set of sequences. This set is produced by calling SpellTrieFromLeaves on the pruned search space trie that has all the valid motifs. It follows from the discussion in Section 4.1 that if a sequence is a valid motif, all its subsequences, including all its proper suffixes are valid motifs. We use this, together with Lemma 1 to show that calling SpellTrieFromLeaves produces the right-supermaximal set of motifs M_{rs}, i.e. if a motif m is in M_{rs}, any other motif in M_{rs} is neither a proper prefix (follows directly from Lemma 1) nor a proper subsequence of m. We show next that if a sequence is in M_{rs} it is not a proper subsequence of any other string in M_{rs}. Assume m is a valid motif in M_{rs}, and m_{sub} is a proper subsequence of m. m_{sub} is

Table 5.1: Specifications of the systems used to evaluate FAST.

#	**Architecture**	**Cores**	**RAM**	**Cache**		
				L1	**L2**	**L3**
1	64-bit Linux	12 @2.67GHz	192GB	64KB	256KB	12MB
2	64-bit Linux SMP	32 @2.27GHz	624GB	64KB	256KB	24MB
3	IBM Blue Gene/P supercomputer	16,384 × 4 @850MHz	4GB each 64TB total	64KB	2KB	8MB
4	64-bit HPC Linux cluster	480 @2.1GHz	6GB each 3TB total	128KB	512KB	5MB

a proper prefix of some other sequence that is a proper suffix of m, and all proper suffixes of m are in the pruned search space trie, so it follows from Lemma 1 that m_{sub} can not be in M_{rs}.

5.4 Evaluation

Table 5.1 shows the specifications of the machines used to evaluate FAST. The parallel scalability experiments were run on *System#4* and *System#3* and the comprehensive motif extraction support experiments were run on *System#1* and *System#2*. The sequences used in these experiments are from the human genome DNA sequence and protein sequences. The size is noted in each experiment along with the other parameters, namely, frequency, length, distance, and prefix length.

5.4.1 Parallel scalability

This section investigates ACME's parallel scalability. We test the so-called strong-scalability, where the number of cores is increased while the problem size is fixed. We first compare ACME against PSmile, the only parallel competitor. PSmile uses grid-specific libraries to parallelize a previous sequential motif extraction algorithm. Speedup efficiency (SE) is a metric for measuring resource utilization by

Table 5.2: Scalability of PSmile on *System#4* using the DNA dataset. The speedup efficiency of PSmile is hindered by load imbalance due to improper search space partitioning and static scheduling ($SE < 0.8$ is considered low).

$Q(|S|$=32MB, σ=10K, l_{min}=10, l_{max}=15, d=3)

	Time (sec)		**Speedup Efficiency**	
Cores	**PSmile**	**ACME**	**PSmile**	**ACME**
5	19,972	18,883	1.00	1.00
10	9,894	8,476	0.90	0.99
20	4,869	3,978	0.86	0.99
40	2,786	1,969	0.74	0.98
80	1,787	989	0.57	0.97
160	1,130	580	0.44	0.82

dividing serial time over parallel time multiplied by number of cores. Calculating SE from the experiments reported in the PSmile paper, the utilization drops to 0.72 when using 4 nodes only. In practice $SE < 0.8$ is considered low. We suspected that the bad performance was partially due to inefficient implementation. For fairness we implemented the search space partitioning and task scheduling scheme of PSmile within ACME, utilizing our cache-efficient trie traversal algorithms.

Table 5.2 shows the results, using our optimized implementation of PSmile. The experiment was run on *System#4* (refer to Table 5.1). Due to resource management restrictions, the minimum number of cores used in this experiment was 5; therefore, speedup efficiency is calculated relative to a 5-core system. PSmile does not scale efficiently not even on 40 cores, due to problematic space partitioning and scheduling, which creates load imbalance. In contrast, for this particular query, ACME scales easily to more than 160 cores.

The next experiment investigates ACME's scalability to the extreme, by utilizing up to 16,384 cores on a supercomputer. We use the Protein dataset, which results to a much larger search space than DNA because of the larger alphabet (i.e.,

Table 5.3: Scalability of ACME on a supercomputer for the Protein dataset. ACME scales to tens of thousands of cores with high speedup efficiency.

$Q(|S|$=32MB, σ=30K, l_{min}=12, l_{max}=∞, d=3)

Cores	**Time (Hrs.)**	**Speedup Efficiency**
256	19.83	1.00
1,024	4.97	0.99
2,048	2.51	0.98
4,096	1.29	0.96
8,192	0.68	0.91
16,384	0.31	0.98

20 symbols). With larger alphabets, a small prefix length creates enough tasks; in this case $l_p = 5$. The experiment was run on *System#3*. Due to resource management restrictions, the minimum number of cores used in this experiment was 256 cores; hence, speedup efficiency is calculated relatively to a 256-core system. The results are shown in Table 5.3 and demonstrate the excellent scalability of ACME to thousands of cores. On 256 cores, the query takes almost 20 hours to execute; whereas with 16,384 cores it finishes in only 18.6min, achieving almost perfect (i.e., 0.98) speedup efficiency. It is worth mentioning that the same query on a high-end 12-core workstation (i.e., *System#1*) takes more than 7 days. Recall that each core of the workstation is much faster (i.e., 2.67GHz) than a supercomputer core (i.e., 850MHz).

5.4.2 Comprehensive motif extraction support

In addition to supermaximal motifs, ACME extracts maximal and exact-length ones. The following paragraphs evaluate ACME's scalability in terms of input size and query complexity; and compare ACME against state of the art systems for maximal and exact-length motifs.

Table 5.4: Supermaximal Motifs from the complete DNA for the human genome (2.6GB) categorized by length. The total number of supermaximal motifs is more than total number of exact-length motifs.

$Q(|S|$=2.6GB, σ=500K, l_{min}=15, l_{max}=var, d=3)

Supermaximal ($l_{max} = \infty$)						**Exact-length** ($l_{max} = l_{min}$)	
Len	Count	Len	Count	Len	Count	Len	Count
15	359,293	20	30,939	25	443	15	446,344
16	82,813	21	33,702	26	143		
17	22,314	22	12,793	27	37		
18	7,579	23	5,289	28	2		
19	2,288	24	2,435				
		Total	560,070			Total	446,344

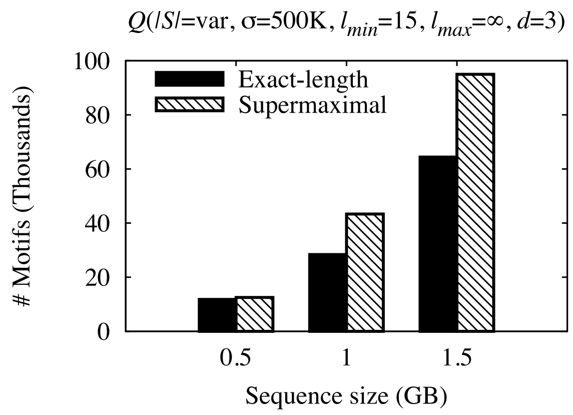
Table 5.5: Analysis of three sequences of different alphabets, each of size 1GB.

	Query	**Motifs**	**Longest**	**Time**
DNA	σ=500K, l=12$-\infty$, d=2	5,937	20	0.6 m
Protein	σ=30K, l=12$-\infty$, d=1	96,806	95	2.1 m
English	σ=10K, l=12$-\infty$, d=1	315,732	42	3.5 m

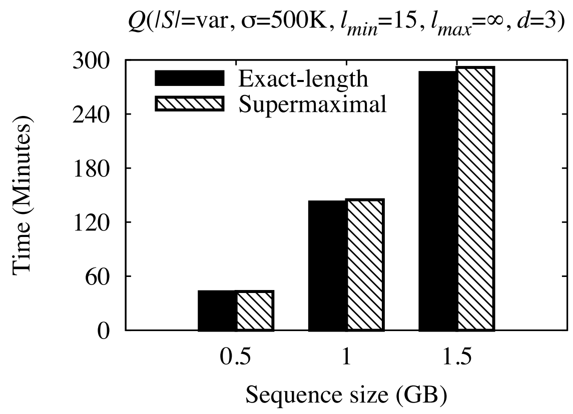
Table 5.6: The overhead of extracting supermaximal motifs over maximal motifs is not critical due to ACME's pipelined strategy for filtering motifs that are subsequences of others.

$Q(|S|$=1.5GB, σ=500K, l_{min}=15, l_{max}=∞ d=3)

	Time	**Motifs**
Maximal motifs	303.7 min	144,952
Supermaximal	313.7 min	87,680
Difference	10 min	57,272
Percentage	3.3%	39.5%



(a) Number of motifs



(b) Time performance

Figure 5.5: Supermaximal vs Exact-length motifs extraction using ACME.

Gigabyte-long sequences and varying alphabets

The experiments discussed in this section were run on *System#1* and *System#2* (refer to Table 5.1). Table 5.4 shows the count of all supermaximal motifs (i.e., no bound for l_{max}), grouped by length, that appear at least $\sigma = 500K$ times in the entire human DNA (i.e., 2.6GB). For reference, the count of all *maximal* motifs with length 15 is also shown. The longest supermaximal motif is 28 symbols long. This means that the CAST array size did not exceed 252 bytes in a 32-bit system (28 elements of 9 bytes each). With current CPU cache sizes, not only the CAST array will fit in the cache but most probably the occurrences array too. Consequently, ACME handles the extra workload of extracting maximal and supermaximal motifs efficiently.

Observe that, in the entire human genome, there are around 20% more supermaximal motifs of length 15 and more, compared to the number of motifs with exact length 15. Figure 5.5(a) shows the corresponding counts by varying the size of the input sequence (i.e., using prefixes of the entire DNA). The number of supermaximal motifs is in all cases significantly more than the exact-length ones. Figure 5.5(b) compares the time to extract exact-length versus all supermaximal motifs. The difference is negligible (i.e., around 4%), confirming the efficiency of our supermaximal extraction algorithm.

ACME supports different alphabet sizes. Table 5.5 shows the results of extracting supermaximal motifs from 1GB sequences of different alphabets. Notice that the number of extracted motifs increases with respect to the alphabet size. This is because given approximate matches, more motifs are discovered. The lengths of these motifs are however not related to the alphabet size but to the nature of the underlying sequences. For instance, it is difficult to have very long motifs in English but not in Protein. We also extracted maximal and supermaximal motifs from 1.5GB of the DNA sequence. Table 5.6 shows that ACME's pipelined strategy for filtering motifs that are subsequences of others introduces an overhead of 3.3% over the maximal motifs extraction time. In this process, about 40% of the maximal motifs are discarded because they are subsequences of other ones.

