2017

Efficient similarity computations on parallel machines using data shaping

Parijat Shukla
Iowa State University

Efficient similarity computations on parallel machines using data shaping

by

Parijat Shukla

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Arun K. Somani, Major Professor

Akhilesh Tyagi

Joseph Zambreno

Sigurdu (Siggi) Olafsson

Zhao Zhang

Iowa State University

Ames, Iowa

2017

DEDICATION

Dedicated to my parents, Dr. Girish Chandra Shukla and Dr. Savita Shukla.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my gratitude to people who helped me with my graduate education and research.

First and foremost, I would like to thank Dr. Arun K. Somani for being my major professor. He has been the guiding beacon for my research. I have always been inspired by his discipline, dedication, and intelligence. I would like to specially thank him for his constant effort to impart good presentation and technical writing skills to me, and for encouraging free thinking.

I would also like to thank Dr. Akhilesh Tyagi, Dr. Zhao Zhang, Dr. Joseph Zambreno, and Dr. Siggi Olafson for serving on my committee and for providing me with constructive feedback regarding my research. I am also thankful to other professors in the ECpE department and ISU: Dr. Philip Jones, Dr. Ahmed Kamal, Dr. Vivekanand Roy, Dr. Manimaran, Dr. Mani Mina, Dr. Umesh Vaidya, and others for teaching me various courses and for providing encouragment. I would like to extend a special thanks to the ECpE department and Iowa State University for providing me an excellent atmosphere to conduct my research. I would like to sincerely thank Dr. Nebbe Carver (Senior Physician at Thielen Health Center), and Staff at Student Counseling Services at ISU for helping me overcome a difficult medical condition (Chronic Fatigue Syndrome) and continue pursuing PhD program.

I have been fortunate enough to work with many wonderful people like Mr. Larry Kaplan (Chief Software Architect, Supercomputing Products and my supervior at Cray, Inc.), Dr. Jie Hu (my Master's theis advisor at New Jersey Institute of Technology, NJ), Dr. Asim Banerjee (my B. Tech. project advior at Dhirubhai Ambani Institute of Information & Communication Technology, India), and Mr. Rajkuamr Mishra (Director, Ericsson India Limited, Mumbai & my Manager at Reliance Communications) among others in this regard. I would like to express my gratitude to all such wonderfull people for their words of encouragement and support.

Research is often inspired by the daily conversations we have with other people. My research group at Iowa State University, namely the Dependable Computing and Networking Laboratory, enabled me to network with a large set of such people. I would like to specially thank Prem, Prasad, David, Jin Xu, Koray, Lizandro, Matt, Pavan, Cory, Ashish, Joy, Karthik, Teng, Piyush, Pratik, Ravi, Utkarsh, Haoyuan, and Dr. Mayak Mishra.

I am grateful to my parents for motivating me to pusrsue higher studies. Finally, I would like to thank my wife Shweta. Her support, encouragement, and quiet patience were undeniably the bedrock upon which the years of my graduate life have been built. I must accept that this work would not have been possible without her support.

ABSTRACT

Similarity computation is a fundamental operation in all forms of data. Big Data is, typically, characterized by attributes such as volume, velocity, variety, veracity, etc. In general, Big Data variety appears as structured, semi-structured or unstructured forms. The volume of Big Data in general, and semi-structured data in particular, is increasing at a phenomenal rate. Big Data phenomenon is posing new set of challenges to similarity computation problems occurring in semi-structured data.

Technology and processor architecture trends suggest very strongly that future processors shall have ten's of thousands of cores (hardware threads). Another crucial trend is that ratio between on-chip and off-chip memory to core counts is decreasing. State-of-the-art parallel computing platforms such as General Purpose Graphics Processors (GPUs) and MICs are promising for high performance as well high throughput computing. However, processing semi-structured component of Big Data efficiently using parallel computing systems (e.g. GPUs) is challenging. Reason being most of the emerging platforms (e.g. GPUs) are organized as Single Instruction Multiple Thread/Data machines which are highly structured, where several cores (streaming processors) operate in lock-step manner, or they require a high degree of task-level parallelism.

We argue that effective and efficient solutions to key similarity computation problems need to operate in a synergistic manner with the underlying computing hardware. Moreover, semi-structured form input data needs to be shaped or reorganized with the goal to exploit the enormous computing power of *state-of-the-art* highly threaded architectures such as GPUs. For example, shaping input data (via encoding) with minimal data-dependence can facilitate flexible and concurrent computations on high throughput accelerators/co-processors such as GPU, MIC, etc.

We consider various instances of traditional and futuristic problems occurring in intersection of semi-structured data and data analytics. Preprocessing is an operation common at initial stages of data processing pipelines. Typically, the preprocessing involves operations such as data extraction, data selection, etc. In context of semi-structured data, twig filtering is used in identifying (and extracting) data of interest. Duplicate detection and record linkage operations are useful in preprocessing tasks such as data cleaning, data fusion, and also useful in data mining, etc., in order to find similar tree objects. Likewise, tree edit is a fundamental metric used in context of tree problems; and similarity computation between trees another key problem in context of Big Data.

This dissertation makes a case for *platform-centric data shaping* as a potent mechanism to tackle the data- and architecture-borne issues in context of semi-structured data processing on GPU and GPU-like parallel architecture machines. In this dissertation, we propose several data shaping techniques for tree matching problems occurring in semi-structured data. We experiment with real world datasets. The experimental results obtained reveal that the proposed *platform-centric data shaping approach* is effective for computing similarities between tree objects using GPGPUs. The techniques proposed result in performance gains up to three orders of magnitude, subject to problem and platform.

CHAPTER 1. INTRODUCTION

Rise in volumes of data is unprecedented. The data generated between years 2010-2012 exceeded the total data generated by entire mankind prior to year 2010 [37]. IDC Digital Universe study predicts that by 2020 about 1.7 megabytes of new data will be generated every second for every individual [27]. With ever increasing number of sensors, machine-machine communication, spread of mobile devices, and several other factors, this trend is only going to increase in future. This phenomenal increase in the data is termed as Big Data. The Big Data exhibits several characteristics such as volume, velocity, variety, veracity, etc. The variety characteristics of Big Data has the following forms: Structured, Semi-structured, and Unstructured.

A significant portion of Big Data is semi-structured and is growing rapidly. Tree structured data objects are key component in many applications such as XML databases, bioinformatics, natural language processing, etc. Semi-structured data offers several advantages when compared to its counterparts such as structured forms and completely unstructured forms. For example, ease of representation, etc. Big Data phenomenon is pressing for novel computing approaches for key problems.

Now, we discuss important trends in Big Data, computing, and storage systems.

1.1 Trends in Big Data

Some of the important trends in Big Data, in general, and for semi-structured data, in particular, are as follows: (1) Massive volumes of data, (2) Need for rapid analytics over data sets, (3) Heterogeneous nature of data, and (4) Data has quality issues.

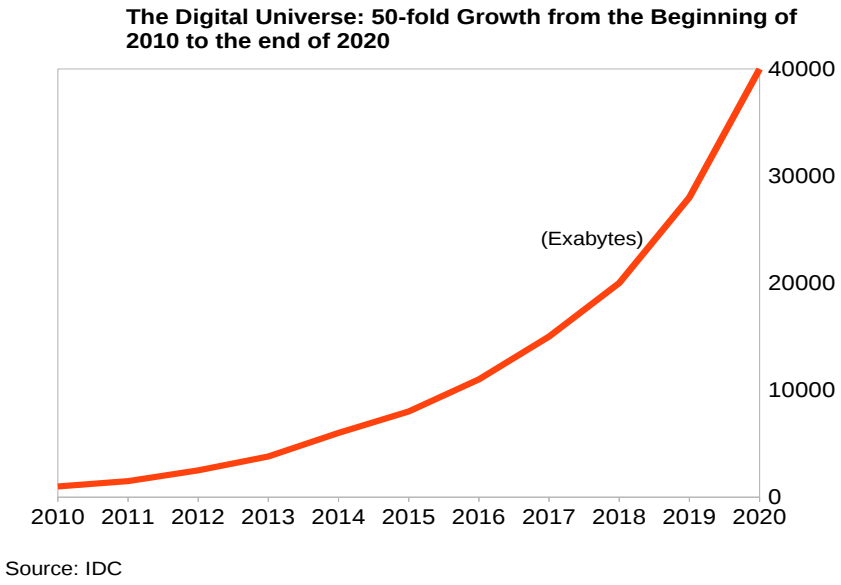


Figure 1.1: Growth of digital information across the universe: trend and forecast. (Source: IDC Digital Universe Study, Dec. 2012).

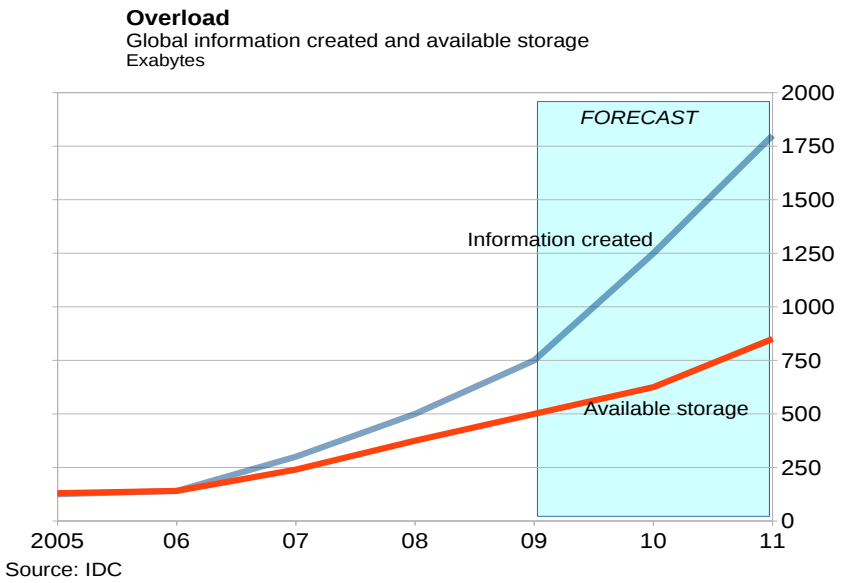


Figure 1.2: Gap between volume of global information created and available storage [39].

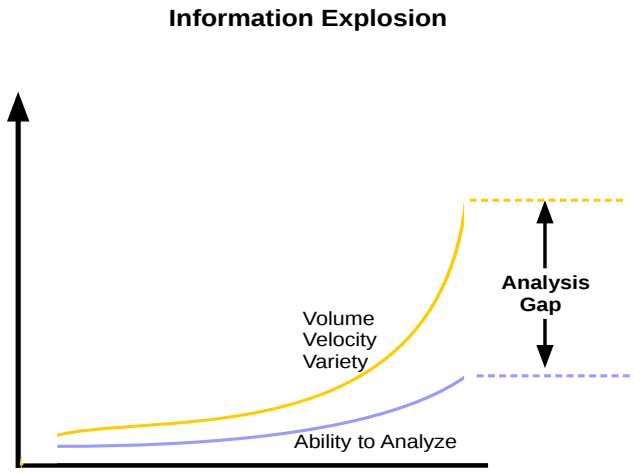


Figure 1.3: A qualitative portrayal of the gap between information created and ability to analyze that information[60].

Massive volume of data. The volume of Big Data in general, and semi-structured data in particular, is increasing at a phenomenal rate. Since, semi-structured data sets do not follow a fixed schema and hence are easier to generate. It is convenient to represent varying types of information in a semi-structured form. In future, the data processing operations are going to deal with a significant volume of data which occurs in semi-structured form. Such semi-structured data sets may vary in structure as well as in size. For example, health-care organizations are collecting and storing gigantic volumes of data. The sheer volume of datasets collected demands intelligent data management approaches in order to identify, store and leverage the most useful data.

Demand for rapid analytics over data. Big data occurs in structured, semi-structured, and unstructured forms. The Big data can be stationary or in motion i.e. as a stream of data. Fast, rapid analytics on Big data is becoming important. Fast analytics are especially important in context of streaming data since the data generated by machines, devices or sensors needs to be processed in a timely manner. Definition of timely processing is context based

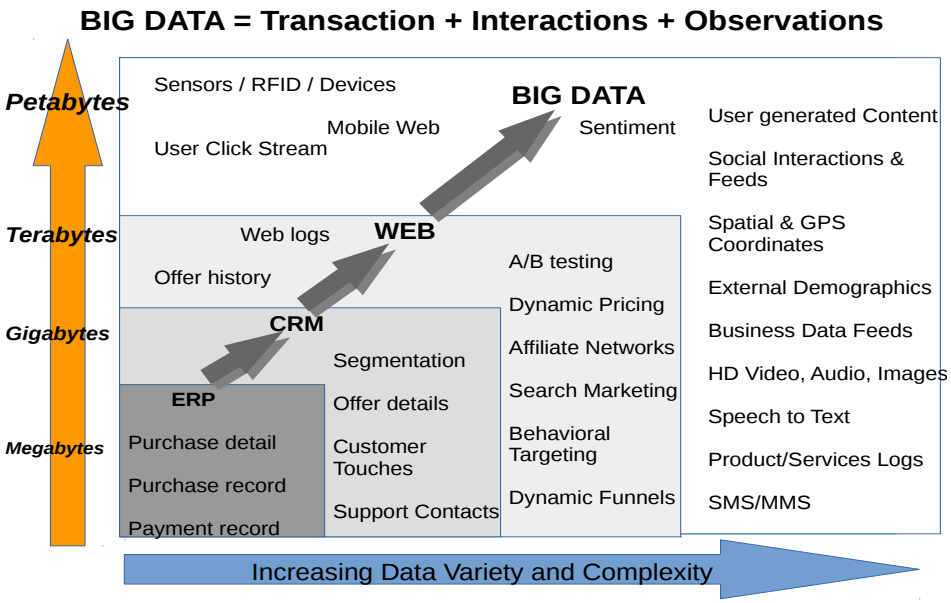


Figure 1.4: Prime constituents of Big data. (Source: Data Magnum, Oct. 2013)[145].

and it could be in the range of few seconds, minutes or several hours. Oftentimes, the data streams originate from multiple sources of data, can be heterogeneous in nature, and need to be processed in (near-) real-time. A recent study envisages that in future organizations have to deal with more and more unstructured and semi-structured data generated from sensors and devices [29]. This study also predicts that in future a majority of workload in organizations will comprise the real-time analytics. Another study reveals that a vast majority of the Big Data remains untapped and only a tiny fraction of Big Data, about 0.5 percent, is ever analyzed [120].

Heterogeneous nature of data. In order to manage data originating from multiple sources data integration is needed. The heterogeneity factor of the multi-sourced data is overcome by employing schema-matching (schema-translation) measures. Oftentimes, data fusion, where multiple records corresponding to same real world object are identified and combined into a single record, is needed.

Data has quality issues. Typically, data in its raw form is not suitable for performing analytics and data needs some preparation. For example, data has quality issues. The poor data quality is a result of many different types of errors occurring due to numerous reasons [117]. Poor data quality can result in loss in business. A previous study estimates that poor data quality can cost businesses close to $600 million each year [38]. The data needs cleaning before it can be used for any analytics. Problems like data integration, data cleaning, data fusion, etc. have to identify duplicate objects. Duplicates are multiple representations of the same real world object that differ from each other because single (multiple) representation(s) is storing erroneous, incomplete information or data is missing.

The trend of generating and processing large volumes of data is likely to continue in future. A lot can be done using this Big Data. As an example, the financial value of Big Data motivates a company to formulate a data strategy policy for managing data. Businesses strive to sift through Big Data sets in order to derive actionable intelligence, which can be leveraged for developing new and useful products, develop new marketing strategies, acquiring new customers while retaining the existing ones, and increasing level of customer satisfaction etc. Similarly, health-care sector is increasingly harvesting the Big Data for better delivery of health-care at various levels. Next, we review trend in computing and storage systems.

1.2 Trends in computing and storage

There are three prominent trends in computing and storage landscape: (1) Growth in Big Data vs. device scaling, (2) Core-level parallelism, and (3) Rise of document databases.

Growth in Big Data vs. device scaling. The rate of growth in data generation outpaces the rate at which memory and transistor count are increasing. The chip performance and drive capacity doubles in only eighteen months. However, the data is growing at a much higher rate. Therefore, a relatively limited storage is available with the computing platforms. A study by International Data Corporation (IDC) estimates nearly 2X gap between the amount of storage available and information being created [39] (Refer Figure 1.4). Therefore, device scaling is not adequate to tackle the volume-borne challenges of Big Data. For example, the volume as-

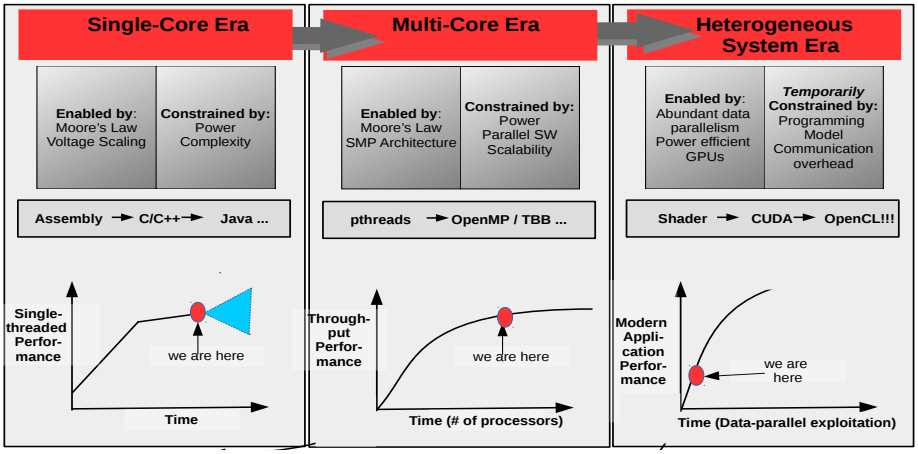


Figure 1.5: Transition from Single-Core to Multi-Core to Heterogeneous Systems [56].

pect and need for rapid analytics over semi-structured complicates the on-line data processing. Likewise, the heterogeneity and quality issues affect the data storage aspect adversely.

Core-level parallelism. There is an increasing trend of deployment and use of hundreds, thousands of processors in parallel. And the processors used vary from Intel's Atom-like [30] smaller cores to Single Instruction Multiple Thread/Data (SIMT/D) like powerful and general purpose graphics processors. For example, almost all the top ten supercomputers harness the compute power of accelerators such has SIMT-based General Purpose Graphics Processing Units (GPGPUs) [102] and Xeon Phi co-processors [31]. As depicted in Figure 1.5 the heterogeneous computing comprising general purpose graphics processors shall continue to drive the high performance (high throughput) computing. Also, GPU based computing systems occupy smaller footprint. For example, NVIDIA stated that Tianhe-1A supercomputer (the worlds fastest supercomputer, October 2010), it would have taken 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone [55]. Refer to Section 2.3 for more on accelerators and co-processors.

Accelerator-based computing. State-of-the-art computing systems rely on parallel architecture machines for compute power. GPGPUs- and MIC-based compute systems are be-

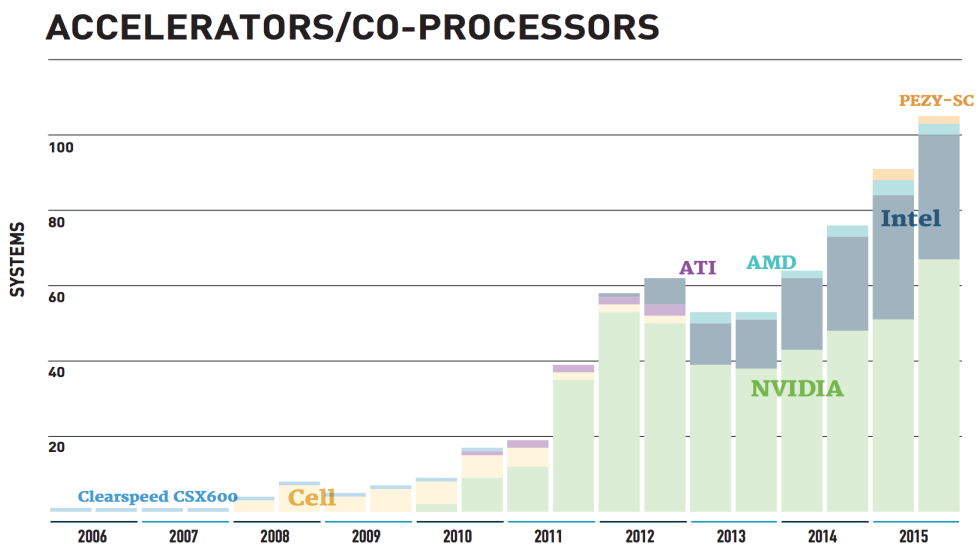


Figure 1.6: Proportion of accelerators/coprocessors in top 500 systems (Courtesy: Top500 release at SC15).

coming quite prevalent. As per TOP500.org release of November 2015, four out of the top ten supercomputers were powered by accelerators such as Intel Xeon Phi or Nvidia K20X [3] (Refer Figure 1.6). The June 2015 release of TOP500.org indicates that five out of top ten supercomputers were powered by accelerators. A current study by Intersect360 predicts that within a year or two the majority of new systems will be equipped with accelerators of some type [123]. Another study from Intersect360 reports that approximately one third of the HPC systems operating till date deploy accelerators [149].

Energy efficient computing. Accelerators and co-processors offer significant energy efficiency advantages when compared to their traditional multicore counterparts. The heterogeneous accelerator-based systems dominate the top places in Green500 list [46]. As per June 2015 release of Green500, top 32 supercomputers were powered by accelerators. Six of the top ten supercomputers used Nvidia's K20 GPGPUs or some advanced version. Back in November 2014, top 23 supercomputers used accelerators. Overall, heterogeneous accelerator-based systems continue to dominate the top places of the Green500.

Rise of document databases. Every single data item (or collection of data items) cannot be represented appropriately using a structured data model. For example, hierarchical documents, graphs, geospatial data, etc. which are extremely useful for a number of aspects such as social network analysis, natural language processing, semantic web analysis, and others. Data management solutions such as BaseX [32], HBase [52], Cassandra [24], etc., are widely used for storing and managing such semi-structured data sets. The document-oriented databases, such as NoSQL (a.k.a. not only sql) databases [93], are built around the concept of a document. Typically, NoSql-type databases encode data in some standard format (encoding) such as Extensible Markup Language (XML) [153], YAML Markup Language (YAML) [70], JavaScript Object Notation (JSON) [58], Binary JSON(BSON) [57], etc. In a document-oriented data storage system, data is stored in its native form. Storing data in its native form is advantageous in several respects. Two such advantages are: (1) It preserves original structure of the data, and (2) It preserves the time-order among the information recorded.

1.3 Motivation

Several data analytic pipelines involve similarity computations, tree comparison operations, etc., as their core components. For example, tree comparison between ordered or unordered trees, finding the edit distance between a given pair of trees (i.e. tree edit distance), etc. Typically, tree comparison problem has quadratic complexity in terms of the number of nodes in the given trees. Further, the structure of tree structured data also presents processing challenges. As volume of semi-structured data sets (e.g. XML, JSON, etc.) increases and real-time data analytic over such data sets become indispensable, there is a need to leverage parallel processing platforms effectively. For example, we can accelerate the tree matching component using parallel hardware such as a General Purpose Graphics Processing Units (GPGPUs) effectively.

We make several key observations in the context of Big Data processing on parallel architecture machines:

- Big Data processing problems are compute- as well as data-intensive. And state-of-the-art parallel architecture machines are rich in compute resources and thus a good fit for

the compute intensive component of Big Data problems. However, the compute units of parallel machines must be fed data timely and appropriately to gain any meaningful speedup. Also, the memory resources of parallel machines need to be used effectively as some of them, for e.g. GPGPUs, have memory hierarchy limitations.

- Aligning data items in certain manner can help overcome the challenges posed by computing platform. For example, data alignment followed by appropriate orchestration of the computations can overcome complexity of the solution to a given problem. And a significant amount of speedup is possible. For example, if the data items are already present close to the cores (streaming processors) working on those data items, then additional computations can be launched almost for free.

We argue that effective and efficient solutions to key similarity computation problems need to operate in a synergistic manner with the underlying computing hardware. Moreover, semi-structured form input data needs to be shaped (reorganized) with a goal to exploit the enormous computing power of *state-of-the-art* highly threaded architectures such as GPUs, MICs, etc.. For example, shaping input data (via encoding) with minimal data-dependence can facilitate flexible and concurrent computations on a high throughput GPGPU accelerator.

1.4 Research Goals and Contributions

To demonstrate applicability, viability, and efficacy of the data shaping approach, we consider several instances of traditional and futuristic problems occurring at the intersection of semi-structured data and data analytics. For example, twig matching is widely used in asynchronous communication paradigm, duplicate detection is a core component in record linkage, data mining, etc. Likewise, tree edit distance computation is a fundamental metric used in tree comparison problems, and similarity computation between trees another key problem in context of Big Data.

1.4.1 Research goals

We address these problems from a different perspective–this dissertation argues for a data shaping as a potent mechanism to tackle the data-borne and architecture-borne issues. Specifically, the research goals are as follows:

- To develop a data shaping technique to address the problem of elastic publish-subscribe system in heterogenous cloud environment. We address the research problem in the following manner: Design a data shaping technique which enables application elasticity and portability in heterogeneous cloud environment.]
- To study data pre-processing techniques in context of duplicate detection problem enabling efficient usage of a GPGPU.
- To develop a data shaping technique to parallelize the computation of tree edit distance.
- To develop a data shaping technique to accelerate the delta encoding pipeline and alleviate the performance bottlenecks of existing delta encoding solutions.

1.4.2 Research Contributions

This research makes the following contributions:

- We develop a data shaping-based technique to enable an elastic and portable publish subscribe mechanism for heterogeneous Cloud environment. Our data shaping technique enables large number of work (kernels) resulting in efficient utilization of compute resources of a highly parallel architecture machines. Experiments with real-worlds datasets indicate that our data shaping-based solution achieves a filtering throughput of more than thirty MB per second (for a set of one hundred twenty eight queries).
- We develop a set of data shaping-based heuristics to reduce computational complexity and increase avenues for parallel processing in duplicate a duplicate detection problem. Experiments with real-world data sets show speed-up of up to 8X over state-of-the-art schemes, while maintaining up to ninety two percent accuracy. In addition, our data

shaping technique for GPGPU processing speeds up the duplicate detection throughput by up to two orders of magnitude.

- We develop a novel data shaping algorithm to identify substructures (subtrees) within a tree which enables parallel processing of intermediate computations involved in tree edit distance computation. Experiments reveal that our data shaping approach results in a speedup of up to 12X when compared to the state-of-the-art in tree edit distance (TED) computation.

- We develop a novel technique to compute degree of similarity in tree-structured data via identifying the similarity patterns in context of delta encoding problem. Experiments indicate that our similarity pattern-based approach achieves an overall compute throughput of more than six hundred document comparisons per millisecond.

1.5 Dissertation Organization

This dissertation is organized in seven chapter. Chapters 2-6 describe the problem and research conducted.

Chapter 2 provides a review of existing technique for efficient processing of Big Data using accelerators-based systems. Chapter 3 describes data shaping-based elastic and portable publish-subscribe system for heterogeneous Cloud environment. Chapter 4 describes context-aware duplicate detection on SIMT using data shaping. Chapter 5 describes computation of tree edit distance on SIMTs using data shaping. Chapter 6 describes computation of document similarity using parallel architecture machines. Chapter 7 concludes this dissertation and provides future outlook.

CHAPTER 2. LITERATURE REVIEW

2.1 Introduction

In the light of phenomenal increase in volumes of data, it is important to leverage the compute power of accelerator-based computing systems. However, effective usage of such parallel architecture machines to meet the compute and memory requirements of Big Data workloads is not easy. Reason being the challenges arising due to two factors: (1) mismatch between highly structured nature of machines and irregular form of data workloads, and (2) architectural features of the accelerators.

In this chapter, we first discuss the idea of data shaping and its advantages, and prominent parallel architectures machine and Big Data processing using these machines. We especially focus on techniques developed for GPGPUs and MIC-based architectures. Next, we discuss GPU-specific data reorganization approaches from literature. We focus on data reorganization technique such as data linearization, data compression, data reuse, etc. Finally, we discuss data reorganization approaches proposed in context of semi-structured data where we focus on approaches such as data reorganization, etc. which help utilize parallel processing capabilities of the underlying machines.

Chapter Organization. Rest of the chapter is organized as follows: Section 2.2 discusses the idea of data shaping. Section 2.3 covers parallel architectures machine and Big Data processing using these machines. Section 2.4 deals with GPU-specific proposals on data reorganization. Section 2.5 reviews work on reorganization of semi-structured data for efficient processing. Section 2.6 provides conclusion.

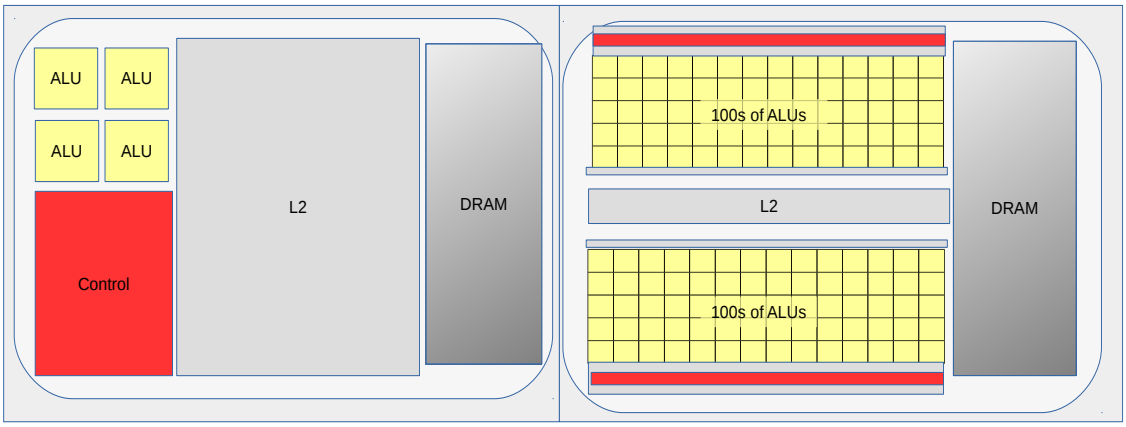


Figure 2.1: A conceptual comparison of distribution of compute, memory, and control hardware in CPU, GPU (Courtesy: code.msdn.microsoft.com).

2.2 Defining Data Shaping

We define data reorganization a.k.a. shaping as the process of transforming (reorganizing) the given input data in order to facilitate the processing of the same on a target computing platform. In order to be effective, efficient, and feasible, a data shaping technique must operate in a synergistic manner. The data shaping process must take into account the nature of the given input data, type of processing to be done on given data, and limitations and capabilities of the target computing platform.

2.2.1 Examples of Data Shaping

Mechanisms such as data compression, data encoding, data reduction, data shedding, etc can be effectively exploited for the purpose of data shaping. For example, if transferring data is a limitation due to limited I/O channels or memory bandwidth, while computing is cheap at source as well as sink of data, then data sets can be compressed (decompressed) at the data source (sink) and transferred. However, applying data compression involves several issues. For example, compressibility factor i.e. reduction in the size of data set after being compressed, depends upon the data set. And the time overhead incurred during compression

may not be acceptable in real-time data processing scenarios. Hence, feasibility of a data shaping technique(s) is (are) totally context dependent. Alternatively, the feasibility of a given data shaping technique depends entirely upon the factors such as nature of data, type of data processing, and the target computing platform, etc.

Consider proper alignment of data enabling coalesced memory accesses as another example of data shaping. Proper alignment of data and coalesced memory accesses affect the performance of any given algorithm irrespective of the underlying hardware. An appropriate data shaping technique which can work with initial set of input and ensure that the data being accesses by algorithm always results in coalesced transactions help in extracting a much higher throughput from any given architecture.

2.2.2 Advantages

Data shaping can help neutralize the computational complexity of data preprocessing tasks such as data filtering and duplicate detection. This is possible by increasing the usage of data accessed i.e. ensuring a high data reuse factor. If data items are already present close to the cores (i.e. streaming processors of a GPU) working those data items, then additional computations can be launched for nearly free of cost. In other words, we can exploit the temporal locality to address the issue of computation overhead. However, exploiting data locality needs an appropriate algorithm. Also, there is an implicit trade-off between the data reuse and computational complexity of the algorithm deployed. For example, we would like to maximize the data reuse factor for every data item fetched to the on-chip memories. But, the algorithm used should enable such computations.

2.3 Parallel Architectures and Big Data Processing

First, we review key architectural features of the parallel computing machines. We cover GPGPUs, MICs, and others. We also analyze the challenges associated in processing Big Data workloads, especially semi-structured data, on such parallel machines.

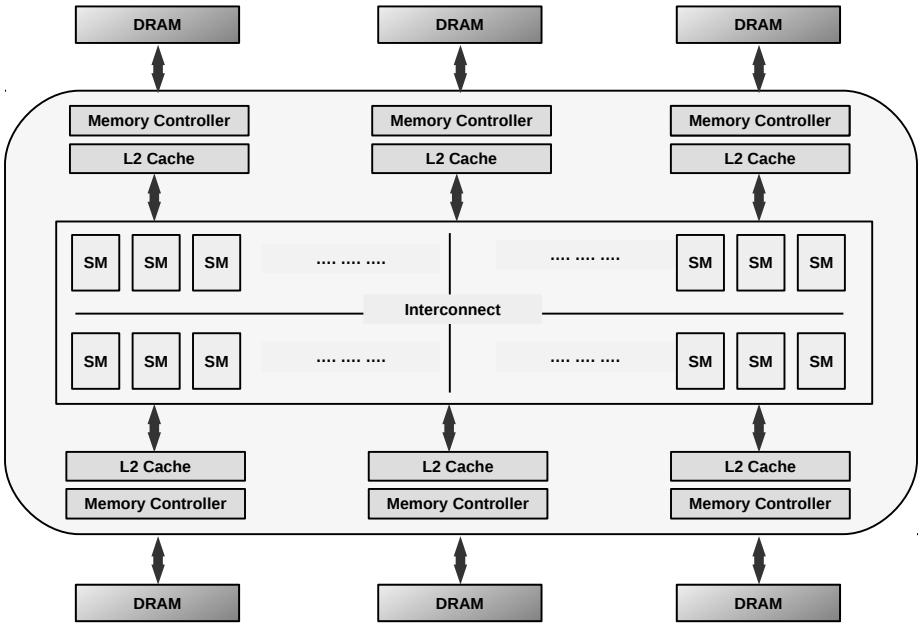


Figure 2.2: Schematic of a typical General Purpose Graphics Processor Unit (GPGPU). (Courtesy: cinwell.wordpress.com)

2.3.1 SIMT Machines or GPGPUs

2.3.1.1 The Architecture

Single Instruction Multiple Thread (SIMT) machines comprise hundreds, thousands of compute cores, large register files, and on-chip and off-chip memories. General Purpose Graphics Processing Unit (GPGPU) is an example of SIMT machine.

On a GPGPU, 32 Streaming Processors (SPs) or cores can execute in parallel within a Streaming Multiprocessor (SM). They however provide small amount of special purpose on-chip memories that include Shared Memory, Constant Memory (CM), and Texture Memory (TM) besides the regular L1/L2 cache memories. However, in order to use the fast on-chip Constant Memory (CM) effectively, it is necessary that at-least half of the threads within a Thread Warp (or Thread Block) read from the same address in Constant Memory. If threads within a half Warp access more than one address in the CM, then access is serialized which results in higher memory access latency compared to the off-chip global memory.

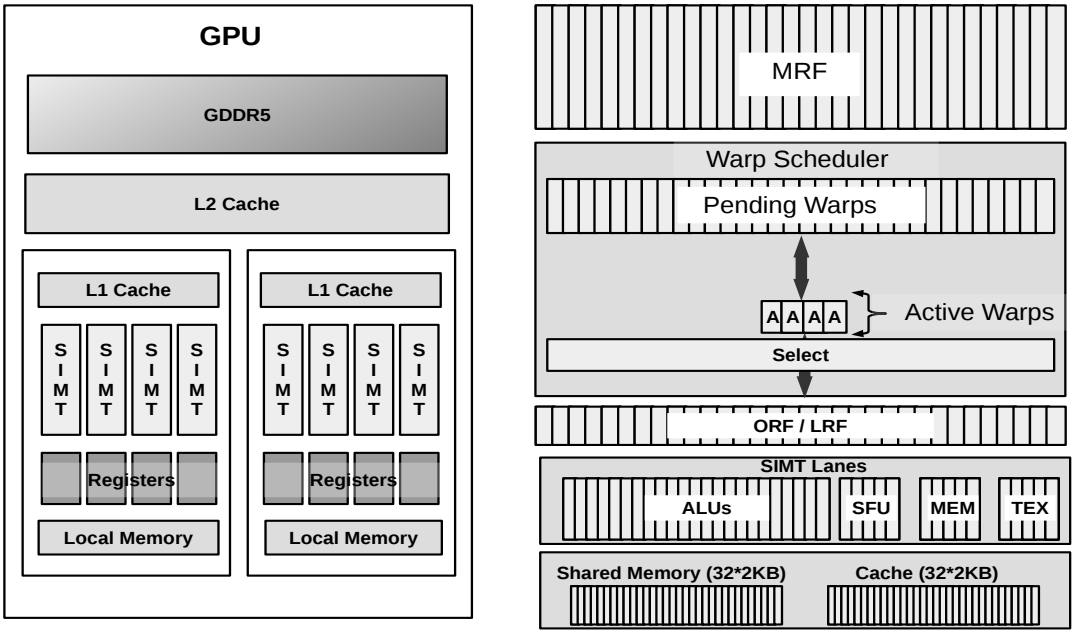


Figure 2.3: Architecture of a typical SM. (Courtesy: cinwell.wordpress.com)

Compute to global memory access (CGMA) ratio is defined as the number of floating point calculations performed for each access to the global memory within a region of a CUDA program. *Register pressure* i.e. number of registers required by a SIMT kernel is related to the size of computation performed by the kernel. A large number of data fetches and computations increase the register pressure. A *coalesced memory transaction* is one in which all of the threads in a half-warp access global memory at the same time. A way to do it is to have consecutive threads access consecutive memory addresses. Coalesced memory transaction or *memory coalescing* is crucial to obtain any significant gain in performance (or throughput) from a SIMT machine.