Chapter 6

Automatic Tuning and Elasticity

Try to be a rainbow in
someone's cloud.

Maya Angelou
1928 — 2014 CE

This chapter introduces the automatic tuning feature that allows ACME to efficiently utilize thousands of cores on a supercomputer without user specifying decomposition or number of cores. The chapter also discusses our elasticity model that suggests the appropriate amount of cloud resources to rent while meeting the user's requirements in terms of processing time and financial cost. Finally, a generalization of our method to bag-of-tasks applications is discussed and evaluated using two different applications.

6.1 Background

The goal of automatic tuning is to find a good decomposition of the search space (i.e., parameter l_p) that minimizes runtime, while achieving high utilization of computational resources. To minimize runtime, we need to utilize efficiently as many CPUs as possible, which translates to: (i) enough tasks per CPU, in order to achieve good load balance; and (ii) few false positives, in order to avoid useless work. As explained in the previous section these goals contradict each other. Therefore, we need to solve the following optimization problem:

Problem. *Find the value of parameter l_p that maximizes scalability (i.e., number of CPUs) under the constraint that speedup efficiency $SE \geq SE_{min}$.*

Let $\mathcal{C}$ be the number of workers, T_1 is the time to execute the query using one worker (i.e., serial execution) and $T_{\mathcal{C}}$ is the time to execute the query using $\mathcal{C}$ workers. For example, if a query executes serially in 10 minutes, then optimally it would execute in 1 minute using 10 cores. *Speedup efficiency* is defined as:

$$SE = \frac{T_1}{\mathcal{C} \cdot T_{\mathcal{C}}} \tag{6.1}$$

In practice there are always overheads, therefore we require $SE \geq SE_{min}$, where SE_{min} is a user-defined threshold. Typically, $SE_{min} = 0.8$ is considered good in practice.

Since the processing time of each task is not known in advance, it is difficult to find an analytical solution for Problem 6.1; therefore our solution is based on heuristics. Note that the accuracy of the results is *not* affected; our algorithm will still return the correct and complete set of supermaximal motifs. If our heuristics fail to achieve optimal space decomposition, then only the execution time will be affected, due to sub-optimal utilization of computational resources.

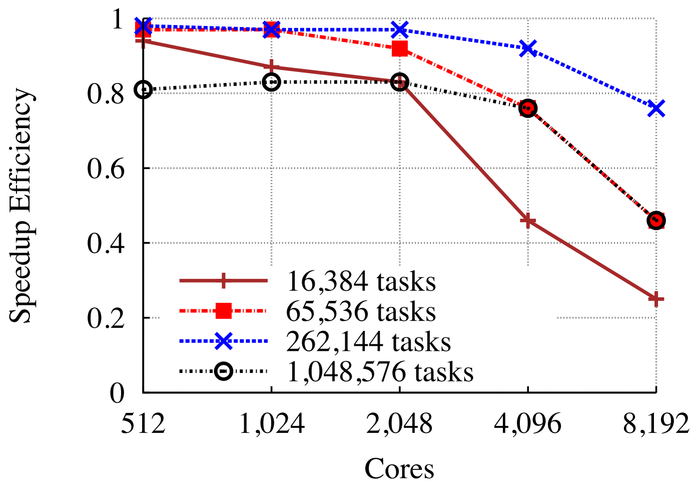


Figure 6.1: Speedup efficiency of ACME for a query on the human DNA. Each line corresponds to a different decomposition of the same query. Too few tasks (i.e., 16,384) cannot achieve load balance. Too many tasks (i.e., 1,048,576) result in useless computation. For this particular combination of dataset and query parameters, the optimal decomposition generates 262,144 tasks.

We ran ACME on an IBM BlueGene/P supercomputer for different decompositions, varying the number of cores; the results are shown in Figure 6.1. When there are too few tasks, the granularity is not fine enough to balance the workload. Hence, the program does not scale to more than 2,048 cores. Surprisingly, if the problem is decomposed into too many tasks, scalability again suffers. This happens because many CPUs performing useless computation by processing false positive branches of the tree. We will further investigate this issue in Section 6.3.2. The best speedup efficiency up to 8,192 cores is achieved for a moderate number of 262,144 tasks. Note that the optimal decomposition cannot be determined in advance because it depends on the input sequence and the query parameters (e.g., minimum frequent pattern support, or maximum errors allowed) [72].

Table 6.1: Example query Q running on 240 workers. For $l_p = 2$, we cannot achieve load balance. For $l_p = 4$, there are too many false positive tasks. The optimal search space decomposition is found using $l_p = 3$, achieving very good speedup efficiency $SE = 0.91$.

$Q(|S|\text{=32MB}, \sigma\text{=30K}, l_{min}\text{=7}, l_{max}\text{=}\infty, d\text{=2})$

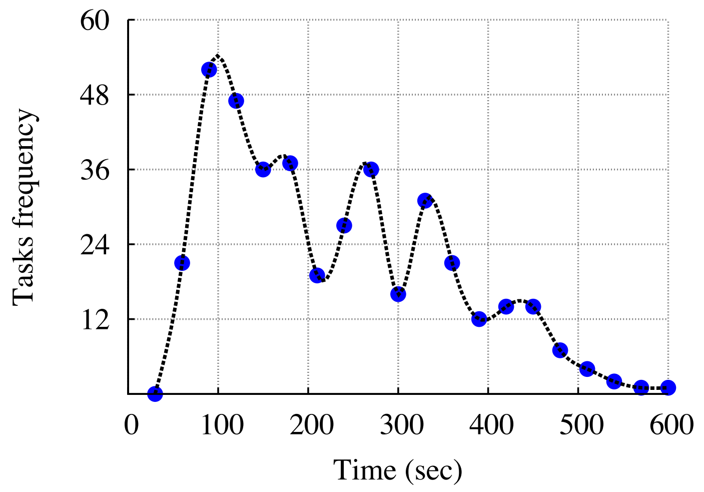
Prefix Length (l_p)	**Tasks ($\lvert\Sigma\rvert^{l_p}$)**	**Average Tasks/Worker**	**Speedup Efficiency**
2	400	1.66	0.47
3	8,000	33.33	0.91
4	160,000	666.66	0.22

6.1.1 Distribution of Workload Frequency

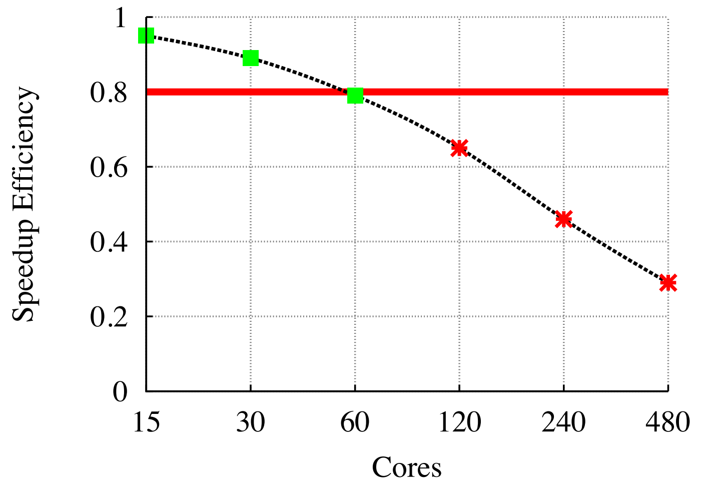
In the following, we will analyze the workloads of different decompositions of the same example query and their respective scalability. Table 6.1 shows basic statistics.

Let us start with prefix length $l_p = 2$ that decomposes the search space of Q into $20^2 = 400$ tasks (recall that the alphabet contains 20 symbols). We run each task on one CPU and measure its execution time. The results are shown in Figure 6.2(a), which represents the *workload frequency distribution* for the combination of Q and l_p. For a point (x, y), x represents execution time, whereas y represents the number of tasks that require time x to run. The total execution time for Q is given by the area under the curve.

The coarse decomposition of the search space leads to an irregular distribution with many "heavy" tasks. For instance, there are about 70 tasks that run in less than 100sec, but there are also around 130 tasks that need more than 300sec; some extreme cases need more than 500sec. Even with dynamic scheduling, balancing such a workload on a parallel system is challenging. We executed Q with varying number of CPUs and measured the speedup efficiency SE; the results are shown in Figure 6.2(b). Assuming the threshold for good speedup efficiency is $SE_{min} = 0.8$,



(a) Workload frequency distribution for 400 tasks



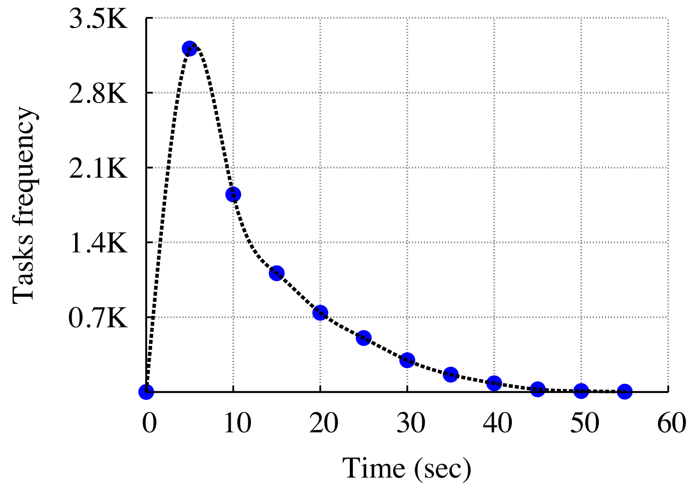
(b) Speedup efficiency as number of cores is varied

Figure 6.2: For example query Q and $l_p = 2$, the search space is decomposed to 400 large tasks. Load balancing is poor and the speedup efficiency drops when using more than 60 CPUs.