2.3.1.2 Analysis of data processing on SIMT

Big Data tasks involving semi-structured data are typically both compute as well memory intensive. For example, tree matching problem, a common component in various data analytic pipelines, is compute intensive involving quadratic order pair-wise comparison opera-

tions. Parallel machines like GPGPU offer tremendous processing power. High computational complexity of the problem motivates us to use highly parallel machines like General Purpose Graphics Processing Unit (GPGPU) for compute intensive problems.

A key observation being that if the data items are already present close to the cores (streaming processors) working on those data items, then additional computations can be launched almost for free. This increases the data reuse which results in a higher CGMA. The memory bandwidth required and system wide data movement are further constrained by energy consumption and power dissipation limits. Reduced system wide data movement allows increased power allocation to computation cycles. However, increasing data reuse factor has implications on the algorithm considered.

Another important observation is that aligning data items in certain manner can help overcome the challenges posed by computing platform. For example, data alignment followed by appropriate orchestration of the computations can overcome complexity of the solution to a given problem. And a significant amount of speedup is possible. However, GPGPUs possess architectural features which make tree processing challenging. (1) Lock-step execution of the streaming processors (SPs) in streaming multiprocessor (SM). (2) High memory access latency associated with device memory (or main memory on a GPGPU) (3) Limitations of memory hierarchy of the GPGPUs, for e.g., data access constraints associated with constant memory.

It is possible to overcome the architecture-specific processing challenges by addressing issues such as: (1) Size of the computation executed in parallel, (2) Size of the data item to be processed on parallel, (3) Pattern and order of accessing data.

2.3.2 Many Integrated Core Architecture

2.3.2.1 The Architecture

Intel's Many Integrated Core (Intel MIC) architecture basically combines several Intel CPU cores onto a single chip. Xeon Phi coprocessors are an example of Intel's Many Integrated Core Architecture (MIC) machines. The first generation Intel Xeon Phi products were codenamed as Knights Corner (KNC). The KNC coprocessor have a large number cores, over 50, when

compared to traditional multi-core processors such as Intel Xeon processor. The advanced versions of Xeon have higher core counts. The second generation of Phi is codenamed as Knights Landing (KNL) and has over 60 cores. The third generation of Phi is codenamed as Knights Hall (KNH) is expected to have even a higher number of cores. A high-performance on-die interconnect used for connecting these cores.

The Xeon Phi cores are equipped with caches and vector processing capabilities. To be specific, the Xeon Phi cores contain a 512-bit wide vector unit with the vector register files (32 registers per thread context). The 512-bit SIMD unit can work on 8 double precision or 16 single precision data elements in a simultaneous manner. Each core on Xeon Phi has a 32KB of L1 data cache and instruction cache each. There is also a core-private 512KB unified L2 cache. This amounts to a total of 30MB on the die.

The Intel Xeon Phi coprocessor and the Intel Xeon multicore processor have some similarities. For example, the Xeon Phi coprocessor inherits key architectural features, namely, SIMD units and cache-based memory systems, from traditional Intel Xeon processors. One key difference between Xeon Phi coprocessor and Xeon multicore is how L2 cache, the last level cache is used. The Xeon Phi uses the L2 cache as the last-level wherein each cache slice is private to a core. This is in contrast to the shared L2 cache on Xeon multicore processors where L2 cache is shared between two cores. Another key difference between Xeon Phi coprocessors and Xeon multicore processors being that the Xeon Phi uses graphics memory, a 6 GB of GDDR5. Provisioning of graphics memory results in a much higher bandwidth on Xeon Phi, up to 6X, when compared to bandwidth on the Sandy Bridge Xeon processor.

2.3.2.2 Analysis of data processing on MIC

The data processing performance on Xeon Phi degrades if data elements being accesses are not aligned contiguously in memory. Same hold for the Xeon processors in general. Poor performance when data elements are not contiguous in memory is not a new observation. There could be a several reasons responsible for this. For example, the non-contiguous memory access is affects the efficiency of memory bandwidth adversely which affects the performance of memory-bound kernels. Non-contiguous memory access also disturbs data prefetching gains.

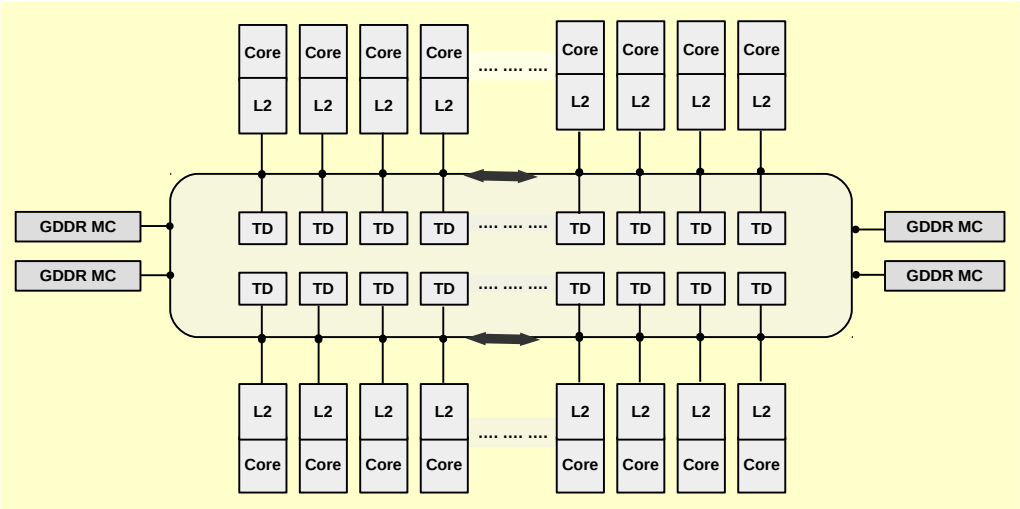


Figure 2.4: A high level view of MIC architecture.

Variations in cache architectures can lead to different memory access behaviors for specific type of memory constructs in the code. For example, multiple threads updating a shared memory space can lead to a significance degradation in performance. This is because this type of exclusive memory access results in repeated cache-line invalidations or transfers between the cores i.e. remote cache access. However, the remote cache access characteristics could be different in Xeon Phi and Xeon. This is due to the variation in cache architectures, namely the last level cache, found on the two machines. Specific reason is the fact that all cores on Xeon share the last level of cache (L3). But, on Xeon Phi the last level cache (L2) is private to a core and L2 is distributed among cores.

In MIC architecture, it is not possible to issue instructions from the same thread context in consecutive cycles. This feature can have significant implications on the performance. Consider the case of a highly optimized vector code written in intrinsics (to ensure 100 percent of vector usage). For a case like this, the instruction throughput is going to be low as issue width becomes the bottleneck. In order to take full advantage of the hardware resources, we have to run at least two threads per core.

2.4 GPU-centric data reorganization approaches

In this section, we cover important studies which deal with processing semi-structured data sets and filtering over semi-structured data sets.

2.4.1 Tree processing using GPUs

Kim et al. present a fast architecture sensitive tree (FAST) search algorithm in [61], [62]. FAST is a binary tree, managed as hierarchical tree wherein elements of the tree are rearranged based on various attributes of the underlying machine architecture such as page size, cache line size, etc. We observe that in-memory tree structured search problem in context of database operations is somewhat different from the data stream processing problem which is the focus of our work.

A key difference is that in our work updating of trees is not required. Size of the dataset to be processed is unbounded and not known a-priori. And it is not required to maintain the tree objects for future references which allows us to search, glean information, and then discard the dataset. Another key difference is that semi-structured data such as tree objects are not restricted to be k-ary, with a fixed k.

2.4.2 Algorithmic approaches for tree-model data

We review some of the notable algorithmic approaches proposed in context of XML filtering.

2.4.2.1 Moussalli et al.

Moussalli et al. propose using GPU for filtering the XML data streams [95]. Their proposal filters path queries over a stream of XML data streams. The proposal uses dynamic programming approach in order to filter the path queries. Queries and input data stream is pre-processed and represented in a compact manner. The pre-processed queries are represented as an array. Each entry in the array represents a query node. And each entry corresponds to a single kernel instance in GPU.

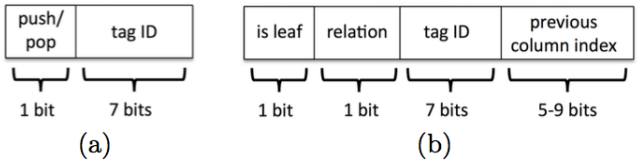


Figure 2.5: Preprocessed Tree input data. (a) representation of XML document event. (b) representation of XML query on GPU

Figure 2.5 shows the alternate representation of input method used in their work. Figure 2.5(a) shows the representation of events in input document i.e. opening and closing of a tag. Figure 2.5(b) shows how query is represented in GPU.

The input tree documents are read-only. The GPGPUs provide some on-chip constant memory. The authors perhaps also wanted to exploit the constant memory, a special type of on-chip cache, to speedup the query processing. However, due to limited size of the constant memory only minimal speedup was observed. Finally, they ended up with using GPGPUs device memory. The input document were processed using device memory i.e. global memory which offers much larger capacity when compared to constant memory.

2.4.2.2 Shnaiderman et al.

Research in [128] leverages GPU to accelerate the job of processing a single query in XML databases. The goal is to exploit some key features of GPU, for example, massive parallelization on GPGPUs which is order of magnitude higher than CPUs, high bandwidth of a GPGPU to transfer data from the global memory to the local memories which is again more efficient than data transfer from main memory to the CPU.

To realize these goals, authors use the XML stream representation scheme to represent the an XML document (note that it is different from data streams). In XML stream representation, there is a dedicated stream for each element label in document. The streams contain some defining information about the node, i.e., positional representations of XML nodes, etc. The authors extend the original stream representation scheme and added two additional fields: (1) A list of all streams that contain ancestors of the node, and (2) A boolean array whose size equals the number of nodes in the twig pattern. The extended stream representation consumes

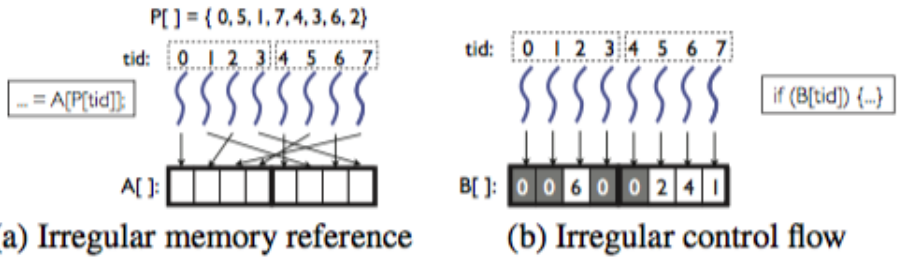


Figure 2.6: Illustration of the problem of dynamic irregularities (warp size=4; segment size=4). (a) depicts the problem of increased memory transactions due inferior mappings between threads and data locations. (b) depicts the problem of threads in the same warp diverging on condition due to inferior mappings between threads and data values.

~4.2X the space that is consumed by the basic stream scheme. The authors argue that a ~4.2X increase in space requirement is acceptable for real-life applications. The streams are stored compactly in memory as arrays of structs. The compact storage facilitates the fast access to each tree node which is supposedly convenient for GPGPUs.

The twig processing algorithm is proposed to take advantage of the stream representation scheme. The algorithm divides the work to hundreds or even thousands of threads and enables parallel processing. This work does not makes use of advanced feature of GPGPUs such as shared memory or coalesced memory access. The algorithm is not compute-intensive. The global memory bus bandwidth is the performance bottleneck.

The algorithm operates in two phases. It processes only parts of document having relevancy to the query. In first phase, the algorithm iterates over all the relevant streams deriving information about the structure as per the input query. The second phase of algorithm involves using the structural information (derived earlier) to generate answer set.

2.4.2.3 Other Hardware and Software-based Approaches

A mixed hardware/software approach for filtering simple XPath queries is proposed in [144]. Queries having only parent-child axis are handled by this approach.

A pure hardware approach for XML filtering proposed [90]. The filtering approach in [90] cannot handle recursion in XML documents.

2.4.3 Algorithms for Data Reorganization

2.4.3.1 Zhang et al.

Zhang et al. study the problem of dynamic irregularities occurring in both control flows as well as in memory references [158].

To address the problem of dynamic irregularities, Zhang et al. present heuristics- based algorithms and runtime adaptation techniques and framework called G-Streamline. The dynamic irregularities are removed through the transformation techniques – data reordering and job swapping.

Data reordering. Consider an irregular reference such as A[P [tid]]. The Data reordering algorithm creates a new array A[tid] where A[tid] = A[P[tid]]. The memory references in kernel code are also redirected, and references to A[P[tid]] are replaced by A[tid]. By these set of transformations, all memory accesses made by a warp become contiguous. And the non-coalesced memory access problem is solved.

The size of new array A is as large as the number of GPU threads i.e. T, and no longer depends upon the size of the original array A. Also, a new array of size T is created for each reference to the same array. For example, two arrays are created for memory references like this: A[P [tid]] + A[P[tid] + v].

The Data Reordering algorithm is also referred as *Duplication* Algorithm. The reason being that it makes duplicated copies of a data element if the data element occurs multiple time in the indexing array P.

2.4.3.2 Wu et al.

Wu et al. study the problem of data reorganization to to eliminate the non-coalesced memory accesses that are a result of irregular memory references [150]. The study analyzes the inherent complexity of the data reorganization problem and proves that the problem is NP-complete. And points that the design of an appropriate data reorganization solution is nothing else but a tradeoff among three space, time, and complexity. Wu et al. propose two algorithms for data reorganization: (1) Padding, and (2) Sharing.

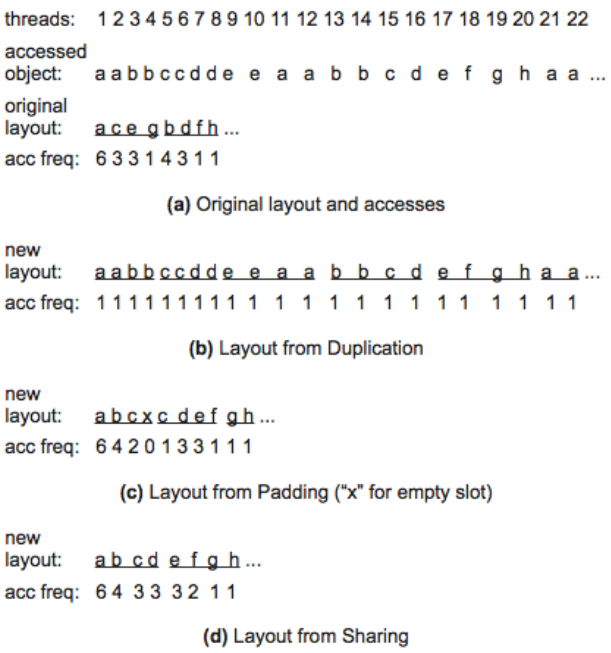


Figure 2.7: Illustration of data layouts generated and the access order in (b) duplication, (c) padding, and (d) sharing algorithms. (a) depicts the original layout and accesses. Note that the example shown assumes four objects per memory segment, four threads per warp, and four warps per block.

Padding Algorithm. The Padding algorithm seeks to reduce the number of data copies made in the duplication algorithm while also retaining the optimization quality. Key observation behind the Padding algorithm is that if threads (t1 and t2) belonging to the same warp are accessing the same data element (a), then we need not create two copies of data element (a). Instead, the threads can be allowed to access the same copy of data element (a). This approach avoids creating non-coalesced accesses as both threads are still accessing the same segment of memory. However, the Padding approach changes the one-to-one type of regular mapping between data and threads provided by the duplication algorithm [158].

Sharing Algorithm. The Sharing algorithm overcomes the limitation of the padding algorithm. This is realized by leveraging the shared memory available in GPGPUs. Shared memory is used to increase applicability of duplication avoidance. The Sharing algorithm exploits several features of shared memory such as writes are visible only to threads of same thread block, low access latency when compared to that of global memory, performance of shared memory being insensitive, to some extent, to irregular memory accesses.

The Sharing algorithm operates as follows:

(1) Create a copy of the data accessed by a thread block and place the copy in a consecutive chunk of memory.

(2) Load the data in a consecutive fashion into shared memory thus ensuring the memory coalescing effect.

(3) Redirect the memory accesses made by the threads in the thread block to the respective copies in the shared memory (instead of global memory). The algorithm uses clustering mechanism to further increase the sharing opportunity avoiding duplications.

The Sharing algorithm basically shifts the irregular accesses from global memory to shared memory. Since, the shared memory is visible to all the threads of the thread block, the scope of sharing is now no longer limited up to the warp level (as provided by Duplication algorithm), and is increased to the thread block level. In other words, the scope of duplication avoidance is now a thread block and not just a warp.

```
GPU-side code
clusterKernel(particles, numP, clusters) {
   for( i = 0; i < numP; i++)
      particles[i].cid = findClosestCluster(
               particles[i].x, particles[i].y,
               particles[i].z, clusters);
}
```

Figure 2.8: A CPU code for K-means clustering.

2.4.3.3 Mokhtari et al.

BigKernel [92], a compiler and runtime technique to address several challenges associated with data processing involving GPGPU. The BigKernel also addresses the problem of uncoalesced memory accesses occurring in Big Data-style computations. The key idea behind BigKernel is as follows: GPU threads identify the data they will be accessing in their computations online (Note that GPU threads do not access the data and do not perform any computation at this point). GPU transfers this information to CPU. CPU assembles the data and then transfers to the GPU memory. Now, GPU accesses the data in their computations.

The BigKernel technique operates as a four stage pipeline:

(1) Prefetch address generation (GPU side) when GPU threads calculate the memory access information and record it in an address buffer. This address buffer is sent to the CPU for data assembly.

(2) Data assembly (CPU side) – CPU assembles the prefetch data based upon the information contained in address buffer.

(3) Data transfer – the assembled prefetch data is transferred to the data buffer on the GPU side by the DAM engine.

(4) Kernel computation – the GPU threads access the data values from data buffer instead of original memory locations and do the computations. The kernel code on GPU is also transformed to accommodate this redirection.

The BigKernel technique also provides a notion of pseudo-virtual memory to GPU application. The BigKernel technique simplifies programming model wherein programmers can write kernels using arbitrarily large data structures that can be partitioned into independently operable segments.

GPU-side code

```
clusterKernel(particles, numP, clusters) {
   start = myParticleStartIndex( threadId, numP );
   end = myParticleEndIndex( threadId, numP );
   for(i = start; i < end; i++)
      findClosestCluster(particles[i].x,
      particles[i].y, particles[i].z, clusters);
}
```

Figure 2.9: A GPU code for K-means clustering.

GPU-side code

```
counter = 0;
for( i = start; i < end; i++ ) {
   addrBuf[counter++][threadId] = &particles[i].x;
   addrBuf[counter++][threadId] = &particles[i].y;
   addrBuf[counter++][threadId] = &particles[i].z;
}
```

Figure 2.10: GPU code for prefetch address generation (Stage 1) [92].

GPU-side code

```
counter = 0;
for( i=start; i < end; i ++ )
   findClosestCluster (
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            clusters
            );
```

Figure 2.11: Transformed GPU code for kernel computation (Stage 4) [92].

2.4.3.4 Goldfarb et al.

Goldfarb et al. propose general transformations for GPU execution of tree traversals [43]. The authors describe general-purpose techniques which can be used to implement irregular algorithms on GPUs. A key feature of these techniques is that they exploit commonalities present in structure of algorithms instead and do not rely upon the application-specific knowledge.

A transformation called *auto-roping* in developed. The idea of auto-roping is based on a key observation. The key observation being: primary costs involved in general tree traversals on GPUs is due to cost of frequent moves up and down the tree during the traversal process.

Typically, the tree traversal process involves stack-management. Some studies develop "stackless" traversals which encode the traversal orders into the tree itself via auxiliary pointers, called "ropes". The provision of ropes avert the need for any stack-management. However, the task of encoding ropes into the tree is not trivial and involves tradeoffs. Reason being that it involves development of algorithm and implementation-specific passes which may result in dropping generality in favor of efficiency. Auto-roping is basically a generalization of ropes applicable to any recursive traversal algorithm.

2.4.4 Algorithm design model for GPUs

Some studies focus on problem of designing parallel algorithms for GPUs. Example is the study in [65], which provides parallel algorithm design model for GPU architecture. This model addresses limitations of previous parallel models proposed by accounting for unique architecture of GPU machines, such as diverse types of memories and their characteristics.

The work in [65] builds upon a set of previous works namely PRAM model [42], BSP model [142], and parallel phase model [155]. PRAM model [42] or Parallel Random Access Machine model is one of the earliest parallel algorithm design models. A prime limitation of the PRAM model is that it makes some unrealistic assumptions such as a zero communication overhead and instruction-level synchronization. Another limitation of PRAM model being it expresses the time complexity of an algorithm in big-O notation, which is misleading at times especially if the machine size n is small in parallel computers.

The bulk-synchronous parallel (BSP) model was proposed in order to overcome the limitations of PRAM model [142]. THE BSP model is essentially a Multiple Instruction Multiple Data (MIMD) in nature. It uses a notion of a super-step wherein a super-step comprises three steps: (1) Computation step, (2) Communication step, and (3) Synchronization step. The BSP model captures all overheads missing in PRAM model except one which is the parallelism overhead incurred in process management. Recently, authors of [142] proposed an improved version of BSP model for multi-core CPUs [143].

The parallel phase model builds upon the PRAM model and BSP model [155]. The parallel phase model accounts for various overheads due to load imbalance, interaction, and parallelism, and is closer to real machine/program behavior when compared to the previous models (PRAM and BSP).

These approaches focus on designing algorithms or providing guidelines for designing an algorithm for parallel machines. However, our work focuses on a different aspect: Designing a tree encoding algorithm and guidelines for designing a tree encoding algorithm for efficient computation on GPUs.

2.5 Reorganization of Semi-structured Data

We discuss the existing work related to the proposed research. Specifically, we describe (1) Tree encoding techniques proposed in the literature, (2) Semi-structured data stream filtering systems based on tree encoding, and (3) Platform-centric or architecture specific data processing approaches.

We describe the mechanism for encoding tree model data and how the data encoding mechanism helps overall in processing of trees in context of the algorithm(s) used.

For parallel machine part we are interested in how the problems exploit the compute and memory resources available. For example, for a GPGPU like machine, we want to study how the constructs in algorithm leverage the hardware constructs present in the machine.

2.5.1 Encoding Tree Model Data

2.5.1.1 Eduardo et al.

Eduardo et al. provide a comparison of tree encoding schemes in context of evolutionary algorithms [23]. The authors considered six classical algorithms characteristic vector, Prufer numbers [108], Network random keys [117], Edge sets [115], Node biased encoding [99], and Link and node biased encoding [99] for comparison. Eduardo et al. basically study time efficiency and applicability of tree encoding schemes for genetic algorithms (GA).

2.5.1.2 Picciotto

In [113] Picciotto describes three types of codes: Happy, Blob, and Dandelion. These codes are basically Transformation codes and based on a set of previous results.

2.5.1.3 Micikevicius et al.

A linear-time algorithm for encoding/decoding trees as sequences of node labels is proposed by Micikevicius et al.[88]. The proposed algorithm encodes all the known encoding types of labeled trees such as Deletion (or Prufer) codes and Transformation codes i.e. Happy, Blob, and Dandelion codes. The Deletion codes iteratively delete leaf nodes, recording their neighbors to the code. Whereas, Transformation codes convert trees into directed graphs and then record the nodes as parents to the code.

2.5.1.4 Deo

An algorithm for encoding finite labeled tress is proposed [34]. The proposed algorithm establishes a one-to-one mapping between trees of order n and (n - 2)-tuples of the node labels. The encoding allows to efficiently determine the center(s), radius, and diameter directly from the code, avoiding any construction of the tree.

2.5.1.5 Caminiti et al.

Caminiti et al. consider the problem of coding labeled trees by means of strings of node labels [23]. The authors show that both coding and decoding can be reduced to integer (radix) sorting for different codes like Prufer, Neville, and Deo and Micikeviius. The sequential coding and decoding schemes require optimal O(n) time when applied to n-node trees. And the parallel version of these schemes work in O(log(n)) time.

2.5.1.6 Zhuang

A recent study proposed a full tree-based (FT) encoding technique for dynamic XML labeling schemes [163]. The FT encoding technique generates the codes in the encoding table sequentially and avoids need for Encoding Table while XML initial labeling.

2.5.2 Tree encoding techniques in context of data filtering/querying

Several tree-transformation (-encoding) techniques have been proposed in context of data filtering/querying.

2.5.2.1 NETS

A Node Encoded Tree Sequence (NETS) is proposed [124]. NETS models an XML document as a rooted ordered labeled tree such that each node corresponds to an element tag, attribute or value. The structural relationship between these nodes is represented as edges.

The NETS encoding works as follows: For every node in a given tree, the NETS of the tree is two symbols referred to as "start-symbol" (S) and "end-symbol" (E). Consider a tree T with a node having label "x", the NETS(T) i.e. encoding of T is "start-symbol" (Sx) and "end-symbol" (Ex) for a node with label "x". The Node Encoded Tree Sequence of T, NETS(T), is defined recursively. An example of a XML tree T and its corresponding NETS is shown in Figure 2.12 (a).

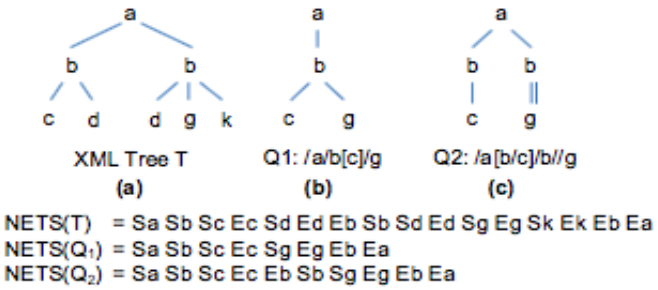


Figure 2.12: Encoding of Tree data using NETS representation.

2.5.2.2 PRIX

PRIX [119] and FiST [69] use Prufer sequence encoding for filtering twig. FiST first transforms XML document and queries into their Prufer sequence, and then carries out subsequence matching to find matches. FiST supports ordered query matching.

2.5.2.3 ViST

ViST [146] transforms XML tree pattern into structure encoded sequence and uses rootpath encoding where each node is expressed as a sequence of intermediate nodes from root to itself. The worst-case storage requirement can be higher than linear in the total number of elements of the XML documents. A key difference between our tree encoding approach and [146] is that we encode selective nodes and storage space requirement in our case is O(n).

2.5.2.4 Dewey Labelling, ORDPATH, Region Encoding

Dewey ID labeling to represent XML order in relational data model is proposed in [135]. ORDPATH, a variation of prefix labeling, is proposed [105]. ORDPATH scheme leverages Dewey ID labeling scheme to preserve document order during XML query processing. ORDPATH deals with insertion of XML nodes in the database. The region encoding is introduced in [26], where focus is on searching and indexing the text database.

2.6 Conclusion

State-of-the-art computing systems as well near-future high performance systems comprise heavily parallel cores. Big data workloads comprise semi-structured data in significant proportions.

Big Data phenomenon is pressing for novel computing approaches for key problems. A significant portion of Big Data is semi-structured data and growing rapidly. One important problem in the context of Big data is comparing data objects which exist in semi-structured format. Specifically, comparing tree objects which is a core component in several applications. Comparing ordered (unordered) trees is both compute- and data-intensive.

In this chapter, we reviewed the methodologies and techniques for efficient processing of semi-structured workload on parallel hardware.

CHAPTER 3. ELASTIC PUBLIC-SUBSCRIBE IN CLOUD USING DATA SHAPING

Summary. Advances in Internet of Things (IoT) and Big Data phenomenon are pressing for effective information dissemination mechanisms, for example, publish-subscribe systems to facilitate exchange of information between interested entities. A *data filtering algorithm* is at the core of a publish-subscribe system. The *data filtering algorithm* determines the matching of a document (message) with queries (profile). Cloud computing offers cost effective and easy access to vast computing resources which are oftentimes comprise heterogeneous computing platforms. Existing publish-subscribe solutions do not leverage heterogeneity prevalent in Cloud effectively, thus limiting their operational effectiveness. To be precise, the existing publish-subscribe solutions are not elastic in face of heterogeneous Cloud settings. In this chapter, we propose a elastic publish subscribe system. Specifically, our approach is as follows: (1) Develop a data shaping technique to reorganize semi-structured data in a manner amenable for processing on parallel architecture machines, and (2) Design a data shaping-based publish subscribe system for heterogeneous Cloud environment to facilitate elastic and portable computing. Our data shaping technique enables application of large number of cores/threads resulting in efficient utilization of compute resources of a highly parallel architecture machines. We experiment with real datasets on multicore and GPGPU architectures. Our experiments demonstrate that our data shaping-based approach delivers a scalable and high throughput solution.

Keywords: Cloud computing, accelerators, heterogeneous system, algorithms, publish-subscribe

3.1 Introduction

Internet of Things (IoT) [47] phenomena in concurrence with Big Data [84], [86] is revolutionizing the computing landscape. The Big Data comprises structured, semi-structured, and unstructured data. Semi-structured data is a prominent component of Big Data and is becoming more important day-by-day. The semi-structured data is hierarchical in nature, and occurs in formats such as JSON [58], Avro [12], XML [153], etc. The IoT in Big Data era thrives on fast and efficient exchange of information (messages). Fast and rapid data generation of information (messages) demand efficient ways for exchange and dissemination of those information (messages).

A publish-subscribe system facilitates asynchronous exchange of information, messages between various participating entities. Specifically, a publish subscribe system determines the matching of a document (message) with queries (profile) and forwards the matching documents (messages) to the user. A data filtering algorithm, which is the core of any given publish-subscribe system, determines the matching of a document (message) with queries (profile).

Cloud computing is a computing paradigm where hardware, system software, and compute resources are offered as a service [87], [14] . Typically, end users are charged for these services on pay-as-you-go basis. The Cloud model of computing benefits end users as it eliminates the upfront costs and operational costs of hosting the computing infrastructure needed. One of the key characteristics of Cloud computing model is elasticity. Elasticity is defined as the ability to expand the resource provisioning on need basis. Elasticity helps users and cloud service providers alike. In fact, elasticity is the key to exploit the low-cost benefits on cloud computing. Elasticity is preferred by end users as time to solution can be reduced drastically at no additional cost. For example, cost of using one core for thousand hours is same as cost to use thousand cores for one hour. Users prefer their applications to be executed as quickly as possible. For example, oftentimes, due to larger input data volumes, elasticity helps reduce execution time.

Literature comprises several publish-subcribe proposals [35], [7], [36], [44], [69], [96], etc. However, the existing research on publish-subscribe system lack on a key account – do not take advantage of elasticity in cloud – and are thus, not a good match for cloud computing. For example, consider the software-based approaches. The software-based approaches ([35], [7], [36], [44], [69]) rely on traditional Von Neumann machines and are inherently sequential in nature. Such systems typically yield filtering throughput in the range of few megabytes per second. These filtering systems were primarily designed to run on smaller 1/2/4 core chip multiprocessors. Some of these filtering systems use data encoding. Typically, either this data encoding stage is a bottleneck or a cause for bottleneck in subsequent stages. Another common limitation, shared by a number of filtering systems, is that they generate large intermediate data. Managing such data is challenging and affects the filtering throughput adversely.

3.1.1 Research Problem and Contribution

In this chapter, we study the following research problem: How can we leverage application portability to enable elastic publish-subscribe system in heterogeneous cloud environment?

We address the research problem in the following manner: Design a data shaping technique which enables application elasticity and portability in heterogeneous cloud environment.

Specifically, we make the following contributions:

- We develop a data shaping technique to reorganize semi-structured data in a manner amenable for processing on parallel architecture machines.
- We propose a data shaping-based publish subscribe system for heterogeneous Cloud environment to facilitate elastic and portable computing. Our data shaping technique enables application of large number of cores/threads resulting in efficient utilization of compute resources of a highly parallel architecture machines.

Rest of the chpater is organized as follows. Section 3.2 covers background. Section 3.3 describes our proposed data shaping technique. Section 3.4 describes a data shaping-based performance portable publish subscribe system stream. Section 3.5 presents the evaluation. The related work is covered in Section 3.6. Section 3.7 provides conclusion.

3.2 Background

Here we cover heterogeneity aspect in cloud environment and semi-structured data model, namely the tree data model.

3.2.1 Heterogeneity in Cloud Computing

Cloud computing environment exhibits heterogeneity in terms of types of processors leveraged for computing. Most prevalent processor types are traditional multicore CPUs and co-processors such as Xeon Phi's and General Purpose Graphics Processing Units (GPGPUs).

3.2.1.1 Multi-core CPUs and Co-processors

Modern multicore CPUs can have up to 40 cores organized as a two socket node. These multicore CPUs also comprise SIMD execution units for achieving higher performance and power efficiency.

Intel's Many Integrated Core (MIC) Architecture line of co-processors, also known as Xeon Phi's, has up to 60 cores. These core are equipped with 512-bit (or more) wide SIMD units. For example Intel KNL has a pair of 512 bit SIMD units on each core, with 64-72 cores, or a total of ∼130 SIMD units. Moreover, each core supports four hardware threads, or a total of ∼280 hardware threads. Intel Xeon Phi Knights Hill (KNH), future Xeon Phi co-processor, is likely to support even wider SIMD units with possibly larger number of cores.

OpenMP is a widely used runtime for multicore and Xeon Phi's. OpenMP has support for advanced vectorization and SIMD operations. The OpenMPv4.5 describes how these advanced features in MIC, and MIC-type architectures can be exploited by the end user. OpenACC, another runtime, also supports the SIMD and vector processing.

3.2.1.2 GPGPUs

GPGPUs are Single Instruction Multiple Thread (SIMT) machines comprising thousands of cores or streaming processors (SPs). GPGPUs are highly parallel architecture machines. For

details refer Section 2.3.1. In this chapter, we refer GPGPUs and Single Instruction Multiple Thread (SIMT) interchangeably.

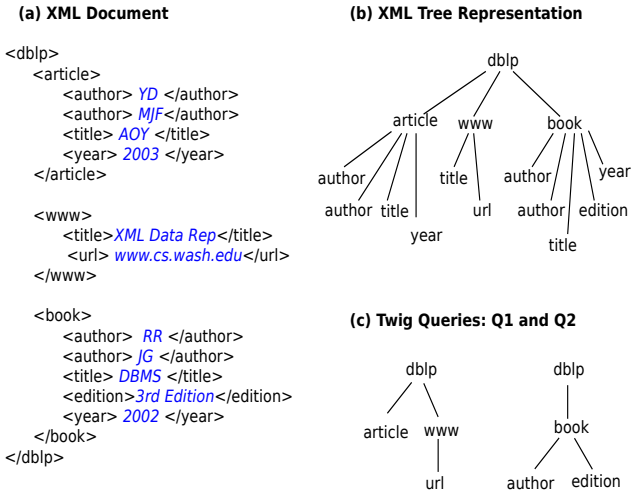


Figure 3.1: (a) XML document. (b) Tree representation of the XML document. (c) Queries Q1 (left) and Q2 (right).

3.2.2 Data Model

We consider a typical *publish-subscribe* data communication model for operating at the core of an Information Exchange.

The query set (also referred as user profiles) are obtained from subscribers *a-priori*, encoded and maintained as a set of data structures. We assume unordered twig queries. Order can be enforced using additional checks.

We assume that semi-structured input data is in the form of a tree. The nodes in input tree can repeat any number of times. The tree data set is represented in XML form [153]. The XML representation of input data set has a hierarchical structure comprising of markup (tags) and content (attributes). A tag begins with the character "<" and ends with a ">" . The tags can be of two types: "start-tag" and "end-tag". A node begins with a "start-tag" and ends with a corresponding "end-tag". For example, a node "*author*" begins with "< *author* >" and ends with "< /*author* >" in a XML document. There is a root node and a number of sub-nodes.

As an example consider a XML document and its tree representation shown in Figure 3.1 (a) and (b), respectively. Also, consider two queries Q1 and Q2 shown in Figure 3.1 (c).

3.3 Data Shaping for Elastic Computing

We first discuss requirements for an elastic application, motivation for data shaping. Next we discuss our proposed data shaping technique for portable publish-subscribe in Cloud environment.

3.3.1 Motivation

3.3.1.1 Scalability

Barazzutti et al. [17] observed that a given application should be intrinsically scalable in order to support elasticity in Cloud environment. Specifically, the addition (removal) of resources must result in an increase (decrease) of the processing capacity of the service or application. Besides scalability, that study points two other requirements – support for dynamic allocation of resources and decision making – for elastic computing.

GPU comprise a several thousand Cuda cores. For example, K20 GPUs have 2496 Cuda cores which are organized as 13 SMs with 192 cores per SM. Due to architectural considerations, applications running on GPUs must exhibit a high degree of parallelism. Degree of parallelism required is related to occupancy of SMs. For example, in order to attain an occupancy of more than 50%, we need 12,480 threads. This means that the kernel running on GPU needs to have 12,480 threads organized into several blocks. Multi-GPU systems comprising eight GPUs are not uncommon [4]. For compute node with eight GPUs, the application kernel needs to have ∼100,000 threads. Pascal GP100, the latest GPUs from Nvidia, comprises 3840 Cuda cores [102], and threading requirement further increases.