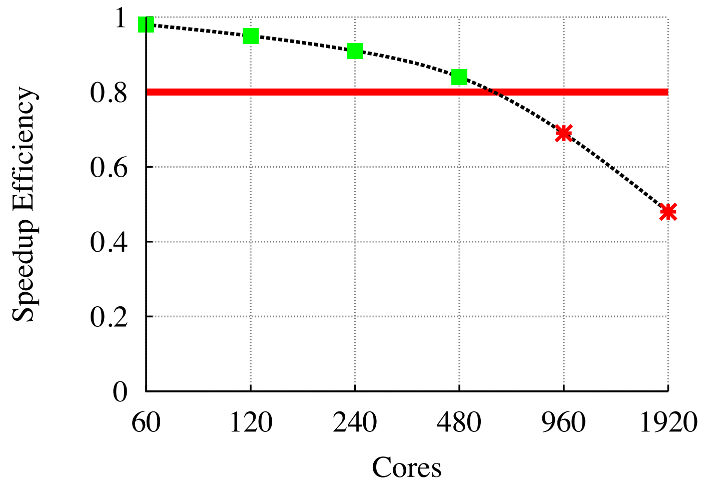
the figure shows that this particular decomposition does not allow Q to scale efficiently to more than 60 CPUs. Note that, if more than 60 CPUs are used the total execution time for Q will decrease, but due to load imbalance many CPUs will be underutilized, so computational resources will be wasted. In our experiment, if instead of 60 we use 480 CPUs (i.e., 8x increase), the total execution time will drop from 30min to 10min (i.e., only 3x improvement). This is the practical meaning of low SE values.

Our scheduling corresponds to an instance of the online dynamic bin packing problem. When items are few and large (i.e., coarse decomposition of search space), bins cannot be filled optimally. Intuitively, more and smaller objects are needed. This corresponds to a longer prefix length, resulting in a finer decomposition. We run again the same experiments for $l_p = 3$, which generates $20^3 = 8,000$ tasks. The workload frequency distribution is shown in Figure 6.3(a); it resembles a leptokurtic and positively skewed non-symmetric distribution. While we do not know the processing time of tasks beforehand, we expect their execution time to decrease monotonically as they are further decomposed. Indeed, the figure shows that the majority of tasks run in around 5sec, whereas very few need from 40 to 60sec. Consequently, there are enough small tasks to keep all CPUs busy while the few larger ones are executed. Moreover, the probability of a large task being executed last is low because there are only a few of them; therefore, we expect good load balance. Figure 6.3(b) shows the speedup efficiency for a varying number of CPUs. Observe that the algorithm scales well (i.e., $SE \geq 0.8$) up to about 500 CPUs, which is an order of magnitude more, compared to Figure 6.2(b).

It is tempting to generate an even finer search space decomposition in order to scale to more CPUs. Figure 6.4(a) shows the workload frequency distribution for $l_p = 4$. The graph resembles a power-law distribution. Out of the 160,000

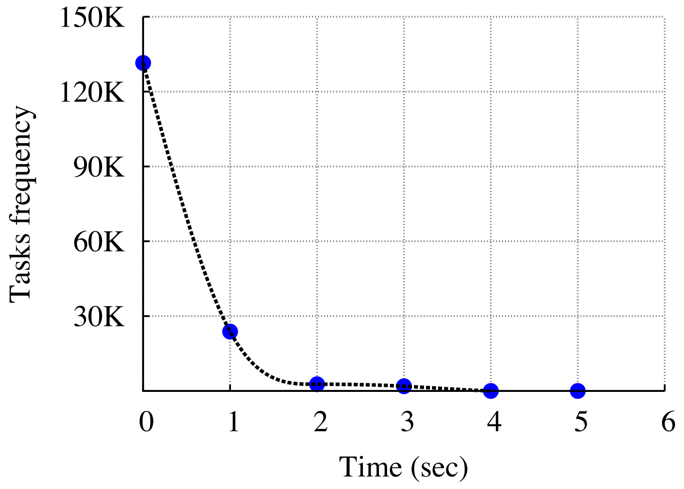


(a) Workload frequency distribution for 8,000 tasks

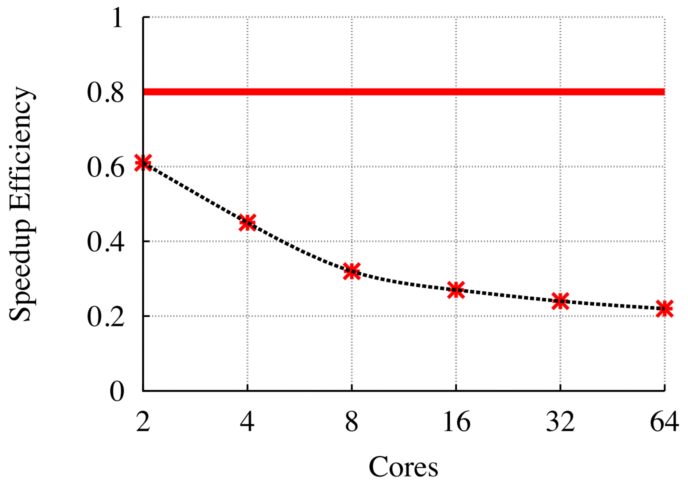


(b) Speedup efficiency as number of cores is varied

Figure 6.3: For example query Q and $l_p = 3$, the search space is decomposed to 8,000 tasks. Load balancing is near optimal and the speedup efficiency is high up to 500 CPUs.



(a) Workload frequency distribution for 160K tasks



(b) Speedup efficiency as number of cores is varied

Figure 6.4: For example query Q and $l_p = 4$, the search space is decomposed to 160,000 tasks. Since most tasks are false positive prefixes, speedup efficiency is poor even at 2 CPUs.

generated tasks, very few take 3 to 5sec, whereas the vast majority (i.e., around 130,000 tasks) are very small with execution time close to zero. Unfortunately, many of these tasks are false positives that generate useless work. Although the overhead per false positive task is small, because of their sheer number, the cumulative overhead is high. Figure 6.4(b) shows the speedup efficiency for a varying number of CPUs. SE is always less than SE_{min}; therefore, for this decomposition the system cannot scale efficiently not even on 2 CPUs.

6.2 Tuning ACME

6.2.1 Automatic Decomposition

We solve Problem 6.1 as follows: We partition the search space at a specific prefix length l_p and draw a random sample of tasks to run, in order to estimate the speedup efficiency SE. We repeat this process with different prefix lengths until we find the one that allows scaling to the largest number of CPUs with $SE \geq SE_{min}$. The initial decomposition parameter value affects efficiency, but not the final result.

Algorithm 4 describes the process. In line 1, l_p is initialized to the minimum motif length l_{min}. We start with the longest prefix length possible, that is the minimum length of valid motifs; then we try shorter prefixes. This way we arrive to the optimal l_p faster, because longer prefixes produce smaller tasks that run faster. If l_p is decremented to the distance value d without meeting the stopping criterion (see line 7), l_p is set to $d + 1$ and the algorithm terminates. This follows from Observation 1 in Section 5.2.1. To reduce the overhead of the automatic tuning process, sample prefixes can be generated and evaluated in parallel (i.e., lines 4 and 5). In practice, the main loop of the algorithm is executed only a few times before finding a near-optimal decomposition.

```
Input: Sequence $S$; query $Q(|S|, \sigma, l_{min}, l_{max}, d)$; threshold $SA_{min}$
Output: Prefix length $l_p$; number of CPUs $\mathcal{C}_{max}$
1  $l_p \leftarrow l_{min}$
2  $\mathcal{C}_{max} \leftarrow 1$
3  while $l_p > d$ do
       // randomly draw $x$ prefixes of length $l_p$
4      $sample \leftarrow$ RandomPrefixes($x$, $l_p$)
5      $sample\_times \leftarrow$ ExtractMotifs($sample$)
6      $t\mathcal{C} \leftarrow$ EstSpdupEff($sample\_times$, $SE_{max}$)
7      if $t\mathcal{C} < \mathcal{C}_{max}$ then
8          break
9      else
10         $l_p \leftarrow l_p - 1$
11         $\mathcal{C}_{max} \leftarrow t\mathcal{C}$
12 $l_p \leftarrow l_p + 1$
```

Algorithm 4: Automatic Tuning

Function EstSpdupEff in line 6 is the heart of the algorithm. Given a decomposition, for a specific number $\mathcal{C}$ of CPUs, it estimates the corresponding speedup efficiency. The function iterates over a range of values for $\mathcal{C}$ and returns the one that achieves the maximum SE for the given space decomposition. The following paragraphs explain how to estimate the serial (i.e., T_1) and parallel (i.e., $T_{\mathcal{C}}$) execution times, which are required by EstSpdupEff.

Estimating Serial Execution Time

From the previous analysis, it follows that the workload frequency distribution of a good space decomposition should be similar to the one in Figure 6.3(a). It should contain a lot of fairly small tasks, in order to achieve load balance, but should avoid very small ones, since they tend to be false positives. A Gamma distribution [73] is flexible enough to approximate different workload distributions.

A Gamma distribution Γ is characterized by a shape parameter α and a scale parameter β. Recall that line 4 of Algorithm 4 generates a sample of tasks for prefix length l_p. According to our heuristic, we assume that the sample approximates Γ. Therefore, we can use the sample to calculate approximations for the mean μ_Γ and standard deviation σ_Γ of Γ. Then, we calculate α and β as follows [73]:

$$\alpha = \frac{\mu_\Gamma^2}{\sigma_\Gamma^2}, \qquad \beta = \frac{\mu_\Gamma}{\alpha} \tag{6.2}$$

The probability density function (PDF) of Γ is defined as:

$$\Gamma(x;\alpha,\beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{(\alpha-1)!} \tag{6.3}$$

Let $\Lambda(t_i, t_j)$ be the expected number of tasks (in the entire space for a given l_p) with runtime between t_i and t_j. Given Γ, Λ is calculated as follows:

$$\Lambda(t_i, t_j) = |\Sigma|^{l_p} \int_{t_i}^{t_j} \Gamma(x;\alpha,\beta)\, dx \tag{6.4}$$

The serial execution time T_1 is the summation of the execution times of all tasks. The lower bound of runtime for a task is zero, but the upper bound is unknown. Let x be an integer time unit. Then T_1 is defined as:

$$T_1 = \sum_{x=0}^{\infty} \frac{2x+1}{2} \Lambda(x, x+1) \tag{6.5}$$

Estimating Parallel Execution Time

We employ the queuing theory [74] to estimate the parallel execution time $T_\mathcal{C}$. We model the motif extraction process as a finite-source queue of $|\Sigma|^{l_p}$ tasks served by $\mathcal{C}$ servers (i.e., CPUs). Without loss of generality, we assume homogeneous servers. Since our population is finite, numerically simulating the queue provides an accurate representation of the real system [75].