Figure 3.2 depicts the state-of-art model of computation for the publish-subscribe filtering algorithms. In this model, degree of parallelism is governed by the maximum number of queries. For example, since there are two queries in query set, only compute cores are being utilized. Under this approach, we require ∼12500 queries in order to use all the compute resources of a

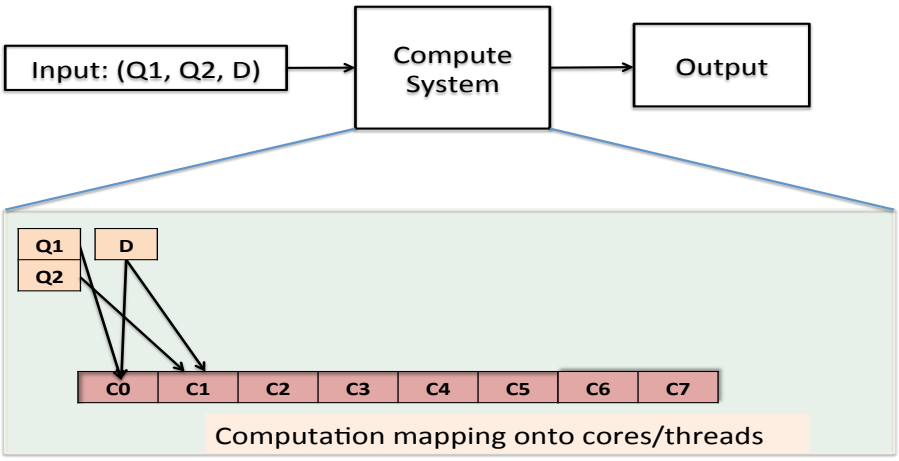


Figure 3.2: Baseline model of computation. Note that only a small subset of the available compute resources are utilized.

single K20 GPU. This requirement increases for later generation GPUs and with higher GPU count per node [102]. This aspect of the state-of-art computation model(s) is pathological to efficient utilization of GPUs and other parallel architecture co-processors/accelerators, hindering high performance and high throughput systems, including publish-subscribe systems.

Enabling Scalability via Data Shaping. Input data can be encoded in a way that it preserves the key structural relationship between the nodes in tree. Key information can be recorded as metadata and used to understand the relationship between the nodes in the tree. The metadata recorded should also be helpful in enabling smaller, independent processing. For example, the input data can be pre-processed to leverage the compute resources of a parallel hardware. Similarly, we consider encoding a tree structured input data using an appropriate encoding technique and enable parallel processing of tree dataset. Focus of data encoding should be to enable a fine grain mapping of these independent computations onto the thread/core available in computing platforms. Fine-grained independent computations are likely to result in a better performance across majority of computing platforms.

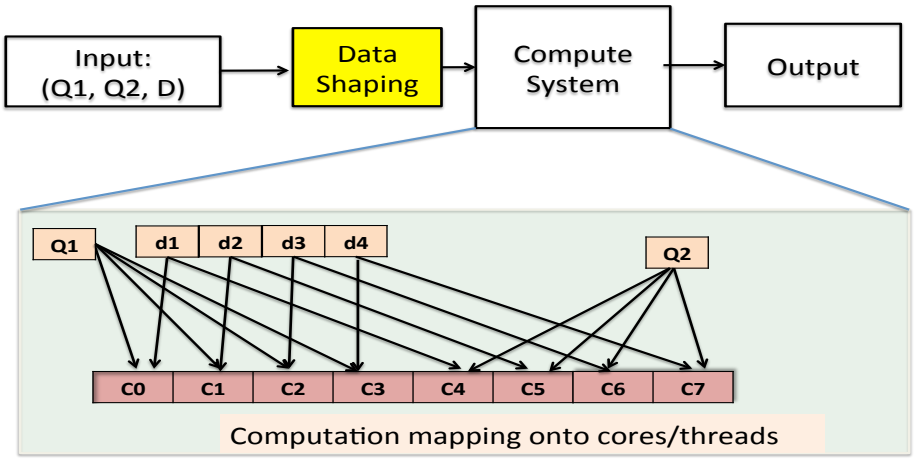


Figure 3.3: Computation model with data shaping. Data shaping ensures utilization of available compute resources.

Consider a computation model depicted in Figure 3.3. In this computation model, we propose a data shaping step which performs are careful reorganization of tree-model data increasing the overall parallelism of the application. For example, by re-organizing input document D into four smaller documents *d1, d2, d3, d4,* and carrying out filtering process in parallel on eight compute cores, the degree of parallelism increases by four-fold. Also, this is likely to result in up to 3X-4X reduction in execution time of the kernel, if algorithm (realized by kernel) is designed cleverly.

3.3.1.2 Application Portability

We argue that application potability is another mechanism which enables elasticity in Cloud. We define application portability or performance portable application as follows: application should be portable across a set of platforms and the addition or removal of compute resources (such as cores/threads) on those platforms should result in an increase or decrease of performance. Performance portability is possible as long as target platform meets basic performance critical criteria for the application. One such key criteria is – selectivity.

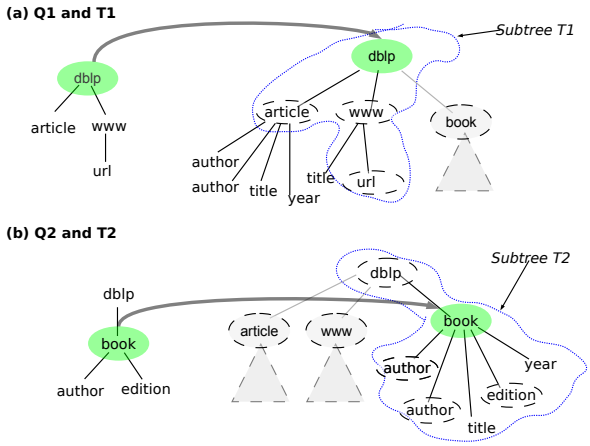


Figure 3.4: Parts of the Tree T relevant to queries Q1 and Q2. Region marked as Subtree T1 (Subtree T2) corresponds to query Q1 (query Q2).

Enabling Selectivity via Data Shaping. Data shaping enables selectivity by considering only relevant parts of the XML document for processing. Relevant parts of the tree can be identified and accessed quickly by maintaining index of the nodes as metadata. For example, only part of the Tree T marked as "Subtree T1" (refer Figure 3.4) is a candidate to match query Q1. Hence, only Subtree T1 should be matched instead of naively matching the query Q1 against complete Tree T (shown in Figure 3.1 (b)). Similarly, for query Q2, only "Subtree T2" needs to be considered for matching.

Precisely, data shaping results in: (1) increase spatial locality, (2) reduce the working set size and overall memory footprint leading to higher degree of concurrency in memory scarce platforms (for example GPGPU), (3) enable selective execution of only relevant parts of the input data

3.3.2 Data Shaping: The Mechanics

We first describe the encoding (or pre-processing) of the tree structured data using *Encode-TreePartial* algorithm. Then, we demonstrate its operation using examples.

The data shaping approach is as follows: (1) Encode the nodes in tree structured input data using an appropriate encoding scheme to facilitate sequential memory access. Maintain a address of encoded nodes in a data structure. (2) Use the address data structure to orches-

trate selective filtering of the tree. (3) Enable a scalable filtering by making all the filtering computations independent of each other.

3.3.2.1 Encoding Tree modeled data

We consider several tree encoding approaches proposed in literature. Specifically, we consider Prufer sequence encoding, Dewey ID labeling, pre-order, post-order, in-order, and root-path encoding schemes. After an informal analysis, we find that root-path encoding is a reasonable choice. Reason being, root-path encoding preserves the "parent-child" and "ancestor-descendant" relationships. Root-path encoding also helps better utilization of memory resources available by increasing the compute to global memory access ratio. Since, whenever, a node is accessed from memory for comparison, it very likely that its neighboring nodes will also be considered for matching.

Maintaining a record of the indices of nodes occurring in the NodeList helps to access and retrieve relevant parts of the tree directly. Also, the summary of the TagIdxMat reveals the frequency of the nodes occurring in the tree. The frequency of occurrence of a specific node in NodeList indicates the computation complexity involved in filtering queries having that node as the pivot node. And indexing of nodes also facilitates speculative filtering.

Finally, root-path encoding helps to address the problem of unordered twig matching, which is a general form.

3.3.2.2 The EncodeTreePartial Algorithm

Consider a tree T with following specifications. Total number of nodes in T is n. Number of leaf nodes in T is l. The average height of T is h.

We use rootpath encoding to encode only leaf nodes in tree T. The Root-path encoding is a type of prefix encoding where a node is represented by a set of nodes on path from the node to the root node of the tree. We refer to nodes in this set as rootpath nodes. We use following data structures to encode a tree: (1) NodeList, (2) TagIdxMat, (3) NodeListCtr, and (4) a set of tag counters TagIdxCtr. NodeList is a one-dimensional array of integers indexed by a value

T: Tree
NodeList: Encoded Tree
NodeListCounter: Number of nodes in NodeList
TagIndex Matrix: 2D array to record indices of nodes
P: Rootpath Stack
N: Node currently being visited
TagID(N): TagID of node N
Counter[tagid]: Number of nodes stored in NodeList
tagid = TagID(N)

```
1: Initialize NodeListCounter, Ctr[.] data structures
2: Traverse every node N of T in pre-order
   (Note:order of tree traversal does not matter)
3: for every N do
4:    if N is a LEAF NODE then
5:       Obtain rootpath encoding of node N in stack P.
6:       for every remaining node in stack P do
7:          K ← POP(P)
8:          tagid ← TagID(K)
9:          NodeList[NodeListCtr] ← tagid
10:         TagIdxMat[tagid][Ctr[tagid]]← NodeListCtr
11:         NodeListCtr++
12:         (Ctr[tagid])++
13:      end for
14:   else
15:      //Do nothing.
16:   end if
17:   Mark N as a visited node
18: end for
```

Algorithm 1 EncodeTreePartial (T)

assigned to it in pre-order traversal of the input tree. TagIdxMat is two-dimensional array of integers.

Each row in TagIdxMat is indexed by TagID. Entries in each row indicate the index of node in NodeList array indicating when this node is visited in pre-order traversal.

We denote tree to be encoded as T and resulting encoded tree as NodeList. We provision a stack (P) to record rootpath nodes and initializes NodeListCtr to zero. Our tree encoding algorithm traverses all nodes in a tree in pre-order (order of traversal is not important though). It marks the node being visited as N. If N is a leaf node then it obtains the root-path of N in stack P, and concatenates the nodes in P to the end of encoded tree NodeList. Concatenation process works as follows: (1) POP one node at a time from stack P; (2) store this node at index given by current value of NodeListCtr; (3) update the array of indices corresponding to TagID of the current node; and then (4) increment NodeListCtr by one. The tree encoding algorithm is described as *EncodeTreePartial* (Algorithm 1).

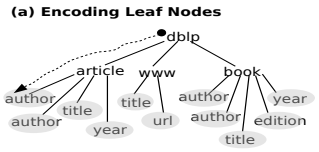
Note that the order of tree traversal in *EncodeTreePartial* algorithm does not matter. The reasoning is the rootpath encoding of a node depends only on the set of nodes occurring en-route from itself to root. In EncodeTreePartial algorithm, the order of tree traversal determines the order in which the leaf nodes are encoded and not the rootpath encoding of these leaf nodes. Traverse every node N of T in pre-order. Also note that decoding of data is not needed for filtering over tree modeled data sets.

3.3.2.3 Complexity of EncodeTreePartial Algorithm

The space complexity of EncodeTreePartial algorithm depends on number of leaf nodes and average height (h). Note that tree T has l leaf nodes and an average height of h. Space requirement is of the order of 'h' times $O(n)$ based on the following discussion. We consider three cases of a T: (1) a unary tree, (2) a balanced m-ary tree, and (3) a shallow bushy tree.

Case 1: A unary tree. In this case, there is a single leaf node. And, rootpath encoding of this leaf node comprises all the nodes of unary tree. The space requirement is $O(n)$.

Case 2: A balanced m-ary tree. Here, $h = log_m n$, and number of leaf nodes is less than n. The space complexity is $(log_m n)*O(n)$.



(b) TagIdxMat

TagID	Index (in NodeList)
0 (dblp)	0,3,6,9,12,15,18,21,24,27,30
1 (article)	1,4,7,10
2 (author)	2,5,20,23
3 (title)	8,14,26
4 (year)	11,32
5 (www)	13,16
6 (url)	17
7 (book)	19,22,25,28,31
8 (edition)	29

(c) NodeList

Index	Node
0	dblp
1	article
2	author
3	dblp
4	article
5	author
6	dblp
7	article
8	title
9	dblp
10	article
11	year
12	dblp
13	www
14	title
15	dblp
16	www
17	url
18	dblp
19	book
20	author
21	dblp
22	book
23	author
24	dblp
25	book
26	title
27	dblp
28	book
29	edition
30	dblp
31	book
32	year

Figure 3.5: Encoding and indexing of nodes in tree T. (a) Root-path of leaf node "author" (shown by dotted line). (b) TagIdxMat. (c) Nodelist.

Case 3: A shallow bushy tree with h = k, a constant.

Here, number of leaf nodes is of the order of $O(n)$, and space complexity is k times $O(n)$ or k*$O(n)$.

3.3.2.4 Input Encoding Example 1: Document Encoding

The operation of the input encoding process is explained using a tree document. We demonstrate encoding of a document using *EncodeTreePartial* algorithm (see Algorithm 1) and update TagIdxMat and NodeList data structures. For document tree shown in Figure 3.1 (b), nodes "author", "author", "title", "year"; "title", "url"; "author", "author", "title", "edition", and "year" are encoded. The root-path of leaf node "author" i.e. path from node "author" (whose parent is "article") to root of the tree "dblp" is (dblp, article, author) as shown by dotted line in Figure 3.5(a).

After encoding, the document tree is represented as TagIdxMat and NodeList data structures. The TagIdxMat and NodeList are shown in Figure 3.5(b) and (c), respectively.

Nodes are represented by their TagIDs when stored in memory. TagIDs are as mentioned in the Column 1 of TagIdxMat (refer Figure 3.5(a)). We use the following data structures for encoding queries: (1) QueryInfo, and (2) QueryArray. QueryInfo is a 2-dimensional array which maintains meta data of queries. QueryArray, a 1-D array, comprises query data. Queries (or profiles) provided by users are encoded using *EncodeTreePartial* algorithm.

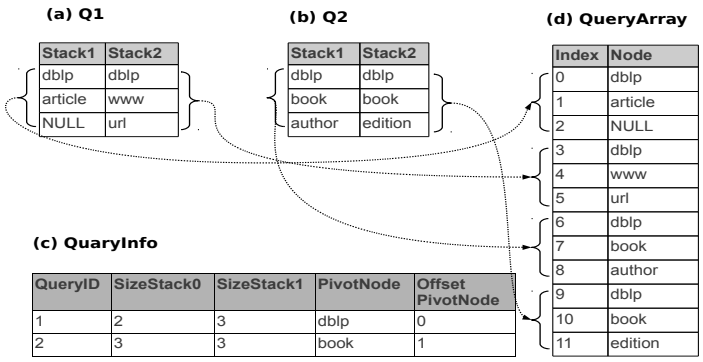


Figure 3.6: Query Encoding. (a) Query Q1 as stacks. (b) Query Q2 as stacks. (c) QueryInfo: Metadata of query set. (d) QuaryArray data structure. Note: nodes in Q1 and Q2 need not be unique.

3.3.2.5 Input Encoding Example 2: Query Encoding

We describe query encoding process using queries Q1 and Q2. Figure 3.6(a) shows Q1 comprising two leaf nodes (article and url). The leaf nodes article and url are encoded as (dblp, article) and (dblp, www, url), respectively. Each leaf node is assigned a stack. Stack is populated with encoding of the corresponding leaf node. Leaf node "article" is assigned Stack1 and node "url" is assigned Stack2. All queries are made fixed length equal to StackSize (here we assume StackSize to be three), by padding with NULL entries. Note that keeping all the stacks (of a query) of the same length is key to allow GP-GPUs to perform efficiently. Finally, metadata of Q1 such as size of stacks (SizeStack0 and SizeStack1), pivot node (PivotNode), and offset to pivot node (OffsetPivotNode) are stored in QueryInfo. Stack1 and Stack2 are copied to "QueryArray". For query Q1, Stack1 after padding becomes (dblp, article, NULL). Stack2 remains unchanged. Similarly, encoding of query Q2 in Figure 3.6(b) results in two stacks namely Stack1: (dblp, book, author), Stack2: (dblp, book, edition). Metadata of Q2 is also stored in "QueryInfo", and Stack1 and Stack2 are copied to "QueryArray".

Listing 3.1: Portable Twig Filter Algorithm

```
1 FilterPortableGPU(...)
2{ if (blkIdx.z < Q)
3  {  MAX_GRIDS_X  = 1;
4     if (blkIdx.y < MAX_GRIDS_X)
5    {  tag_idx = query_info[blkIdx.z].pivot_node;
6       if (query_info[blkIdx.z].pivot_node == '*')
7      {  if (query_info[blkIdx.z].offset_pnode == 0)
8         tag_idx = 0;
9         if (blockIdx.y > 0)   {tag_idx = blkIdx.y;}
10        N1 = tag_id_ctr[tag_idx]; K1 = N1/B;
11        //default value; //find K1
12        if ( N1  (MOD) B)     { K1++;}
13     }
14     if ( blkIdx.x < K1*S)
15     {  if (threadIdx.x < B)
16       {  temp=(blkIdx.x)*(B)/S + (threadIdx.x)/S;
17         //obtain start address in rp_nodelist
18         start_addr=tag_index_matrix[tag\_idx].val[temp];
19         if ( start_addr > (query_info[blkIdx.z].offset_pnode) )
20         start_addr=start_addr - query_info[blkIdx.z].offset_pnode;
21       }
22     stackID = (threadIdx.x) MOD (2);
23     j = (blkIdx.z)*(2*8) + ( stackID*8 )
24     k = start_addr;    matching_axis = 0;
25     ret_k_idx = 0;     currMatch = 0;
26     for ( i=0; i < query_info[blkIdx.z].max_sz; i++)
27     {  if (query_array[j] ==  '*')
28        currMatch++;
29        if (query_array[j].tag_id == rp_nodelist[k].tag_id)
30        currMatch++;
31        if (matching_axis)
32        {  matching_axis = 0; increment_axis = 0;}
33       if (matching_axis) {  j--;  }
34       if (query_array[j].tag_id  ==  '//')
35       {  currMatch++;  matching_axis =1;  increment_axis =1;  }
36       j++; k++;
37     }
38   if (query_info[blkIdx.z].stack_size[stackID] > 0)
39   if(currMatch>=query_info[blkIdx.z].stack_sz[stackID])
40     TwigMatch[blkIdx.z].stack[stackID] = 1;
41 }}}
42}
```

3.4 Performance Portable Publish-Subscribe System

We first provide a brief overview of the proposed performance portable publish-subscribe system and then describe the filtering process.

3.4.1 Overview of System Architecture

The filtering system comprises a limited size buffer to store streaming data, a mechanism to encode and store query set, a filtering engine to process query set over input data, and a result reporting module. The query set is encoded and maintained as a set of data structures.

The framework uses information about size of available system memory and 'data stream meta-data' and determines the chunk size. The XML "data stream" is runtime input to filtering framework. The input data stream is stored in a data Buffer. Data is read in chunks of a predefined size, line-by-line from the data buffer. For parsing XML data streams we use Expat [1], an open source XML parser. The input data chunk is encoded and transferred to the GPU memory, and filtering engine is invoked. The filtering engine filters the query set over encoded data chunk and reports back the queries (referred as matching queries) which appears at least once in the data chunk. The corresponding data chunk can be forwarded to users for matched queries. This process of reading data chunk from buffer, filtering queries over data, and reporting back matching queries is repeated continuously.

3.4.2 Filtering Process

The twig (or path) query filtering operation is carried out by an algorithm depicted implemented as Listing 3.1. The implementation in Listing 3.1 corresponds to single instruction multiple thread (SIMT) machine. In SIMT machines (such as GP-GPU), computation is carried out by several threads. A number of threads are arranged as blocks and blocks are further organized as grids. Important parameters are as following: Q: Maximum number of queries; B: Block Size; N: Maximum number of nodes in document; K = $\lceil N/B \rceil$; S: Number of stack in a query; StackSize: Size of Stack; NumBlocks (K*S, NumMaxTags, Q): A 3-d organization of blocks; NumThreads(B, 1): A block of B threads.

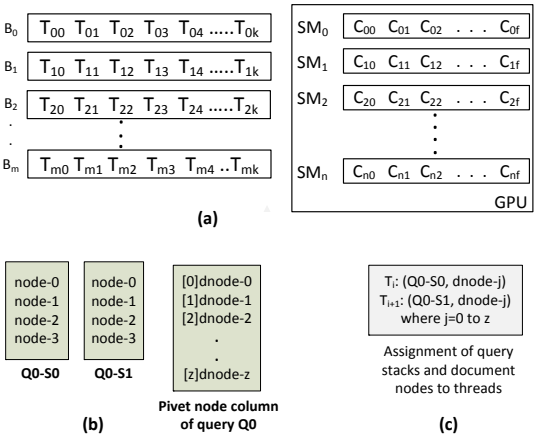


Figure 3.7: (a) (Left) Group of threads as blocks. (a) (Right) Set of cores as SMs. (b) (Left) Query Q0 encoded as stacks, Q0-S0 and Q0-S1. (b) (Right) Pivot node column of query Q0. (c) Assignment of query stacks to GPU threads.

Twig filtering computation is organized as a set of grids. Each grid comprises several thread blocks. One query is mapped to one grid. There are a total of Q grids. Within each grid, there are a number of blocks. Blocks are organized three dimensionally and represented as NumBlocks (K*S, NumMaxTags, Q). Third dimension in NumBlocks() represents grid id. A block is defined as NumThreads(B, 1) where B is number of threads in a block. Each thread is assigned a computation kernel comprising of algorithms Listing 3.1.

Parameters Q, S, and StackSize depend on the query set. The NumMaxTags parameter depends on the input document stream. These parameters are available beforehand (courtesy query set and data stream meta data information). Parameters B and N are system specific. Rest of the parameters such as NumBlocks and K are derived at runtime.

3.4.2.1 Thread Level Execution

Each thread handles single stack of twig query. A thread fetches its stack (a part of twig) and compares it with a part of the document tree. Note that a query set is encoded and organized into QueryInfo and QueryArray data structures. Each thread obtains its stack data in following way. First, it obtains meta data of the query stored in QueryInfo data structure. Thread accesses QueryInfo data structure using its QueryId (grid id) given by blockIdx.z. Note

ThreadID	Q (QueryID)	S (num stacks)	StackSize	StackID	Starting address in QueryArray=Q*(S*StackSize) + (StackID*StackSize)
0	0	2	5	0	(0*(2*5)+(0*5))=0
1	0	2	5	1	(0*(2*5)+(1*5))=5
2	0	2	5	0	(0*(2*5)+(0*5))=0
3	0	2	5	1	(0*(2*5)+(1*5))=5

(a)

PivetNode of Q0	a6
TagID of node a6:	6

blockIdx.x	B	S (num stacks)	threadIdx.x	Temp= (((blockIdx.x)*B)/S) +(threadIdx.x/S)	Starting address in NodeList= TagIndex[6] [temp]
0	8	2	0	(((0)*(8)/2)+(0/2))=0	TagIndex[6][0]=1
0	8	2	1	(((0)*(8)/2)+(1/2))=0	TagIndex[6][0]=1
0	8	2	2	(((0)*(8)/2)+(2/2))=1	TagIndex[6][1]=4
0	8	2	3	(((0)*(8)/2)+(3/2))=1	TagIndex[6][1]=4

(b)

Figure 3.8: The thread level execution of query Q0. Calculation of starting address by thread in :(a) QueryArray and (b) in NodeList.

that meta data of a query also contains the count of stacks in twig query. Each thread computes the address of its stack in QueryArray as: Q*(S*StackSize) + (StackID*StackSize).

Each thread operates only on a small data chunk of the document tree. The document tree is encoded into NodeList and TagIndex matrix. Each thread needs to obtain the corresponding data chunk from the NodeList array. To this end, thread combines its thread signature and query meta information as: (((blockIdx.x)*B)/S) + (threadIdx.x/S), and performs a look up in TagIndex matrix. TagIndex matrix is indexed by TagIDs. A correction, if needed using value of offset, is done. Lines 17-21 in Listing 3.1 show how a thread obtains starting address in NodeList. Lines 27-37 in Listing 3.1 carries out stack matching. Total number of matches between data segment in NodeList and query stack is recorded in currMatch variable. If value of currMatch equals actual size of stack (obtained from query meta data), then match flag corresponding to query's stack is set. Finally, for every query, the filtering system checks all the matching flags. If all the matching flags are set, then filtering system declares that query as a matching query.

3.4.2.2 Filtering Operation

Consider document tree T (Figure 3.5(a)) and a query set comprising two queries Q0, Q1 shown in Figure 3.6(a) and (b). Also, the encoded form of tree T is shown as NodeList and

TagIndex in Figure 3.5(a) and (b), respectively. Encoding of Q0, Q1 results in QueryInfo and QueryArray data structures (refer to Figures 3.6(c)-(d)). Numerical value of the relevant parameters are: B = 8, Q = 2, N = 34, K = $\lceil 34/8 \rceil$ = 5, S = 2, StackSize = 5. NumMaxTags = 11. Next, we describe the filtering computation.

The filtering computation is divided into two grids as follows: NumBlocks(5*2, 11, 2), NumThreads(5, 1). Grid 0 manages Q0 and grid 1 manages Q1. Each thread of grid 0 access 0th row of QueryInfo, obtains query meta data, and computes starting address of stacks in QueryArray.

For thread 0: threadID = 0, Q = 0, S = 2, StackSize = 5, StackID = threadID (MOD) 2; starting address stack is 0*(2*5)+(0*5) = 0. For thread 1, the starting address is = 5.

Figure 3.8 shows computation of starting address in NodeList. For Q0 PivotNode is a6. For thread 0: blockIdx.x = 0, temp = (0*8/2)+(0/2) = 0. Starting address start_addr = TagIndex[6][0] = 1 and offset is 0. Hence, starting address for thread 0 in NodeList is 1. Likewise, for thread 1, starting address = TagIndex[6][0] = 1 (note that the offset is 0). Thread 0 and 1 matches part of document tree starting at index 1 (in NodeList) with stack0 and stack1. Starting address for other threads is computed accordingly.

Likewise, for Q1, starting address for threads 0, 1 are 1*(2*5)+(0*5) = 10 and 1*(2*5)+(1*5) = 15, respectively, in QueryArray. PivotNode for query Q1 is a2. Note that offset is 1 for Q1. Each thread accesses second row of TagIndex matrix and obtains their starting address in NodeList. For thread 0, computation is: TagIndex[2][(0*8/2)+(0/2)] = 12. And for thread 1: TagIndex[2][(0*8/2)+(1/2)] = 12. After subtracting offset, it becomes 11.

3.4.2.3 Limitations

The Algorithm realized in Listing 3.1 yields some false positives and false negatives. The *FilterPortableGPU* algorithm works by matching individual paths of a twig query in parallel, resulting in some false positives. False negative results are platform dependent. The number of false negatives depend on memory of computing platform. A large tree is encoded (and processed) in segments and size of a segment depends on memory available for computation. In other words, a twig query is filtered over a segment of the tree instead of complete tree.

While the number of false positives should be minimal, but since the extremely fast filtering systems is/are deployed in first phase of data processing, our goal is to identify "data-of-interest" and reduce the size of data to be processed by the subsequent data processing layers. Therefore, we can afford to have a higher number of false positives.

3.5 Evaluation

3.5.1 Experimental Setup

3.5.1.1 Platform

We experiment with CPU and GPU. CPU node comprises two 2.0 GHz 8-Core Intel E5 2650 providing 16 cores in total. Each CPU node has 128 GB memory. We used Kepler K20 as GPU. K20 GPU comprises 13 Symmetric Multiprocessors (SMs) and 6 GB of memory. Each SM has 192 Cuda cores operating at 706 MHz.

3.5.1.2 Dataset

We experiment with 'DBLP' and 'psd7003' data sets. The DBLP and 'psd7003' data sets are obtained from the XML repository maintained at University of Washington. Table 3.2 lists the specifics of the data sets.

We used YFilter query generator [36] for generating twig queries. We generate distinct twig queries using parameters listed in Table 3.3. We measure time elapsed in execution using timeoftheday() utility of Unix.

Table 3.1: Experimental platform.

Compute Node	Core count	Description
SB-16/32	16/32	Sandybridge CMP
K20	2496 (as 13*192)	Kepler-20 GPGPU

Table 3.2: The input parameters (Document Description)

Document	DBLP	psd7003
Size	2.5MB - 64MB	
Avg. depth	2.9	5.15
Max. depth	6	7
Num. unique tags	35	65

Table 3.3: The input parameters (Query Set Description)

Queries:	
Set size	4-32, 512, 1K, 2K, 4K
Maximum depth	8
Prob. of '*'	0.1
Prob. of '//'	0.1
Num. of nested paths	1

3.5.2 Memory Overhead of Data Encoding

We define data compression factor (or compression) as a ratio of size of encoded document and size of raw XML document. In general, the *EncodeTreePartial* algorithm yields a compression of 0.632. The *EncodeTreePartial* algorithm encodes a 128 MB DBLP XML document into NodeList and TagIndexMatrix data structures of size 81 MB. We would like to add that the primarily goal of this work is to enable a high throughput data filtering, and not to develop an efficient encoding technique for compressing tree structured data sets.

3.5.3 Impact of Data Shaping on Work Generation

First, we discuss effect of data shaping-based on work creation. Figure 3.9 plots the time elapsed in filtering smaller query sets. X-axis represents the query set sizes from 4 - 32. Y-axis plots the time elapsed in execution of filtering computations in logarithmic scale for block sizes 32 - 1024. Smaller workload (query set size and document size which is 2.5 MB) have been used in this set of experiment. Idea behind using smaller work and specifically, smaller query sets is to measure the ability of our data shaping approach to distribute work among processing threads (Cuda threads) of SMs.

From Figure 3.9 we observe that even for smaller query sets, for example 4-32 queries, by

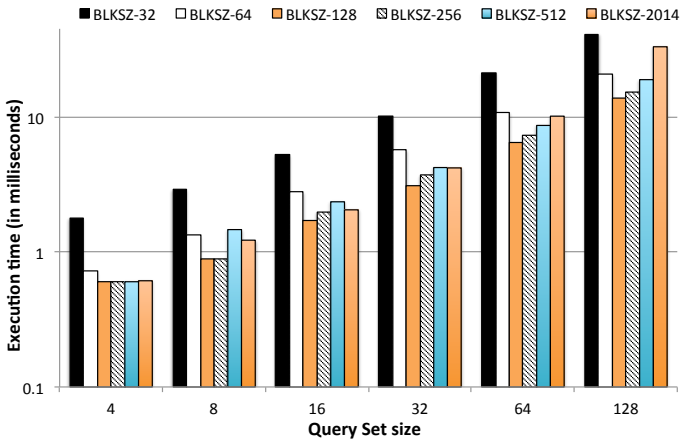


Figure 3.9: Impact of Data Shaping technique on work generation for query sets of size 4-32 and 2.5 MB psd7003 document.

data shaping approach, we can generate more than 330,000 kernels when filtering over 2.5 MB dataset. Note that maximum number of kernels no longer depends upon number of queries in the query set. Had that been the case, maximum number of kernels are limited to 4 for a query set comprising four queries. Table 3.4 lists more total number of kernels generated for various configurations of block sizes (B) and query set sizes (Q) for a 2.5 MB psd7003 document.

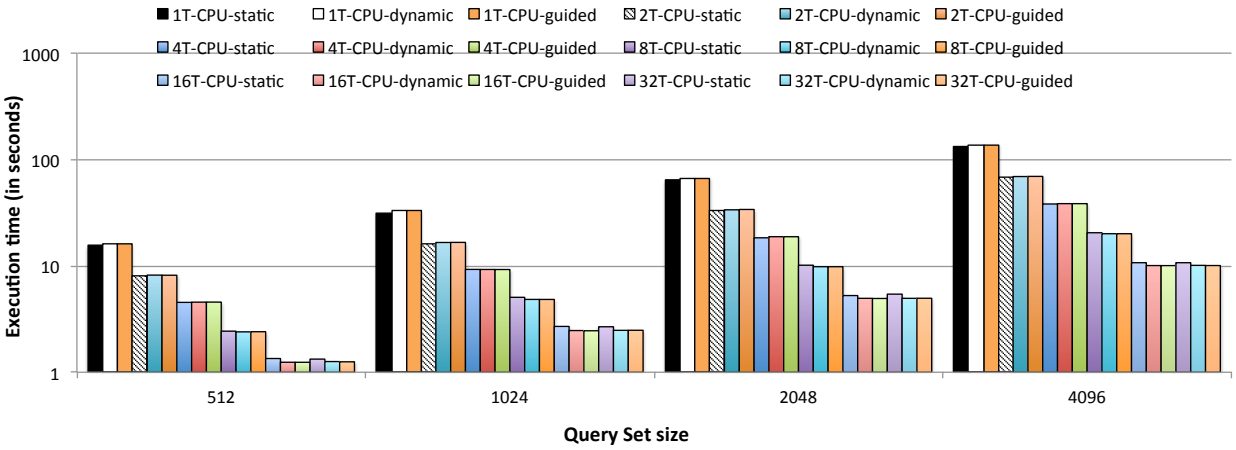


Figure 3.10: Effect of various OMP thread scheduling strategies (static, dynamic, guided) on performance for 64 MB DBLP dataset.

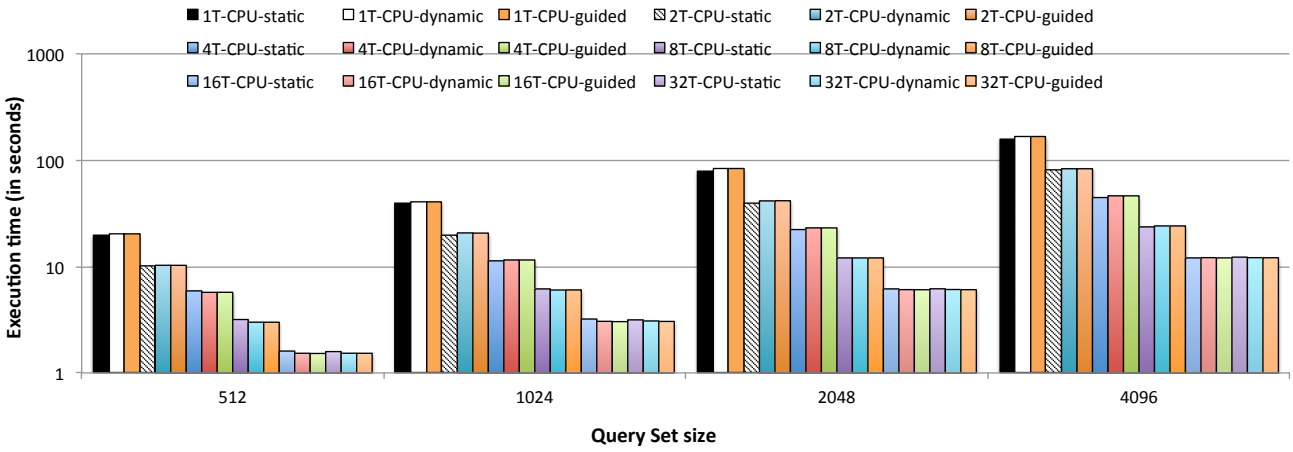


Figure 3.11: Effect of various OMP thread scheduling strategies (static, dynamic, guided) on performance for 64 MB DBLP dataset.

3.5.4 Performance on CPU and GPU

Here, we discuss performance of our data shaping-based filtering solution on CPU and GPU.

3.5.4.1 Effect of Thread Scheduling Schemes

Figure 3.10 depicts the effect of OMP thread scheduling strategies on performance for 64 MB DBLP dataset. We experiment with three OMP runtime scheduling schemes: static, dynamic, and guided. X-axis represents the query set sizes from 512 - 4K. Y-axis plots the time elapsed in execution of filtering computations in logarithmic scale.

From Figure 3.10, we observe that static scheduling performs better that dynamic and guided scheduling schemes when number of OMP threads are 1/2/4. However, guided scheduling scheme outperforms static and dynamic schemes when 8/16/32 OMP threads are used. For these threads, guided scheduling scheme reports performance gain in the range of 2% to 9%, on an average. For example, for 2K query set size, guided scheduling outperforms static scheduling scheme by 9% when number of OMP thread are set to 32.

Figure 3.11 depicts the effect of OMP thread scheduling strategies on performance for 64 MB psd7003 dataset. We experiment with three OMP runtime scheduling schemes: static, dynamic, and guided. X-axis represents the query set sizes from 512 - 4K and Y-axis plots the time elapsed in execution of filtering computations in logarithmic scale. From the figure, we observe that scheduling schemes perform similar to DBLP dataset. Static scheduling scheme outperforms dynamic and guided schemes when number of OMP threads are less than 8. And when number OMP threads are increased to 8 or more, dynamic scheme performs better than its counterparts.

3.5.4.2 Effect of Thread Blocks on GPU Performance

Figure 3.12 depicts the effect of GPU block sizes on filtering performance. X-axis represents the query set sizes from 512 - 4K. Y-axis plots the time elapsed in execution of filtering computations for block sizes 128 - 1024.