ACME implements a discrete event simulator. We start with all tasks in the queue as tasks are generated by simply enumerating prefixes. The workloads of the tasks follow the workload frequency distribution of our sample prefixes. Equation 6.4 is used to create bins of tasks. The servers randomly consume tasks from different workload bins until all bins are empty. The output of the simulator is our estimation for the parallel execution time $T_{\mathcal{C}}$.

6.2.2 Elastic Deployment

The elasticity model of ACME furnishes users with an accurate estimation of the minimum amount of resources required to process a query within specific constraints. User constraints may involve the maximum allowed execution time; maximum amount of CPU hours, if the system is deployed in a typical shared research computing infrastructure; or a limit on the financial cost, if a commercial cloud computing provider is used.

Algorithm 5 describes the elasticity model. It takes the execution time of each of the tasks in the random sample (see Section 6.2.1) and the user constraints as input, and outputs the number of CPUs to use, together with the estimated time and speedup efficiency. In line 5, a queue is setup by randomly taking tasks with workloads according to our probability density function. The execution of the query is simulated using a certain number of CPUs. This simulation is done in a loop where the number of CPUs is varied until the user constraints are met.

Given the expected performance variability on public clouds [76], users should be able to reevaluate the situation online and adapt accordingly. Our serial time estimation may be reevaluated at runtime to guide user decisions and meet their constraints. A slight modification of Equation 6.4 is used to account only for the

```
    Input: Sample times $sample\_t$, $user\_constraints$
    Output: Suggested number of cores $\mathcal{C}_p$, estimated parallel time $T_{\mathcal{C}}$
    // estimate PDF from sample execution times
1   $\alpha \leftarrow (\textsc{Mean}(sample\_t)/\textsc{StDev}(sample\_t))^2$
2   $\beta \leftarrow \textsc{Mean}(sample\_t)/\alpha$
    // predict serial time left
3   $T_1 \leftarrow \sum_{t=0}^{\infty} (\frac{2t+1}{2}\Lambda(t, t+1))$
    // predict parallel time and utilization
4   while $user\_constraints \neq$ TRUE do
5       $\textsc{SetupQueue}(user\_constraints, \mathcal{C}_p)$
6       $(T_{\mathcal{C}}, \mathcal{C}_p) \leftarrow \textsc{SimulateQueue}(sample\_t)$
```

Algorithm 5: Elasticity

tasks not executed yet. We substitute $|\Sigma|^{l_p}$ with $|\Sigma|^{l_p} - k$, where k is the number of already completed tasks.

The output of our model can be used in many ways. For example, if the pricing scheme of a cloud computing provider is given, our model can predict accurately the expected financial cost. We evaluate such a case in Section 6.4.

6.3 Generalization to Bag-of-Tasks Applications

Many scientific and commercial applications, such as chemoinformatics [77] and bioinformatics [78], are designed to leverage large scale parallel computing infrastructures. In bag-of-tasks applications [79], a large computational problem is decomposed into many (i.e., thousands or millions) loosely coupled tasks, which are executed on many CPUs on a private cluster, a supercomputer, or a commercial cloud. Typically, users request computational resources according to their budget. In a research environment, budget is an awarded amount of core-hours, whereas budget represents actual financial cost on a commercial cloud. In both cases, there

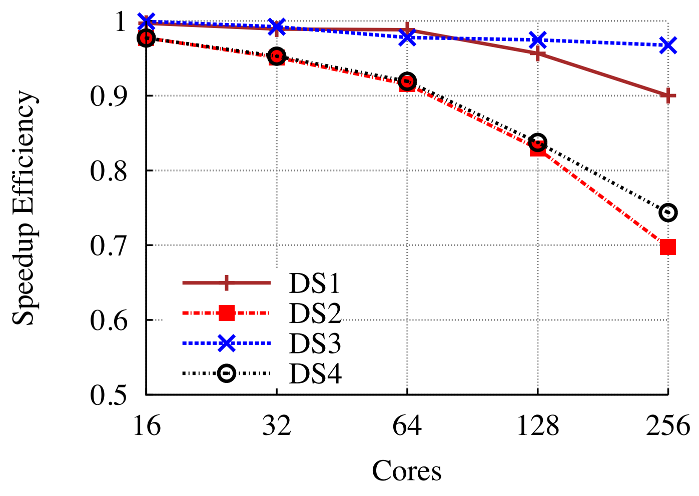


Figure 6.5: Speedup efficiency of VinaLC using four subsets, DS1 to DS4, of 2,000 lead-like compounds each (i.e., same dataset and same size). DS3 scales very well to 256 cores and beyond. On the other hand, assuming SE_{min}=0.8, DS2 does not scale efficiently to more than 128 cores.

is an incentive to efficiently utilize the computational resources.

Users predict the expected scalability of a program by consulting studies on typical workloads [80]. Consider the following example from computational chemistry: VinaLC [77] computes dockings between molecules. The developers state that VinaLC "scales up to more than 15K CPUs" [77], which is true for their dataset and computing infrastructure. We extracted four subsets, DS1 to DS4, each containing 2,000 lead-like compounds[1] from the ZINC dataset. Each compound was docked against the Thermus thermophilus gyrase B complex. We executed VinaLC on a local Linux cluster by varying the number of cores; the resulting speedup efficiencies are shown in Figure 6.5.

Interestingly, although all subsets come from the same dataset and contain the same number of compounds, the execution times of their tasks vary significantly. On the one hand, DS3 achieves excellent speedup efficiency for 256 cores and

[1]See Section 6.4 for details about the experimental settings and datasets.

beyond. Therefore, it is advisable to employ more cores in order to finish execution faster. On the other hand, the speedup efficiency of DS2 drops below 0.8 (i.e., our cutoff point) after 128 cores; using more cores would waste resources. This experiment demonstrates that the scalability study in the VinaLC paper is too generic for accurate predictions in practical usage scenarios.

6.3.1 Scientific Applications

We consider two representative scientific applications for molecular docking and sequence alignment. The chosen parallel systems are *VinaLC* [77] and *P-SSW* (an MPI version of SSW [81]). Unlike ACME, VinaLC and P-SSW handle static task decomposition problems.

Molecular Docking: VinaLC

Molecular docking predicts the structural orientation in which two chemical molecules bind to make a stable complex [82]. It is used to discover new drugs and assist in drugs repositioning [83]. There are about 30,000 genes in the human genome that can bind to known drug molecules [84]. It is estimated that 10^{60} drug-like molecules exist [85]. To dock m molecules against n protein structures, $m \times n$ docking operations are required.

We integrated our framework with VinaLC [77]. VinaLC is the MPI implementation of Vina [86], a popular serial molecular docking system. Each docking operation is treated as a task that gets scheduled independently. However, docking operations vary in execution times, as shown in Figure 6.5. Users do not know the workload of their tasks in advance. Hence, it is challenging to choose the ideal degree of parallelism for high resource utilization and better response time.

Sequence Alignment: P-SSW

The problem of aligning sequences involves arranging sequences to identify regions of similarity. It is the first step in studying functional, structural, or evolutionary relationships between DNA, RNA, or protein sequences [87, 88]. Multiple sequence alignment is an NP-hard problem [89]. A preprocessing step for multiple sequence alignment is to find pairwise alignments between all input sequences. Given a dataset of n sequences, the number of pairwise alignments is $\binom{n}{2} = \frac{n^2-n}{2}$. So aligning 1,000 sequences requires 499,500 independent pairwise alignments.

We integrated APlug in P-SSW, our parallel implementation of the SSW [81] alignment tool. SSW is a recent extension of Farrar's implementation [90] of the optimal pairwise alignment algorithm, the Smith-Waterman dynamic programming algorithm [91]. P-SSW adopts a dynamic scheduling policy to assign tasks to workers, where each pairwise alignment is a task. This policy guarantees better load balancing when processing pairwise alignments of arbitrary workloads.

Static process decomposition

Following our VinaLC example, docking four subsets of lead-like molecules against Thermus thermophilus gyrase B resulted in different speedup efficiencies. Figure 6.5 shows the speedup efficiencies, where DS3 is best and DS2 is worst. Figure 6.6 illustrates the workload frequency distributions of DS2 and DS3.

The tasks from DS2 have an irregular workload frequency distribution with many "heavy" tasks. Over half of the 2,000 tasks run in more than 200 seconds. It is hard to balance the workload of DS2 tasks since the probability of a large task being executed last is very high, rendering most cores idle towards the end. In contrast, only 67 DS3 tasks run in more than 200 seconds with most tasks finishing in less than a minute. Hence, most of the cores will be busy processing a large

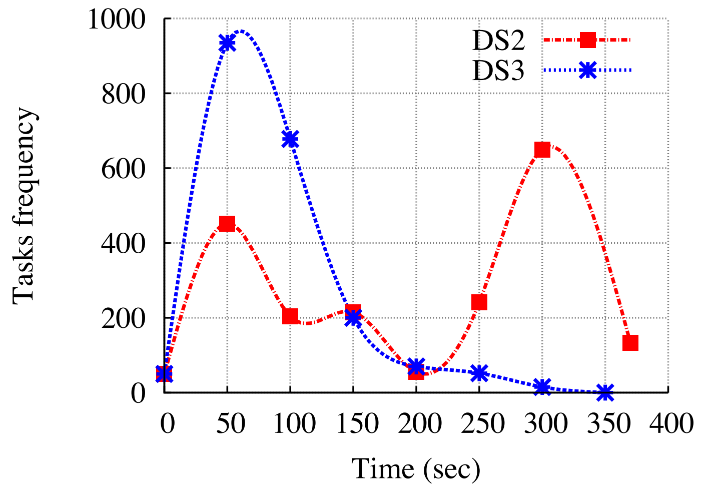


Figure 6.6: Workload frequency distributions for 2 datasets of lead-like compounds (i.e., same dataset and same size).

number of small tasks, i.e., workload is balanced.