From Figure 3.12, we observe that smaller block sizes yield slightly better performance. For

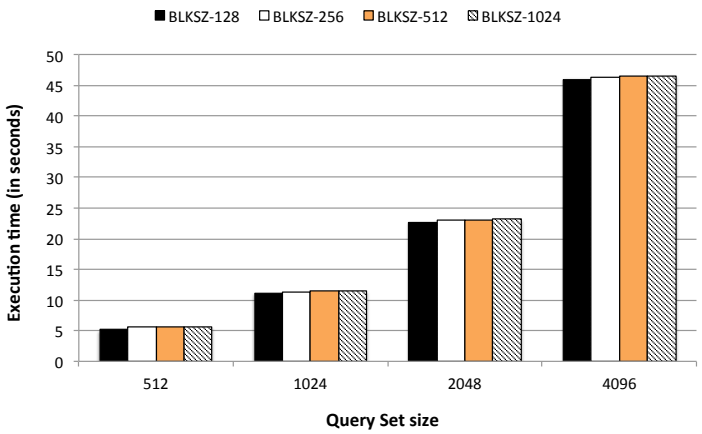


Figure 3.12: Effect of GPU Block size on performance of psd7003 dataset.

example, for query set of 512, increasing block size from 128 to 1024 degrades the performance by ∼10%. We observe similar trend for other query sets. However, performance loss is in the range of 1% - 3%. When block size is kept constant and query set size is increased from 512 to 4K, the execution time increases by 8.9X. However, for larger block sizes, for a eight-fold increase in query set size, corresponding increase in execution time is nearly 8.3X - 8.2X.

3.5.5 Scalability

3.5.5.1 DBLP dataset

Figure 3.13 plots the throughput performance of filtering over 64 MB DBLP data set on CPU and GPU. X-axis represents the query set sizes from 512 - 4K. Y-axis plots the time elapsed in execution of filtering computations. We experiment with various configurations on CPU: Single OMP thread on CPU, denoted as 1T-CPU to 32 OMP threads on CPU, denoted as 32T-CPU. For GPU, we experiment with all the 13 SMs on K-20 GPU, denoted at 13SM-GPU.

From Figure 3.13, we observe that when using single CPU core (1T-CPU) to filter a query set of size 512, the time elapsed is 15.50 seconds. We also observe that using more threads (or more CPU cores) decreases the time elapsed in filtering. When with 8 cores, the execution time decreases by ∼6.3X and with 16 cores the execution time decreases by ∼11.5X. On an average, by doubling number of threads, the time elapsed decreases by ∼0.54X.

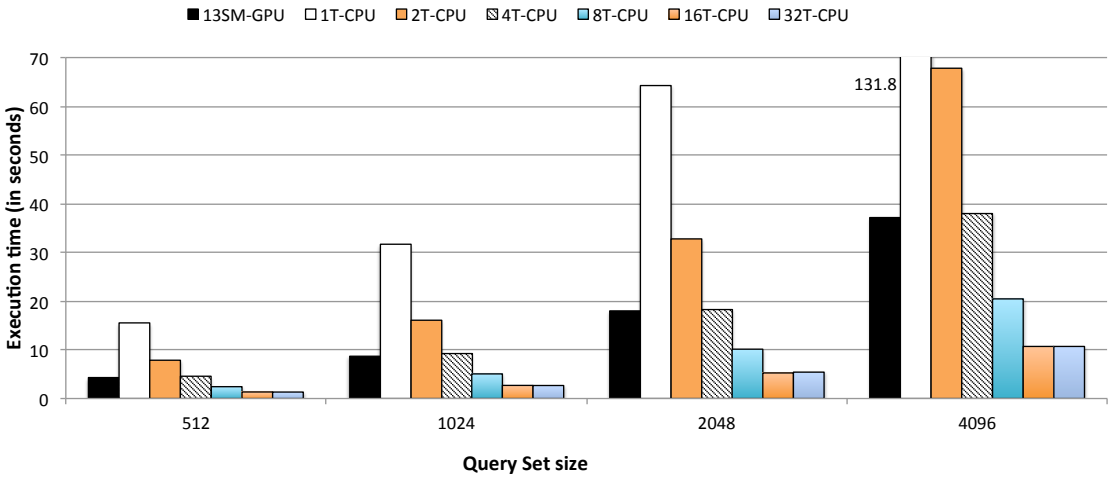


Figure 3.13: Performance of data shaping based filtering algorithm on GPU and multicore CPU for 64 MB DBLP dataset.

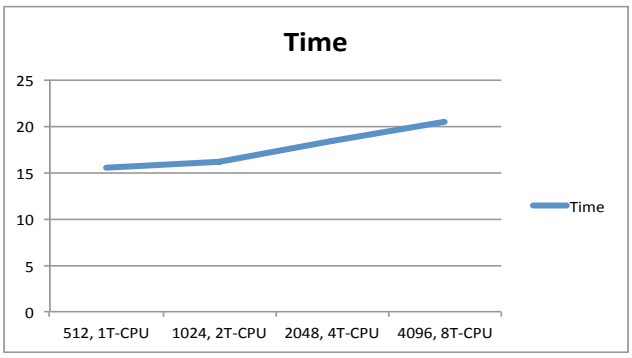


Figure 3.14: Weak scaling performance of data shaping based filtering algorithm on GPU and multicore CPU for 64 MB DBLP dataset.

Similarly, for query set of size 1K, when number of threads are varied from 1 to 16, the time elapsed in filtering decreases by almost linearly. For example, using 8 cores decreases execution time by ∼6.2X, with 16 cores the execution time decreases by ∼11.7X. For query sets 2K and 4K, we observe similar scalability trend.

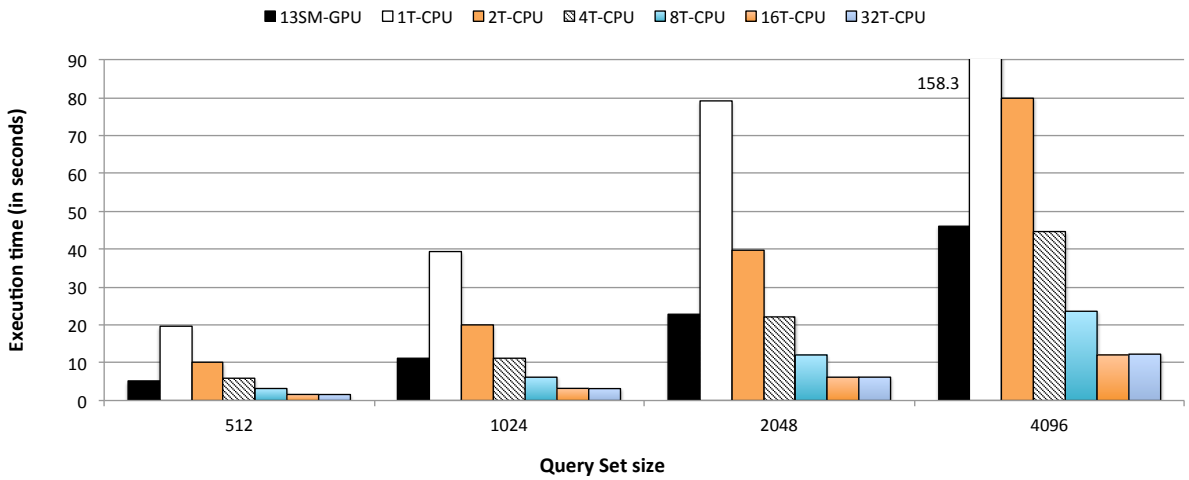


Figure 3.15: Performance of data shaping based filtering algorithm on GPU and multicore CPU for 64 MB psd7003 dataset.

Figure 3.14 summaries the weak scaling performance of our algorithm. From this figure we observe that on doubling the workload and number of cores simultaneously, the execution time decreases by ∼0.9X. For example, when workload in increased by from 512 queries to 4K queries, an 8X increase, and number of CPU cores are increased from 1 to 8, the time elapsed increases by a small factor of 1.3X. These experiments reveal that our solution meets weak-scaling criteria.

3.5.5.2 PSD7003 dataset

Figure 3.15 plots the throughput performance of filtering over 64 MB DBLP data set on CPU and GPU. X-axis represents the query set sizes from 512 - 4K. Y-axis plots the time elapsed in execution of filtering computations. We experiment with various configurations on CPU: Single OMP thread on CPU, denoted as 1T-CPU to 32 OMP threads on CPU, denoted as 32T-CPU. For GPU, we experiment with all the 13 SMs on K-20 GPU, denoted at 13SM-GPU.

From Figure 3.15, we observe that for various query sets when number of threads are varied from 1 to 16, the time elapsed in filtering decreases by almost linearly. Also, using more threads (or more CPU cores) decreases the time elapsed in filtering.

3.5.5.3 False Positives and Limitations

Like some other split-join algorithms for twig filtering, our filtering algorithm produces false positives. We observe 20-25% of false positives in filtering over 5 MB, 25 MB and 50 MB documents across 128, 1K and 4K query sets. Note that filtering using path/pair approaches used [96] result in approximately 25% false positives. False positives can be avoided by maintaining additional information. Also, our twig filtering algorithm does not filter queries having '*' or '//' as pivet node. We seek to address these issues in future.

3.6 Related Work

3.6.1 Publish-subscribe in cloud

Ye and Kim [78] proposed BlueDove which is a scalable and elastic publish subscribe service in cloud. Our study differs from [78] in two ways: (1) we consider a semi-structured data form messages and queries, whereas Ye and Kim consider structured data as messages. (2) We consider structure-based queries whereas they focus on attribute-based queries. and (3) most significant difference being that we work to design an elastic application, while they focus on using elasticity offered in cloud.

Barazzutti et al. [17] propose E-STREAMHUB, an elastic publish subscribe system. Barazzutti et al. [17] study dynamic scaling of state-ful and state-less publish subscribe operators. They also proposed elasticity policies, both local and global, in order to ensure high system utilization and stable operation latencies.

Belyaev and Ray [19] study on dissemination and filtering of XML data streams, and propose a subscriber-centric XML filtering approach for replication and distribution of XML streams. Their approach allows for selective filtering on more efficient nodes.

Tran et. al propose EQS, an elastic and scalable message queue for Cloud [137]. They also propose a message queue-based adaptable scaling algorithm.

Ma et. al propose SREM, a scalable and reliable matching service for content-based publish/subscribe systems in cloud settings [79]. A distributed overlay is proposed to achieve low routing latency and reliability. Authors also propose a hybrid space partitioning technique HPartition, which maps large-scale skewed subscriptions into multiple subspaces which results in high matching throughput.

3.6.2 Tree encoding and data filtering techniques

Several transformation techniques have been proposed in literature. A Node Encoded Tree Sequence (NETS) is proposed in [124]. NETS models an XML document as a rooted ordered labeled tree such that each node corresponds to an element tag, attribute or value. The structural relationship between these nodes is represented as edges. PRIX [119] and FiST [69] use Prufer sequence encoding for filtering twig. FiST first transforms XML document and queries into their Prufer sequence, and then carries out subsequence matching to find matches. FiST supports ordered query matching.

ViST [146] transforms XML tree pattern into structure encoded sequence and uses rootpath encoding where each node is expressed as a sequence of intermediate nodes from root to itself. The worst-case storage requirement can be higher than linear in the total number of elements of the XML documents. A key difference between our tree encoding approach and [146] is that we encode selective nodes and storage space requirement in our case is $O(n)$.

Dewey ID labeling to represent XML order in relational data model is proposed in [135]. In this work, authors also apply Dewey ID labeling scheme to preserve document order during XML query processing. ORDPATH is a variation of prefix labeling is proposed in [105]. ORDPATH deals with insertion of XML nodes in the database. The region encoding is introduced in [26], where focus is on searching and indexing the text database.

A mixed hardware/software approach for filtering simple XPath queries is proposed in [144]. Queries having only parent-child axis are handled by this approach. Authors [90] proposed a pure hardware approach for XML filtering, which cannot handle recursion in XML documents. Work in [96] uses FPGA for holistic twig filtering. This FPGA based approach provide a high throughput averaging more than 200 MB/s, but has a severe limitation in terms of the number of queries being filtered.

YFilter [35] is a FSM-based approach which uses Non-Deterministic Finite Automata (NFA) to represent the user profiles. YFilter operates by breaking twig into root-to-leaf path(s), and builds a unified NFA over set of paths. A lazy Deterministic Finite Automation(DFA) based filtering approach also has been proposed in [45]. A key difference between our data shaping solution and *state-of-the-art* dynamic programming based approach is as follows: We carry out selective processing of the nodes instead of comparing each and every node in the tree against all the nodes in the query set. By being selective we reduce the computational complexity incurred in dynamic programming based approach, which is $O(m \times n)$, to $O(m' \times n)$ where m' can be a fraction of m and it can as small as zero.

Another proposal [95] uses GPU for filtering the XML data streams. This proposal only filters path queries and does not address the twig filtering problem. The proposal uses dynamic programming approach in order to filter the path queries. The study in [95] records 1 MBps throughout for filtering 8K path queries over 50 MB DBLP document using 240 streaming processors (SPs). Research in [128], leverages GPU to accelerate the processing of a single query processing in XML databases.

Roy, Teubner, and Alonso [122] used hybrid systems for processing data sets encoded in XML. Our work differs from the works of [122] in several ways: (1) we do not make any assumption regarding the distribution of item sets in the input data. (2) Our focus is on processing larger documents in the range of hundreds of megabytes as opposed to filtering out the frequent items and reducing the workload of data mining algorithm. (3) We adopt a platform-centric approach and design a general filtering system which can be used to filter data streams whereas [122] seek to accelerate data mining problems.

3.7 Conclusion

Trends in Internet of Things (IoT) and Big Data require effective publish-subscribe systems to facilitate exchange of information between interested entities. A *data filtering algorithm* is at the core of a publish-subscribe system which determines the matching of a document (message) with queries (profile). Cloud computing offers cost effective and easy access to vast computing resources which are oftentimes comprise heterogeneous computing platforms. Existing publish-subscribe solutions do not leverage heterogeneity prevalent in Cloud effectively, thus limiting their operational effectiveness and are not elastic under heterogeneous Cloud environment.

In this chapter, we proposed a elastic publish subscribe system. Specifically, we first developed a data shaping technique to reorganize semi-structured data in a manner amenable for processing on parallel architecture machines and applied it to a publish-subscribe system. Then, we leveraged data shaping-based publish subscribe system to facilitate elastic and portable computing in heterogeneous Cloud environment. Our data shaping technique enables large number of work (kernels) resulting in efficient utilization of compute resources of a highly parallel architecture machines. Experiments using real datasets on multicore processors and GPGPU demonstrate that our data shaping-based approach delivers a scalable and high throughput publish-subscribe system.

Table 3.4: Kernel Generation using Data Shaping

Query set size (Q)	B	K(= $\lceil N/B \rceil$)	Total Kernels (=Q*B*K)
4	32	2650	339200
4	64	1325	339200
4	128	663	339200
4	256	332	339200
4	512	166	339200
4	1024	83	339200
8	32	2650	678400
8	64	1325	678400
8	128	663	678400
8	256	332	678400
8	512	166	678400
8	1024	83	678400
16	32	2650	1356800
16	64	1325	1356800
16	128	663	1356800
16	256	332	1356800
16	512	166	1356800
16	1024	83	1356800
32	32	2650	2713600
32	64	1325	2713600
32	128	663	2713600
32	256	332	2713600
32	512	166	2713600
32	1024	83	2713600

CHAPTER 4. DUPLICATE DETECTION IN SEMI-STRUCTURED DATA USING DATA SHAPING

Summary. State-of-the-art in duplicate detection in semi-structured data obtains significant improvement by exploiting the schema-related knowledge. Such schema-bound duplicate detection approaches, however, have severe limitations when dealing with multi-sourced, heterogeneous, high-velocity data streams. In this chapter, we propose a novel context-aware duplicate detection system which is workload- and complexity-aware, and is adaptable to the underlying computing platform. The system operates in schema-oblivious manner, and relies upon information theory based heuristic and data shaping technique for efficient, and scalable duplicate detection in multi-sourced, heterogeneous data sets. Experiments with real-world data sets show speed up of up to 8X over state-of-the-art schemes, while maintaining up to 92 percent accuracy. In addition, our data shaping technique for GPGPU processing speeds up the duplicate detection throughput by up to two orders of magnitude.

Keywords: Data Streams, Duplicate Detection, Semi-structured Data, GPGPU, Data Shaping

4.1 Introduction

Big Data occurs in structured, semi-structured, and unstructured forms in stationary or stream form. Ability to carry out fast, rapid analytics on Big Data is becoming extremely important for organizations. Fast analytics are especially important in the context of streaming data since the data generated by machines, devices, sensors, etc. need to be processed in a "timely" manner. Definition of "timely" processing is context dependent and ranges from few seconds to several hours.

The data streams originate from multiple sources, are heterogeneous in nature, and need to be processed in near real-time. A recent study envisages that in future, organizations have to deal with more and more unstructured and semi-structured data generated from sensors and devices [29]. The same study also predicts that in future a majority of workload in organizations will comprise the real-time analytics. Thus, the streaming data is that the data needs to be processed in their native forms, i.e. *on-the-fly*.

Data in raw form is not suitable for performing analytics and needs preprocessing. For example, data has quality issues. The poor data quality results into different types of errors and can lead to losses in business [117]. A previous study estimates that poor data quality can cost businesses close to $600 million each year [38]. Thus, the data needs cleaning before it can be used for any analytics.

To manage data originating from multiple sources data integration is needed. Data integration plays a key role in combining clinical, environmental, and demographic data with high throughput scientific data sets such as genomic data. Such data can be heterogeneous and the heterogeneity factor can be overcome by employing schema-matching (schema-translation) measures. Data fusion operation combines multiple records corresponding to a single real world object into a single record.

Problems such as data integration, data cleaning, and data fusion have to identify duplicate objects. Duplicates objects are multiple representations of the same real world object that differ from each other due to multiple representations storing erroneous, incomplete information, or missing data.

Volume and velocity factors associated with Big data make duplicate detection in multi-sourced heterogeneous streams even more challenging. A way to handle the volume and velocity is load shedding i.e. processing partial data sets. However, the load shedding is not desired in several cases as it can easily result into imprecise outcome leading to erratic and detrimental decisions. Hence, the duplicate detection system needs to consider every piece of information contained in the given data corpus. The time and memory efficient duplicate detection solutions can deal with volume and velocity related challenges. Moreover, a time and memory efficient duplicate detection solution can have a tremendous impact on data analytics. For example,

it can enable increased number of analyses over larger data sets which helps in the decision making process.

State-of-the-art in duplicate detection in semi-structured data has improved significantly due to recent studies [75], [71], [73], [72]. However, the Big Data phenomenon brings novel set of challenges for detecting duplicates in semi-structured data such as: (1) Dealing with multi-sourced heterogeneous data sets in a timely manner, and (2) Enabling duplicate detection over large data sets with a high throughput.

Contributions. In this chapter, we propose a novel context-aware duplicate detection system which is workload- and complexity-aware and is adaptable to the underlying computing platform. Our context aware solution is schema-oblivious and relies upon information theory and data shaping for time efficient context aware duplicate detection.