6.3.2 The APlug Framework

APlug is a practical and easy to plug framework due to our novel idea of automatic tuning based on modeling workload frequency distributions. The performance of parallel systems is significantly affected by the workload density and skew in a set of tasks. Our novel idea facilitates the shift toward truly integrated estimation models, where regression analysis per parallel system is avoided. APlug infers the workload frequency distribution to: (*i*) adapt the parallelism of the scientific applications that vary dynamically, and (*ii*) tune dynamic process decompositions automatically. This section presents the workflow and algorithms of APlug.

The Framework Overview

An abstract workflow diagram of APlug is shown in Figure 6.7. APlug's inputs are user constraints and task decomposition type. A random sample of tasks from

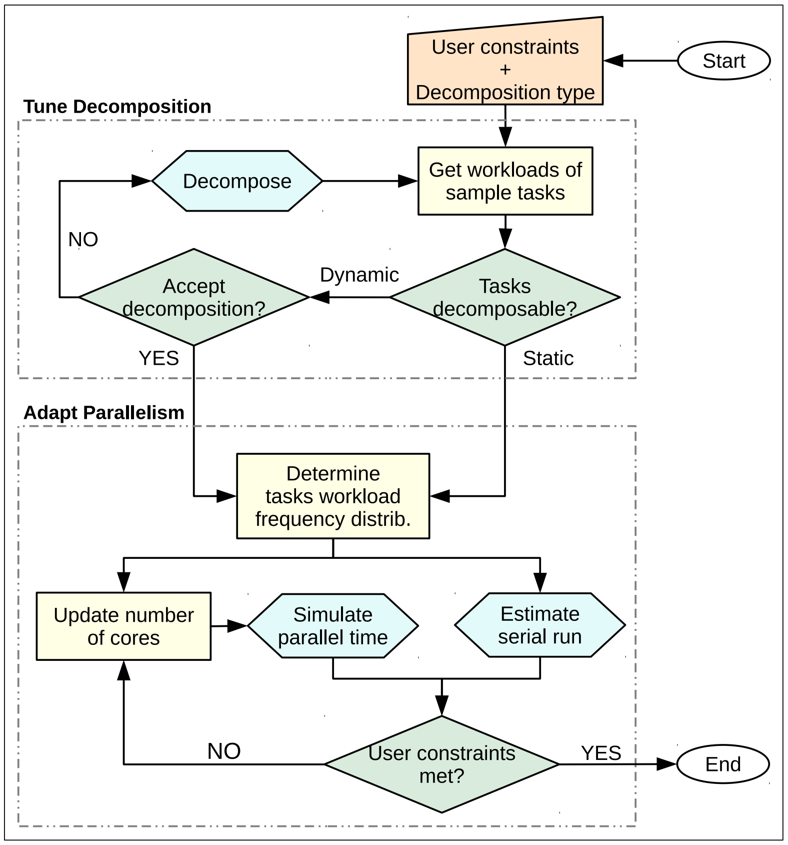


Figure 6.7: A simple flowchart of the APlug framework.

```
class APlug {
 public:
  void setNumOfTasks(int numOfTasks);
  void setSampleSize(int sampleSize);
  void loadSamples(double[] sampleTimes);
  int changeDecomposition(int last);
  int getRecCores(double tLimit, int cLimit);
  double getSerialTime(void);
  double getParallelTime(int cores);
  double getSpeedupEff(int cores);
};
```

Figure 6.8: The APlug C++ API foundation.

the entire dataset is executed to find the sample workload (i.e., runtimes). The decomposition type is either static or dynamic. For a static decomposition, APlug directly adapts the parallelism. For a dynamic decomposition, APlug automatically tunes the decomposition by finding the decomposition parameter leading to the highest scalability in terms of number of cores while the utilization threshold is satisfied.

6.3.3 API and Integration

The complete C++ source code of APlug, including its standalone utility program, is implemented in less than 800 lines of code. The framework is available for download[2] and as a public Amazon Machine Image[3]. This section highlights the most important aspects of our implementation. We present the main C++ API and discuss the integration of APlug in scientific applications. Most APlug methods shown in Figure 6.8 are implemented and ready to use with any application.

A typical scenario for integrating APlug with an application starts by including the APlug class and overriding `changeDecomposition()`, if needed. Users with an application of a fixed number of tasks do not need to implement or use this method. If the process can be decomposed dynamically, APlug needs to know how different decompositions are achieved. For example, decomposing an application with a tree-based search space involves using prefixes to partition the search space into sub-trees.

APlug uses sample tasks to predict workloads and adapt parallelism accordingly. The user needs to execute a random sample of tasks and pass their workloads to the framework using `loadSamples()`. The sampled tasks can (and should) be

[2]http://cloud.kaust.edu.sa/pages/aplug.aspx
[3]Amazon Machine Image ID: ami-df556bb6

part of the final results. Calling the method `getRecCores()` provides the number of cores needed to meet user constraints. The rest of the framework methods are provided for users to use them as needed and to customize their experience.

We integrated APlug with the three scientific applications discussed in Section 6.3.1. We did not reuse the samples in order to minimize code modification. Note that the overhead of running the sample is minimal compared to the gain from tuning. In the case of integrating APlug with VinaLC and P-SSW, only 20 lines of code were added or changed in their source codes. The method `changeDecomposition()` was not used because the number of tasks is fixed. In the case of ACME, the process decomposition is not fixed. We integrated APlug with ACME in less than 50 lines of code including overriding the method `changeDecomposition()`.

The APlug framework comes with a standalone utility program. This is useful in cases where the application source code is not available or if code integration is not preferred. The APlug utility accepts a file of sample tasks workloads and takes user constraints as input. The utility will output useful statistics including expected execution times, speedup efficiencies, and recommended number of cores.

6.4 Evaluation

Our APlug framework and the datasets detailed below are available for download online[2] and on Amazon EC2[3]. We evaluate APlug in the three different scientific applications discussed in Section 6.3.1. The accuracy of APlug's predictions and APlug's minimal overhead are shown. We conduct sensitivity analysis experiments to study the effects of the sample size on APlug. The effective scalability of the applications after integrating APlug with them is then demonstrated.

Experimental Setting

Datasets

We used real datasets to test APlug. (*i*) For molecular docking, similar to the authors of VinaLC, we dock lead-like compounds from the ZINC database [92] against Thermus thermophilus gyrase B[4]. (*ii*) For sequence alignment, 71,501 human protein sequences[5] and 65,685 random shotgun sequences[6] of Shewanella oneidensis bacteria [93]. (*iii*) For pattern mining, 2.6GB DNA[7] for the entire human genome.

Systems

We used various systems with different architectures. Namely, we used a supercomputer, a Linux cluster, and a public cloud cluster. The supercomputer is an IBM Blue Gene/P with 16,384 quad-core PowerPC processors @850MHz with a total memory of 64TB. The Linux cluster is an HP system of 480 cores @2.1GHz with each 24 cores sharing 148GB of RAM. The public cloud cluster was rented from Amazon EC2 and consisted of 40 on-demand M3 large instances[8]. Each instance had 2 cores and 7.5GB RAM.

Queries

For the molecular docking experiments, we randomly chose 10,000 lead-like compounds from the ZINC dataset to dock against Thermus thermophilus gyrase B. For sequence alignments, we created over 1 million tasks (1,000,405 pairwise alignments) using 1,415 random Human protein sequences. All the experiments with

[4]http://www.rcsb.org/pdb/explore.do?structureId=1KIJ
[5]ftp://ftp.ncbi.nih.gov/refseq/H_sapiens/mRNA_Prot/human.protein.faa.gz
[6]ftp://ftp.cbcb.umd.edu/pub/data/asmg_benchmark/
[7]http://webhome.cs.uvic.ca/ thomo/HG18.fasta.tar.gz
[8]http://aws.amazon.com/ec2/instance-types/

VinaLC and P-SSW were run on the Linux cluster. VinaLC could not run on the supercomputer because of library incompatibilities and P-SSW uses architecture specific SIMD operations. In the case of pattern mining, the number of tasks is not fixed and we test our decomposition tuning. On EC2, we chose a light pattern mining query that ends in reasonable time. APlug partitioned the search space tree of this query to 4,096 tasks (using prefixes of length 6). For the supercomputer, a pattern mining query that has enough workload to scale to thousands of cores was used. This query was decomposed automatically by APlug to 262,144 tasks (using prefixes of length 9).

Baseline

The baseline implementation that we compare to is the naïve solution to the degree of parallelism problem. Given the runtimes of a sample of the tasks, the serial time is estimated by multiplying the average sample runtime by the total number of tasks. The parallel runtime is then calculated by dividing the estimated serial time by the number of cores. The samples used are the ones used with APlug and in the case of decomposition tuning we use APlug's decomposition.

6.4.1 Accuracy

It is not difficult to predict runtimes of highly scalable applications when tasks have similar workloads. Indeed, Table 6.2 shows that the baseline method achieves fair accuracy in predicting the runtimes of P-SSW up to 256 cores. The query was to align 1,415 shotgun sequences from the Shewanella oneidensis bacteria dataset. Over a million tasks are run but most of the alignments have similar workloads because the sequences are of similar lengths. While we do not know this fact in advance, it is risky to use the baseline method. Generally, it is not common to

Table 6.2: The baseline method provides fairly accurate parallel time estimations when tasks workloads are uniform. In this experiment, P-SSW is used to alig 1,415 random DNA shotgun sequences from the Shewanella oneidensis bacteria. Most shotgun sequences have similar lengths and tasks workloads are close to uniform.

Cores	**Actual time (hours)**	**APlug error (%)**	**Baseline error (%)**
16	45	0.8	0.9
32	23	0.8	0.8
64	12	0.8	0.9
128	6	0.9	1.0
256	3	0.9	1.0

Table 6.3: Accuracy of APlug and the baseline compared to actual parallel times of P-SSW on the Linux cluster.

Cores	**Actual time (hours)**	**APlug error (%)**	**Baseline error (%)**
16	145	0.1	7.4
32	73	0.1	7.5
64	36	0.2	7.5
128	18	0.2	7.5
256	9	0.3	7.5

Table 6.4: Accuracy of APlug and the baseline compared to actual parallel times of VinaLC on the Linux cluster.

Cores	**Actual time (hours)**	**APlug error (%)**	**Baseline error (%)**
16	31	0.4	8.2
32	16	0.4	8.1
64	8	0.4	7.8
128	4	0.4	7.0
256	2	0.3	5.3

Table 6.5: Accuracy of APlug and the baseline method compared to actual parallel times of ACME on Amazon EC2.

Cores	**Actual time (minutes)**	**APlug error (%)**	**Baseline error (%)**
4	130	0.9	17.8
8	66	0.9	17.9
16	33	0.8	18.1
32	17	1.2	18.8
64	9	1.8	20.3

have queries with uniform workload frequency distributions in practice. Next, we show queries for sequence alignment, molecular docking, and pattern mining where tasks workloads are skewed, which is the common case.

We verify the accuracy of APlug and compare it to the baseline method on different architectures. For each application, we run the same query using different numbers of cores and compare the actual runtimes with the estimations from APlug and the baseline method. The query for Table 6.3 has the same number of tasks as in Table 6.2 only this time we align sequences from the human protein dataset, where sequences are of variable lengths leading to skewed task workloads. APlug captures workload skewness to better estimate runtimes and accurately predict speedup efficiencies. Similarly, Tables 6.4 and 6.5 show that APlug significantly outperforms the baseline method.

Figures 6.9, 6.10, and 6.11 show the actual workload frequencies and the PDFs

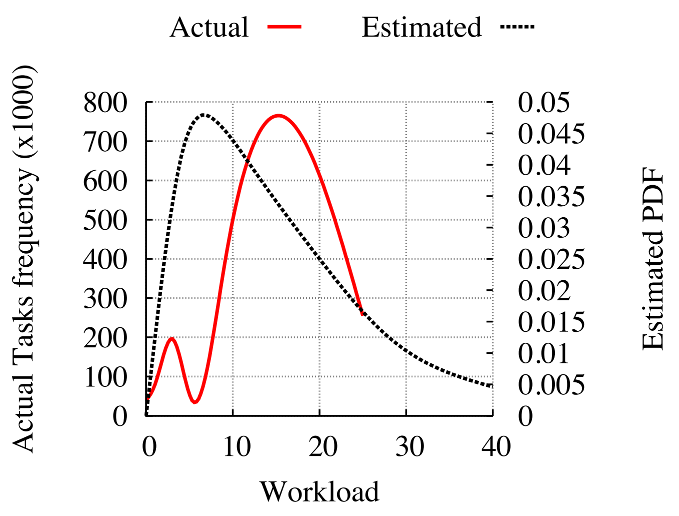


Figure 6.9: P-SSW query from Table 6.3 comparing actual workload frequency distribution and PDF used by APlug.

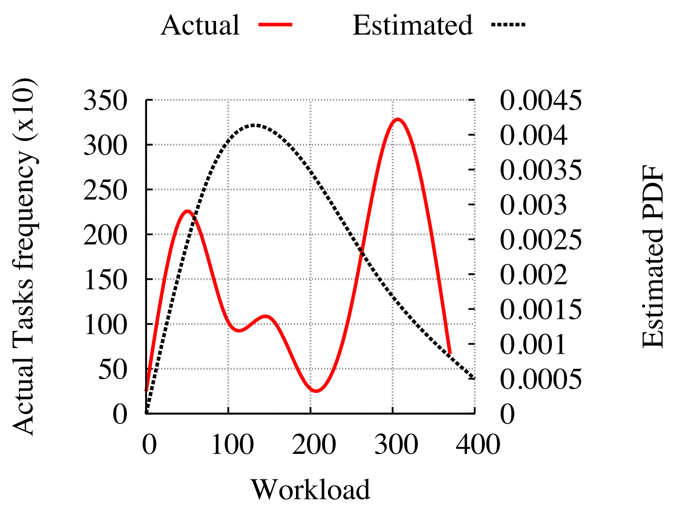


Figure 6.10: VinaLC query from Table 6.4 comparing actual workload frequency distribution and PDF used by APlug. While a single Gamma distribution can not have the shape of the actual workload frequency distribution, it captures workload skewness and approximates the total runtimes accurately.

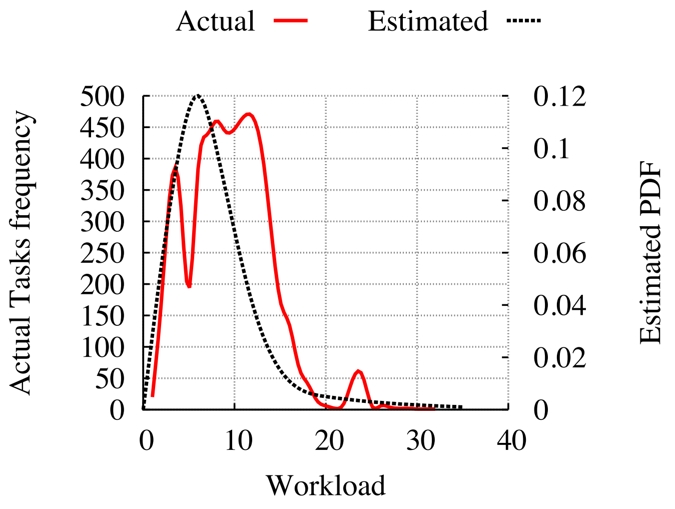


Figure 6.11: ACME query from Table 6.5 comparing actual workload frequency distribution and PDF used by APlug.

Table 6.6: The different speedup efficiencies of ACME on the supercomputer using different decompositions. APlug consistently chooses the best decomposition (256K tasks).

Cores	**Speedup Efficiency wrt #Tasks**			
	16K	64K	256K	1024K
512	0.94	0.97	0.98	0.81
1,024	0.87	0.97	0.97	0.83
2,048	0.83	0.92	0.97	0.83
4,096	0.46	0.76	0.92	0.76
8,192	0.25	0.46	0.76	0.46

APlug used to predict them. Since we care about the total runtime, the shapes of the actual workload frequencies and the PDFs need not be similar. The total runtime is a function of the area under the curves. The values of the corresponding integrals are similar, leading to low estimation errors. APlug achieves this by implicitly considering the infrastructure performance properties using the runtimes of the sample tasks.

The next experiment shows the accuracy of APlug decomposition tuning. Table 6.6 shows the speedup efficiencies when different decompositions are used for our

Table 6.7: Accuracy of the estimated parallel time (and cost) of ACME on Amazon EC2 cloud. APlug is able to predict time so users can meet budget and time constraints.

	Number of Amazon EC2 Instances				
	1	**10**	**20**	**30**	**40**
Cores	2	20	40	60	80
Cost/Hour	\$0.24	\$2.40	\$4.80	\$7.20	\$9.60
Est. Time	2.5 Days	5.9 Hr	3.1 Hr	2.1 Hr	1.5 Hr
Act. Time	2.9 Days	5.1 Hr	4.3 Hr	2.3 Hr	1.6 Hr
Est. Cost	\$14.40	\$14.40	\$19.20	\$21.60	\$19.20
Act. Cost	\$16.80	\$14.40	\$24.00	\$21.60	\$19.20

query. APlug chooses the decomposition that results in the best speedup efficiency using different numbers of cores. However, the default decomposition method of ACME fails in most cases because it considers number of cores to decompose the problem.

Accurate decomposition and runtime prediction saves money for users of supercomputing centers and clouds. Our next experiment shows how APlug guides user decisions to stick to their budget. We decompose and adapt the parallelism of a pattern mining query given a pricing scheme. The user needs to mine human DNA for patterns in less than 3 hours and without spending more than \$20. Table 6.7 shows that renting 10 instances meets the budget but not the time constraint. Similarly, renting 30 instances meets the time constraint but not the budget. It is interesting that renting more instances, in this case 40, meets both constraints. It is not straightforward to arrive at this conclusion without utilizing APlug capabilities.

During execution, APlug guides users if scaling out or in is desired. Table 6.8 shows that APlug provides good expected runtimes and speedup efficiencies online. The error in APlug's online predictions is slightly higher than its initial run because it is difficult to capture the execution of the tasks that are running

Table 6.8: APlug provides users with accurate predictions for adding (or removing) resources online. In this experiment 2,000 VinaLC tasks are started with 130 cores then after half the tasks were done, APlug was consulted to add more cores.

Additional	**Total Time (min)**		**Speedup Efficiency**	
Cores	**Actual**	**Predicted**	**Actual**	**Predicted**
0	33.23	34.43	0.95	0.92
100	28.85	27.78	0.62	0.64
200	28.17	25.13	0.44	0.49
300	26.80	24.58	0.35	0.38

during online prediction. Initially, 130 cores were used. Additional cores were added after half the tasks were done. Counterintuitively, adding cores at this stage negatively affects speedup efficiency since the remaining tasks do not have enough workload to fully utilize a larger number of cores. In this experiment, adding 300 cores improves the time by less than 7 minutes at the cost of a drop in efficiency from 0.95 to 0.35. APlug accurately provides online facts for users to make informed decisions.

6.4.2 Sensitivity Analysis

APlug is expected to be sensitive to the sample size because it uses sample tasks to build its workload model. We use random sampling, the simplest form of probability sampling, where all tasks have the same probability of being in the sample. Random sampling is preferred in cases where the information we know about the total population (i.e., tasks) is little. Statistically, the precision of the estimator given a large number of total tasks depends on sample size, and not sample percentage of the total tasks [94]. For example, the precision of a random sample of 100 tasks from 100,000 tasks and from a million tasks is the same. For each experiment in this section, we note the sample margin of error range for a 95% confidence interval.

Table 6.9: Sensitivity analysis of the sample size used in APlug to estimate the execution time of VinaLC using 256 cores on the Linux cluster. The actual time was over 2 hours. The sample margin of error is between 9% and 3%.

Sample size (%)	**Sample time (minutes)**	**APlug error (%)**	**Baseline error (%)**
1	2	8.0	13.9
2	3	4.7	10.9
4	5	0.4	5.3
8	10	0.3	4.2

Table 6.10: Sensitivity analysis of the sample size used in APlug to estimate the execution time of P-SSW using 256 cores of the Linux cluster. The actual time was over 9 hours. The sample margin of error is between 1% and 0.3%.

Sample size (%)	**Sample time (minutes)**	**APlug error (%)**	**Baseline error (%)**
1	5	0.5	7.2
2	11	0.6	7.1
4	22	0.2	7.5
8	44	0.1	7.7

Table 6.11: Sensitivity analysis of the sample size used in APlug to estimate the serial execution time of ACME on Amazon EC2 cloud. The actual time was over 8.5 hours. The sample margin of error is between 15% and 5%.

Sample size (%)	**Sample time (minutes)**	**APlug error (%)**	**Baseline error (%)**
1	6	13.5	42.0
2	11	11.5	32.4
4	21	1.2	10.8
8	41	1.4	10.2

Table 6.12: Sensitivity analysis of sample size in ACME's decomposition tuning. Best decomposition empirically found to be 262,144 tasks. The sample margin of error is between 21% and 7%.

Sample Size (# Tasks)	**Tuning cost (seconds)**	**Suggested decomposition (# Tasks)**
10	9	16,384
20	11	16,384
40	7	65,536
80	4	262,144
160	6	262,144
320	12	262,144

Tables 6.9, 6.10, and 6.11 show that practically acceptable accuracy is achieved with small sample sizes. The overhead of running samples increases with sample size but is not steep. The overhead is basically the runtimes of the sample tasks, which is dependent on sample workload not only sample size. The time it takes to run the sample tasks is orders of magnitude lower than the total runtime. The accuracy of APlug is significantly better than the baseline even for small samples because APlug captures the workload skewness and irregularity of tasks workloads.

The next experiment studies APlug's decomposition tuning with respect to different sample sizes. We empirically find the best decomposition by exhaustively using different prefix lengths to partition the search space tree of a pattern mining query. Decomposing the example process to 262,144 tasks creates the most fine-grained tasks with minimal useless work. Table 6.12 shows that APlug finds the optimal decomposition with small sample sizes. The tuning time is minimal, especially when compared to the full query execution time.

6.4.3 Effective Scalability after using APlug

Guided by APlug; P-SSW, VinaLC, and ACME adapt their parallelism to achieve high speedup efficiencies with minimal overhead. Tables 6.13, 6.14, and 6.15 show

Table 6.13: APlug accurate prediction for VinaLC on the Linux cluster.

Cores	**Predicted**	**Actual**
	Speedup Efficiency	**Speedup Efficiency**
15	1.00	1.00
30	0.99	0.98
60	0.99	0.97
120	0.99	0.95
240	0.98	0.94
480	0.96	0.93

Table 6.14: APlug accurate prediction for P-SSW on the Linux cluster.

Cores	**Predicted**	**Actual**
	Speedup Efficiency	**Speedup Efficiency**
15	1.00	1.00
30	0.99	0.98
60	0.99	0.98
120	0.99	0.96
240	0.98	0.95
480	0.97	0.93

that APlug is able to accurately adapt the parallelism of different applications on different architectures. Due to time constraints, speedup efficiencies are calculated according to a 15-core system for VinaLC and P-SSW and to a 256-core system for ACME. No manual tuning for ACME on the supercomputer was needed as its decomposition is automatically tuned by APlug. VinaLC was not run on the supercomputer because of library incompatibilities. P-SSW uses architecture specific SIMD operations that prevents it from running on the supercomputer.

Table 6.15: ACME is able to scale to 16,384 cores with high speedup efficiency on the supercomputer using APlug's automatic decomposition tuning. APlug accurately predicts speedup efficiency of ACME using thousands of cores.

Cores	**Predicted Speedup Efficiency**	**Actual Speedup Efficiency**
256	1.0	1.00
1,024	0.99	0.99
2,048	0.99	0.98
4,096	0.98	0.96
8,192	0.98	0.91
16,384	0.97	0.98

Chapter 7

Concluding Remarks

There is a point in every contest when sitting on the sidelines is not an option

Dean Smith
1931 — 2015 CE

This chapter concludes this dissertation with a summary of our contributions and an outlook on possible future research directions.

7.1 Summary of Contributions

String queries are computationally demanding and require parallel execution to finish in reasonable times. Many database management concepts will need to be rethought before being applied to a string database system. In this dissertation, we introduced a scalable and extensible string data model and a declarative query language for strings. We then proposed an architecture of a large-scale string database system and demonstrated StarDB, a distributed database system for large-scale string analytics. We aim to kick-start many string database management systems.

To deal with the challenges of large-scale string analytics, we focus on motif extraction as a core operation that is well studied. Many important applications, such as bioinformatics, time series and log analysis, depend on motif extraction from one long sequence. Existing methods for extracting motifs from a single sequence are cache inefficient and serial.

We introduced ACME, a combinatorial method for extracting motifs repeated in a single long string. First, we target the effective utilization of memory caches. We devise CAST, a cache-aware search space traversal technique that arranges the search space in contiguous blocks; taking advantage of the cache hierarchy in modern architectures. Equipped with CAST, ACME is orders of magnitude faster than existing motif extractors.

Motif extraction can be computationally demanding even for relatively short strings. The next step was to scale out ACME in order to perform tasks in reasonable times. To this end, we introduce FAST, a fine-grained problem decomposition technique that allows ACME to scale with minimal overhead. In our experiments we demonstrated that ACME handles the entire DNA sequence for the human genome on a single high-end multi-core machine; this is 3 orders of magnitude longer compared to the state-of-the-art. We also showed that ACME can be deployed in a variety of large scale parallel architectures, including Amazon EC2 and a supercomputer with 16,384 CPUs.

To run on thousands of cores, tuning and execution time estimation become critical. We Introduce APlug, an automatic tuning framework for large-scale parallel applications with many independent tasks. APlug adapts the degree of parallelism and automatically decomposes the parallel process to enable users to achieve efficient utilization of CPU resources. We studied the correlation between the workload frequency distributions of a set of tasks and the resource utilization. Our models are based on this correlation.

APlug facilitates the shift towards truly integrated estimation models. Our experiments show the viability of our framework for molecular docking, sequence alignment, and pattern mining using 16,384 cores in a supercomputer, 480 cores in a Linux cluster, and 80 cores from a public cloud. APlug estimates the serial and parallel times to suggest the best degree of parallelism in very short time with less than 10% error. Users can use APlug to optimize various quantities; such as, runtime and financial cost.

7.2 Future Research Directions

Currently ACME is an in-memory system. We are working on a disk-based version that will allow ACME to support longer strings. A general nonresident storage manager is needed in distributed environments where local memory is limited and I/O is slow. We started working on handling nonresident data and already achieved three orders of magnitude speedup compared to the naïve approach. The nonresident storage manager uses the collective distributed memory in a large infrastructure as system storage. Large distributed memory is used as if it was local to every machine. This will be accomplished by predicting and prefetching data to emulate an in-memory scenario.

While many commercial and scientific applications have independent tasks, we intend to extend APlug to work with tasks that have inter-dependencies. It is challenging to predict dependencies because they differ according to data size, decomposition, and execution environment. We attempt to address tasks dependencies by two ways: profiling the code, and collecting data during runtime. Machine learning techniques can then be used to construct models for typical application behaviors.

REFERENCES

[1] Thomas P Niedringhaus, Denitsa Milanova, Matthew B Kerby, Michael P Snyder, and Annelise E Barron. Landscape of next-generation sequencing technologies. *Analytical chemistry*, 83(12):4327–4341, 2011.

[2] A. Mathur, A. Sihag, E. Bagaria, S. Rajawat, et al. A new perspective to data processing: Big data. In *Processdings of INDIACom*, pages 110–114, 2014.

[3] Maxime Crochemore, Wojciech Rytter, and Maxime Crochemore. *Text algorithms*, volume 698. World Scientific, 1994.

[4] Seymour Ginsburg and Xiaoyang Wang. Pattern matching by rs-operations: Towards a unified approach to querying sequenced data. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 293–300, New York, NY, USA, 1992. ACM.

[5] Gösta Grahne, Matti Nykänen, and Esko Ukkonen. Reasoning about strings in databases. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, pages 303–312, New York, NY, USA, 1994. ACM.

[6] Seymour Ginsburg and X. Sean Wang. Regular sequence operations and their use in database queries. *J. Comput. Syst. Sci.*, 56(1):1–26, February 1998.

[7] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. String operations in query languages. In *Proc. of PODS*, pages 183–194, 2001.

[8] Sandeep Tata, Willis Lang, and Jignesh M. Patel. Periscope/SQ: Interactive exploration of biological sequence databases. In *Proc. of VLDB*, pages 1406–1409, 2007.

[9] G. Grahne, R. Hakli, M. Nykänen, H. Tamm, and E. Ukkonen. Design and implementation of a string database query language. *Information Systems*, 28(4):311–337, 2003.

[10] N.H. Balkir, E. Sukan, G. Ozsoyoglu, and G. Ozsoyoglu. Visual: a graphical icon-based query language. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 524–533, Feb 1996.

[11] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[12] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[13] Avrilia Floratou, Sandeep Tata, and Jignesh M. Patel. Efficient and Accurate Discovery of Patterns in Sequence Data Sets. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(8):1154–1168, August 2011.

[14] Alberto Apostolico, Matteo Comin, and Laxmi Parida. VARUN: discovering extensible motifs under saturation constraints. *IEEE/ACM Transactions on Computational Biology Bioinformatics*, 7(4):752–26, 2010.

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

[16] Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, Fabio Vandin, Steven Salzberg, and Tandy Warnow. MADMX: A Novel Strategy for Maximal Dense Motif Extraction. In *Proceedings of Workshop on Algorithms in Bioinformatics*, pages 362–374, 2009.

[17] Majed Sahli, Essam Mansour, and Panos Kalnis. Stardb: A large-scale dbms for strings. In *VLDB*, August 2015.

[18] Majed Sahli, Essam Mansour, and Panos Kalnis. Parallel motif extraction from very long sequences. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2013.

[19] Majed Sahli, Essam Mansour, and Panos Kalnis. ACME: A scalable parallel system for extracting frequent patterns from a very long sequence. *The VLDB Journal*, 23(6):871–893, December 2014.

[20] Majed Sahli, Essam Mansour, Tariq Alturkestani, and Panos Kalnis. Automatic tuning of bag-of-tasks application. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, April 2015.

[21] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[22] Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Thure Etzold and Patrick Argos. SRS — an indexing and retrieval tool for flat file data libraries. *Computer applications in the biosciences : CABIOS*, 9(1):49–57, 1993.

[24] Thure Etzold and Patrick Argos. Transforming a set of biological flat file libraries to a fast access network. *Computer applications in the biosciences : CABIOS*, 9(1):59–64, 1993.

[25] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 99–110, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[26] Joel Richardson. Supporting lists in a data model (a timely approach). In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 127–138, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[27] Pierre Wolper. Temporal logic can be more expressive. In *Foundations of Computer Science, 1981. SFCS '81. 22nd Annual Symposium on*, pages 340–348, Oct 1981.

[28] Nizar R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):1–41, 2010.

[29] Santan Challa and Parimala Thulasiraman. Protein sequence motif discovery on distributed supercomputer. In *Proceedings of the international conference on Advances in grid and pervasive computing (GPC)*, pages 232–243, 2008.

[30] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. An efficient multicore implementation of planted motif problem. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pages 9–15, 2010.

[31] Naga Shailaja Dasari, D Ranjan, and M Zubair. High performance implementation of planted motif problem using suffix trees. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pages 200–206, 2011.

[32] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 428–434, 2011.

[33] Benoît Marchand, Vladimir B. Bajic, and Dinesh K. Kaushik. Highly scalable ab initio genomic motif identification. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 56:1–56:10, 2011.

[34] Huang, Yang, Chowdhary, Kassim, and Bajic. An algorithm for ab initio dna motif detection. In *Information Processing and Living Systems*, volume 2, pages 611–614, 2005.

[35] Marie-France Sagot. Spelling Approximate Repeated or Common Motifs Using a Suffix Tree. In *Proceedings of 3rd Latin American Symposium on Theoretical Informatics*, pages 374–390, April 1998.

[36] Laurent Marsan and Marie-France Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *Journal of Computational Biology*, 7(3-4):345–362, 2000.

[37] Dimitris Tsirogiannis and Nick Koudas. Suffix tree construction algorithms on modern hardware. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 263–274, 2010.

[38] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–498. ACM, 2003.

[39] Thomas Guyet, Catherine Garbay, and Michel Dojat. Knowledge construction from time series data using a collaborative exploration system. *Journal of biomedical informatics*, 40(6):672–687, 2007.

[40] David Minnen, Charles L Isbell, Irfan Essa, and Thad Starner. Discovering multivariate motifs using subsequence density estimation and greedy mixture learning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 615, 2007.

[41] Yasser F. O. Mohammad and Toyoaki Nishida. Exact discovery of length-range motifs. In *ACIIDS (2)'14*, pages 23–32, 2014.

[42] Ankur Narang and Souvik Bhattacherjee. Parallel exact time series motif discovery. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 304–315, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] Abdullah Mueen, Eamonn Keogh, Qiang Zhu, and Sydney Cash. Exact discovery of time series motifs. In *SDM*, 2009.

[44] Adrian Daniel Popescu, Andrey Balmin, Vuk Ercegovac, and Anastasia Ailamaki. PREDIcT: Towards predicting the runtime of large scale iterative analytics. *Proc. VLDB Endow.*, 6(14):1678–1689, September 2013.

[45] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[46] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *Proc. of the 2013 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–12, 2013.

[47] Ioan Raicu, Ian Foster, Mike Wilde, Zhao Zhang, Kamil Iskra, Peter Beckman, Yong Zhao, Alex Szalay, Alok Choudhary, Philip Little, Christopher Moretti, Amitabh Chaudhary, and Douglas Thain. Middleware support for many-task computing. *Cluster Computing*, 13(3), 2010.

[48] MohammadReza HoseinyFarahabady, Young Choon Lee, and Albert Y Zomaya. Randomized approximation scheme for resource allocation in hybrid-cloud environment. *The Journal of Supercomputing*, 2014.

[49] Timothy G Armstrong, Zhao Zhang, Daniel S Katz, Michael Wilde, and Ian T Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *Many-Task Computing on Grids and Supercomputers, Workshop on*, 2010.

[50] Ana-Maria. Oprescu, Thilo Kielmann, and Haralambie Leahu. Stochastic tail-phase optimization for bag-of-tasks execution in clouds. In *Utility and Cloud Computing, Intl. Conf. on*, 2012.

[51] Ming Tao, Shoubin Dong, and Liping Zhang. A multi-strategy collaborative prediction model for the runtime of online tasks in computing cluster/grid. *Cluster Computing*, 14(2), 2011.

[52] Maleeha Kiran, Aisha-Hassan A Hashim, Lim Mei Kuan, and Yap Yee Jiun. Execution time prediction of imperative paradigm tasks for grid scheduling optimization. *Int J Comput Sci Netw Secur*, 9(2), 2009.

[53] C.L. Mendes and D.A. Reed. Integrated compilation and scalability analysis for parallel systems. In *Parallel Architectures and Compilation Techniques, 1998. Proc.. 1998 Intl. Conf. on*, 1998.

[54] T. Miu and P. Missier. Predicting the execution time of workflow activities based on their input features. In *High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion*, 2012.

[55] Rubing Duan, F. Nadeem, Jie Wang, Yun Zhang, R. Prodan, and T. Fahringer. A hybrid intelligent method for performance modeling and prediction of workflow activities in grids. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM Intl. Symposium on*, 2009.

[56] J. Delgado, A.S. Eddin, M. Adjouadi, and S.M. Sadjadi. Paravirtualization for scientific computing: Performance analysis and prediction. In *High Performance Computing and Communications, Intl. Conf. on*, 2011.

[57] Laura Carrington, Michael Laurenzano, and Ananta Tiwari. Characterizing large-scale hpc applications through trace extrapolation. *Parallel Processing Letters*, 23(04), 2013.

[58] Darren J. Kerbyson, Kevin J. Barker, Diego S. Gallo, Dong Chen, José R. Brunheroto, Kyung Dong Ryu, George L.-T. Chiu, and Adolfy Hoisie. Tracking the performance evolution of blue gene systems. In *Proc. of the 28th Intl. Supercomputing Conf.*, 2013.

[59] Qishi Wu and V.V. Datla. On performance modeling and prediction in support of scientific workflow optimization. In *Services, IEEE World Congress on*, 2011.

[60] Mitesh R Meswani, Laura Carrington, Didem Unat, Allan Snavely, Scott Baden, and Stephen Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. *Intl. Journal of High Performance Computing Applications*, 27(2), 2013.

[61] Arcot Rajasekar. String-oriented databases. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, SPIRE '99, 1999.

[62] Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. Definable relations and first-order query languages over strings. *J. ACM*, 50(5):694–751, September 2003.

[63] Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. String operations in query languages. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 183–194, 2001.

[64] R. Hakli, M. Nykanen, and H. Tamm. Adding string processing capabilities to data management systems. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 122–131, 2000.

[65] S. Tata, J.S. Friedman, and A. Swaroop. Declarative querying for biological sequences. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 87–87, April 2006.

[66] Giansalvatore Mecca and Anthony J. Bonner. Query languages for sequence databases: Termination and complexity. *IEEE Trans. on Knowl. and Data Eng.*, 13(3):519–525, May 2001.

[67] I. Gorton and D.K. Gracio. *Data-Intensive Computing: Architectures, Algorithms, and Applications.* 2012.

[68] Michael Owens. Embedding an sql database with sqlite. *Linux J.*, 2003(110):2–, June 2003.

[69] Maria Federico and Nadia Pisanti. Suffix tree characterization of maximal motifs in biological sequences. *Theoretical Computer Science*, 410(43):4391–4401, October 2009.

[70] E Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[71] Verónica Becher, Alejandro Deymonnaz, and Pablo Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–53, 2009.

[72] S.M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X.J. Collazo-Mojica. A modeling approach for estimating execution time of long-running scientific applications. In *Parallel and Distributed Processing. IEEE Intl. Symposium on*, 2008.

[73] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes.* Tata McGraw-Hill Education, 2002.

[74] Leonard Kleinrock. *Queueing Systems*, volume I: Theory. Wiley-Interscience, 1975.

[75] David Meisner and Thomas F Wenisch. Stochastic queuing simulation for data center workloads. In *Exascale Evaluation and Research Techniques Workshop*, 2010.

[76] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, September 2010.

[77] Xiaohua Zhang, Sergio E. Wong, and Felice C. Lightstone. Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines. *Journal of Computational Chemistry*, 34(11), 2013.

[78] Changjun Wu, A. Kalyanaraman, and W.R. Cannon. pGraph: Efficient parallel construction of large-scale protein sequence homology graphs. *Parallel and Distributed Systems, IEEE Trans. on*, 23(10), 2012.

[79] Gregory R Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys (CSUR)*, 23(1), 1991.

[80] A.W. Mu'alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, Jun 2001.

[81] Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth. SSW library: An simd smith-waterman c/c++ library for use in genomic applications. *PLoS ONE*, 8(12), 2013.

[82] Thomas Lengauer and Matthias Rarey. Computational methods for biomolecular docking. *Current Opinion in Structural Biology*, 6(3), 1996.

[83] Huameng Li, Aiguo Liu, Zhenjiang Zhao, Yufang Xu, Jiayuh Lin, David Jou, and Chenglong Li. Fragment-based drug design and drug repositioning using multiple ligand simultaneous docking (MLSD): identifying celecoxib and template compounds as novel inhibitors of signal transducer and activator of transcription 3 (stat3). *Journal of medicinal chemistry*, 54(15), 2011.

[84] Andrew L Hopkins and Colin R Groom. The druggable genome. *Nature reviews Drug discovery*, 1(9), 2002.

[85] Peter Kirkpatrick and Clare Ellis. Chemical space. *Nature*, 432(7019), 2004.

[86] Oleg Trott and Arthur J Olson. AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of comp. chemistry*, 31(2), 2010.

[87] Chris Sander and Reinhard Schneider. Database of homology-derived protein structures and the structural meaning of sequence alignment. *Proteins: Structure, Function, and Bioinformatics*, 9(1), 1991.

[88] David W Mount. *Sequence and genome analysis.* 2004.

[89] Isaac Elias. Settling the intractability of multiple alignment. *Journal of Computational Biology*, 13(7), 2006.

[90] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2), 2007.

[91] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1), 1981.

[92] John J Irwin and Brian K Shoichet. Zinc-a free database of commercially available compounds for virtual screening. *Journal of chemical information and modeling*, 45(1), 2005.

[93] John F Heidelberg, Ian T Paulsen, Karen E Nelson, Eric J Gaidos, William C Nelson, Timothy D Read, Jonathan A Eisen, Rekha Seshadri, Naomi Ward, Barbara Methe, et al. Genome sequence of the dissimilatory metal ion–reducing bacterium shewanella oneidensis. *Nature biotechnology*, 20(11), 2002.

[94] Sharon Lohr. *Sampling: design and analysis.* Cengage Learning, 2009.