This work makes the following contributions:

- We consider the problem of approximate duplicate detection in semi-structured data (e.g. XML data) using Single Instruction Multiple Thread (SIMT) machine. Note that we do not address the text duplicate detection problem.

- We develop a heuristic based on document frequency (DF) to reduce computational complexity of the duplicate detection problem. Specifically, we use DF to generate a signature set of objects and leverage these signature sets to filter out the irrelevant duplicity candidates, and hence reduce overall number of pairwise comparisons. This step allows high throughput while sacrificing accuracy slightly.

- We develop a novel schema-oblivious, scalable, and time efficient duplicate detection system for detecting duplicates over semi-structured, multi-sourced, and heterogeneous data streams.

- We develop a data shaping technique to transform the hierarchical data and exploit SIMT machine for time efficient duplicate detection.

Chapter Organization. Rest of the chapter is organized as follows. Section 4.2 and Section 4.3 cover related work and background, respectively. We propose our context-aware

duplicate detection system in Section 4.4. Section 4.5 describes the data shaping technique. Sections 4.6, 4.7, and 4.8 details record linkage method using GPGPUs. Section 4.9 and 4.10 describe the experimental methodology and results obtained. Chapter concludes in Section 4.11.

4.2 Related Work

4.2.1 The Duplicate Detection Problem in Semi-structured Data

Several studies on duplicate detection in semi-structured data exist. DogmatiX [148] proposed an effective and efficient duplicate detection in semi-structured data. Our work differs from DogmatiX [148] in several respects: (1) Unlike DogmatiX, we do not define set of objects to compare. (2) Our work uses a different approach for reducing the number of pairwise comparisons. DogmatiX uses IDF measures for the same purpose. We use signature sets which is much smaller when compared to the object it represents to find duplicate candidates. (3) DogmatiX uses soft-IDF, a variant of inverse document frequency (IDF), to determine measure of similarity. We do not use IDF measures. We use document frequency (DF) values in our work to generate the signature set for objects.

Sorted XML neighborhood method (SXNM) [116] is based on the sorted neighborhood method [53], [40] which is widely used for duplicate detection in relational databases. Key idea in SXNM (and SNM) is to sort the objects based on key(s). We group the objects that are more likely to be similar together and avoid performing useless comparisons between these similar objects.

Milano et al. proposed a distance measure between two candidate XML objects [89]. The distance measure is based on the idea of overlays and it considers structure as well data values of XML objects. We do not use overlays for detecting duplicates. They also propose using Jaccard coefficients for comparing sets of descendants. Jaccard coefficients is not used in our work.

XMLDup [74], [75] uses Bayesian network model (BN) for detecting duplicates in XML objects. We do not use Bayesian network model in our work. XMLDup also exploits structure

of the XML objects to achieve a higher degree of efficiency. But, efficiency of our duplicate detection system is independent of the structure of XML objects. Also, our system does not exploit schema knowledge.

Recently, Leitao et al. reported an improved version of XMLDup which incorporates a pruning algorithm [75]. The pruning algorithm improves the efficiency of duplicate detection and helps XMLDup to outperform DogmatiX [148].

In order to optimize the duplicate detection process some studies exploit the hierarchical structure of semi-structured objects [71], [73], [72]. A detailed overview of the duplicate detection work is provided in [98].

Our work differs from [15], [48] which deal with approximate duplicate detection problem in the context of ordered (unordered) tree data sets. We exploit context related information while ignoring the schema knowledge, and deliver a time efficient solution for time critical applications such as high velocity data streams.

A node encoded tree sequence (NETS) approach is proposed in context of filtering twig queries over XML documents [124]. We build a Pre-Pre-Post encoding algorithm that builds upon the the NETS algorithm. The study in NETS leverages tree sequencing to convert the problem of twig filtering into a sub string matching problem. Our work differs from [124] in two aspects: algorithmic level and objective level. (1) We cover the attributes (which are registered in pre-order traversal) while sequencing the tree structure which is ignored by NETS. (2) We use tree sequencing to overcome the architectural limitations of GPGPU.

4.2.2 The Record Linkage Problem in NoSQL

Recent researches on record linkage problem occurring in semistructured data can be organized into the following three major categories: (1) micro-level optimizations or stage optimization approaches, (2) cloud computing approaches, and (3) parallel processing approaches.

Table 4.1 depicts categorization of recent researches on record linkage problem occurring in semistructured data domain.

Table 4.1: Classification of recent research on record linkage in semistructured data.

Category/approach	Research
Stage optimization	Hassanzadeh [51], Ramadan[118], Wang [147], Kim [63]
Cloud computing based	Paradies [110], Papadakis [109]
Parallel processing based	Kirsten [64], Mamun [82], Sehili [125]

4.2.2.1 Stage Optimizations

Recent studies focusing on optimizations of the stage(s) in record linkage problem can be further categorized based on the particular stage where they are focused. Specifically, following subcategories are possible: identification of linkage points, blocking mechanisms, similarity techniques, and hashing-based approaches.

Identifying Linkage Points Hassanzadeh et al. propose SMaSh framework, which discovers linkage point in a scalable, online manner. The framework operates in multiple stages: (1) a task scheduler, storage, and indexing backend; (2) registering and loading data sets; (3) analysis and indexing; and (4) search and discovery of linkage points. A key component of their work is identification of linkage points in an efficient manner. This is realized by [51]. Hassanzadeh et al. propose a framework comprising a library of efficient lexical analyzers and similarity functions, and a set of search algorithms for identification of linkage points over web data.

Blocking Mechanism. The entity resolution problem in real-time setting for structured data sets was studied in [118], which proposes a dynamic sorted neighborhood indexing methodology. The indexing method proposed uses multiple trees with different sorting keys, which can be used for real-time entity resolution for read-most databases. Papadakis and Nejdl present an entity resolution method for heterogeneous information spaces [109]. Their focus was on developing novel blocking methods that scale up entity resolution within large, noisy, and heterogeneous information spaces. Their approach is attribute-agnostic and relies on values of entity profiles for building blocks.

Similarity Techniques Optimizations. Wang et al. studied similarity measures in the context of entity-matching problem [147]. The study in [147] characterizes various similarity measures used in entity matching and provides a solution to automate the selection process of appropriate similarity functions.

Hashing-Based Approaches. Kim and Lee propose HARPA [63], which enables fast iterative-hashed record link- age for large-scale data collections. HARPA gains by dynamic and reusable hash table and exploitation of data characteristics.

4.2.2.2 Cloud Computing-Based Approaches

A cloud-based solution for entity matching in semistructured data is proposed in [110]. The study in [110] combines ChuQL, an extension of XQuery with MapReduce and a blocking technique for entity matching.

4.2.2.3 Parallel Processing-Based Approaches

Kirsten et al. propose strategies to partition data for parallel entity matching [64]. They also present a service-based infrastructure for parallel entity matching on different hardware settings. In order to reduce communication requirements, caching and affinity-based scheduling of match tasks are supported. The study in [82] reports efficient sequential and parallel algorithms for linking records. There solution can handle any number of data sets. One recent work proposes a GPU-based solution for privacy-preserving record linkage for structured data [125].

A comparison of the popular methods for entity matching is provided in [66]. A tutorial on the state-of-the-art in knowledge harvesting from web and text sources is provided [133]. A survey of entity resolution and record linkage methodologies is provided in [21].

4.3 Background

In this section, we review terminologies and concepts relevant to this chapter. Specifically, we cover NoSQL databases and definitions on document frequency (DF) and distance metrics. We also provide a background on hash table and GPGPU, and tree data model.

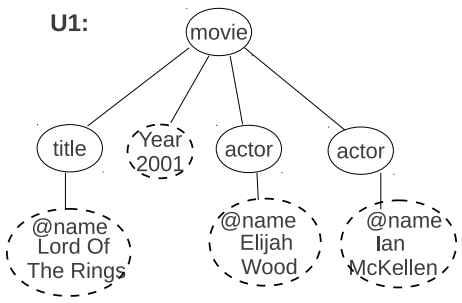


Figure 4.1: Tree U1: Only leaf nodes, shown in dotted circle, are considered for signature sets.

4.3.1 Tree Data Model

In tree data model, labels are represented as nodes [140], [93]. Refer to Figure 4.1. The tree model data set is encoded in XML [153]. Other encodings such as JSON [58] are also applicable. Labels in the tree model correspond to elements in the XML encoding. Elements are represented by an opening tag $< element >$ and a closing tag $< /element >$. Elements can have attributes and text (or content). Texts basically correspond to the unlabeled leaf nodes in the tree. The opening and closing tags delimit texts. Input data set comprises several tree model objects encoded in XML. This is a typical mode.

4.3.2 NoSQL Databases

NoSQL [93] databases are a special type of databases, which are characterized by the features such as being non-relational, distributed, open source, and horizontally scalable. NoSQL databases are scheme free, easily replicable, simple to use, and able to handle a huge amount of data. NoSQL databases are especially useful when businesses need to access and analyze massive volumes of data, which is unstructured or semistructured in nature.

Popular NoSQL databases are Apache Cassandra, Simple DB, Google BigTable, MemcacheDB, etc. The NoSQL databases can be classified as wide-column store, document store, key-value store, graph databases, and others. Refer [93] for more details on NoSQL databases. A survey of NoSQL databases is provided in [50].

4.3.3 Hashing Table

A hash table supports the basic dictionary operations such as insert, find, and delete. A given entry is inserted into the hash table as follows: obtain a hash value of the given entry using an appropriate hash function, and then write the given entry at the address given by hash function. A hash collision occurs when two (or more) keys map to the same hash value in the hash table. In the context of hash table, three design decisions are important: (1) Size of the hash table, (2) Hash function, and (3) Collision resolution. The literature suggests several approaches to handle collisions in hash table. These approaches are classified into open addressing and closed addressing [28], [85]. An example of open addressing is separate chaining. In separate chaining, all the keys that map to the same hash value (in the hash table) are kept in a list or a bucket. Examples of closed addressing are linear probing, quadratic probing, double hashing, perfect hashing [81], cuckoo hashing [107], etc.

4.3.4 Definitions

Before we present our approach, we define the following.

4.3.4.1 Document Frequency

Document frequency ($DF(t)$) [83] of a term t is defined as the total number of documents in a collection in which term t occurs. A related definition is Inverse Document Frequency (IDF). The intuition behind DF and IDF is that when as a query term occurs in several documents, then that term is not a good discriminator. However, if a given term occurs in fewer documents, then its discriminating power is higher and should be assigned a higher weight (or priority).

4.3.4.2 Levenshtein Distance

The Levenshtein distance is a common metric used for measuring the difference between two strings or sequences [76]. Basically, the Levenshtein distance between two strings is the minimum numbers of single-character edit operations needed to change one string into another.

Allowed edit operations are insertions, deletions, or substitutions. The Levenshtein distance is also referred to as string edit distance. Several variations of edit distance exist.

4.3.4.3 Edit-based Similarity

The edit distance between two strings s_1 and s_2 is the minimum cost of transforming s_1 into s_2 using a specified set of edit operations with associated cost functions. A simple variant of edit distance is the Levenshtein distance. The Levenshtein distance LevDist(s_1, s_2) for strings s_1 and s_2 is computed as the minimum number of character insertions, deletions, and replacements needed to transform s_1 into s_2. For more on string matching refer [99]. The idea behind tree edit distance is similar to string edit distance. In tree edit distance based approach, operations involve nodes instead of characters. Refer [20] for details.

4.3.4.4 Similarity Metric

In a given data set, let D and $'$ be two semistructured data records. We define these records as linked if their similarity score Sim(D, D$'$) is greater than a predefined threshold value. A common formula to calculate similarity score is the following: Sim(s_1, s_2) = [1 - LevDist(s_1, s_2)]/len(s_1), where LevDist(s_1, s_2) is the Levenshtein distance between strings s_1 and s_2, and len(s_1) is the length of string s_1.

4.3.4.5 Duplicate Detection Problem

We assume that a data set comprises a number of tree structured objects (or documents). In a given data set let D and D$'$ be two objects. These objects are defined as duplicates if their similarity score Sim(D, D$'$) is greater than a predefined threshold value. A common formula to calculate similarity score is: Sim(s_1, s_2) = [1 - LevDist(s_1, s_2)]/len(s_1), where len(s_1) is the length of string s_1.

4.3.5 GPGPU

GPGPUs are Single Instruction Multiple Thread (SIMT) machines comprising thousands of cores or streaming processors (SPs). GPGPUs are highly parallel architecture machines. For

details refer Section 2.3.1. In this chapter, we refer GPGPUs and Single Instruction Multiple Thread (SIMT) interchangeably.

4.4 Context-aware Duplicate Detection System

We first provide the rationale behind designing a context-aware duplicate detection system which is schema-oblivious and time efficient. Next, we provide the data model and the data stream integration architecture. Finally, we describe our context-aware duplicate detection system.

Time overhead incurred in duplicate detection process can be reduced by two mechanisms: (1) Reduction in size of the needed computation, and (2) Parallelization of the computations. The size of computation can be reduced by minimizing total number of string comparisons required towards the duplicate detection process. The number of comparisons can be minimized in the following two ways. (1) Use only those leaf nodes (external, outer, or terminal nodes having no child nodes) which are distinct and avoid the commonly occurring nodes. (2) Assign numeric codes to tags and text values which avoids string comparison operation where ever possible.

High quality data occurs in business data sets and machine generated data. Machine generated data however may be prone to missing data problem, though. Meta data are also of high quality data. For example, GIS meta data associated with images in image database, etc. High quality data have a high re-use factor. Such data can be trusted and exploited in detecting duplicates.

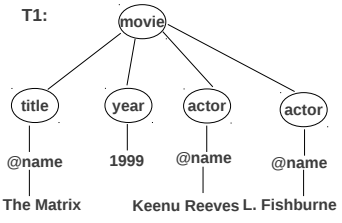


Figure 4.2: Tree T1.

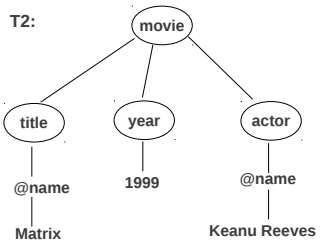


Figure 4.3: Tree T2.

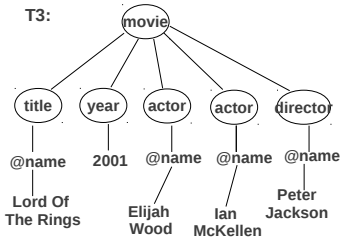


Figure 4.4: Tree T3.

4.4.1 Data Model

We assume a tree-model data where labels are represented as nodes. Refer Figure 4.2–Figure 4.4. The tree model data set is encoded in XML. Other encoding like JSON are also applicable. Labels in the tree model correspond to elements in the XML encoding. Elements are represented by an opening tag <element> and a closing tag < /element>. Elements can have attributes and text (or content). Texts basically correspond to the unlabeled leaf nodes in the tree. Texts are delimited by the opening and closing tags. Input data set comprises several tree model objects encoded in XML.

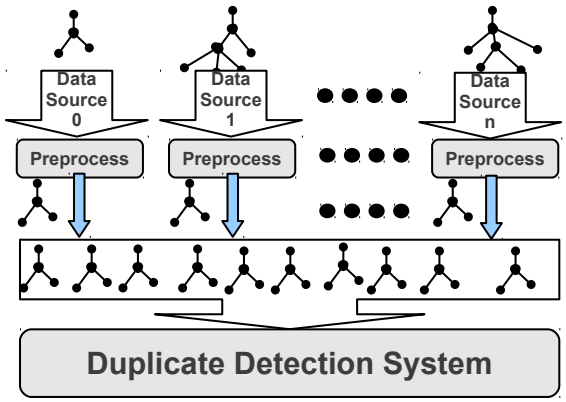


Figure 4.5: A high level data integration architecture.

4.4.2 Data Integration Model

Figure 4.5 shows a high level model of the data integration architecture. The model assumes multiple data streams originating from various data sources. Each data stream comprises semi-structured objects such as trees. Objects belonging to different data streams can be heterogeneous, i.e. dissimilar, in structure. The heterogeneous data streams are preprocessed. Type of preprocessing applied depends upon nature of data streams. For example, unordered data sets are dealt by preprocessings such as lexicographical ordering. The preprocessing step take care of the heterogeneity aspect and results in better shaped data objects. The input to the duplicate detection system are tree objects in their native format such as XML, JSON, etc.

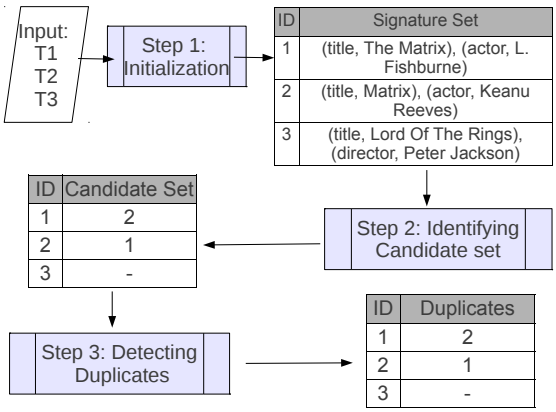


Figure 4.6: Work flow of duplicate detection process.

4.4.3 Context-Aware Duplicate Detection System

Our context-aware duplicate detection system is schema-oblivious i.e. it assumes that schema of the data set is not known *a-priori*. Thus, our system does not rely upon the schema information of data set for detecting duplicates. This is contrary to some of the existing proposals [74], [71] for duplicate detection which exploit *a-priori* knowledge of the schema to design efficient duplicate detection algorithms. Another study [89] uses XML structure aware distances in order to identify duplicates objects from the XML data corpus.

The duplicate detection system operates in three steps: (1) Initialization, (2) Identification of candidate sets, and (3) Detecting duplicates. Figure 4.6 illustrates the work flow.

4.4.3.1 Step 1: Initialization

We make a key observation that a given data corpus can have terms which are too frequent. Such terms do not help in identifying duplicates. On the other hand too infrequent terms can increase false negatives in duplicate identification. Hence, it is sensible to use terms which are neither too frequent nor too rare when looking for duplicates. Document frequency – a robust technique from information theory – helps in defining relative occurrence of terms. We make use of document frequency (DF) to select useful terms and prune the potential duplicate candidates. Specifically, we use terms which fall in the pre-specified range (<MIN_DF, MAX_DF>).

Input to the initialization step is an XML encoded data set (comprising tree-model objects) and output is an object-wise signature set. The initialization step is described as follows.

(1) Parse input data set using an appropriate parser (e.g. expat parser for XML data).

(2) Record the document frequency of text (or content) occuring as an unlabeled leaf node.

(3) Compute document frequency of all the text terms.

(4) Finally, for every object, select text terms which fall in the pre-specified range (<MIN_DF, MAX_DF>) and refer this as signature set of that object.

A tuple in signature set consists of element id (e), content id (vid) and the actual content (val). Note that element id (e) and content id (vid) are represented as numeric values. And actual content (val) is represented as a string. For example, signature set of the i^{th} object comprising k-tuples is represented as $s_i = [(e_0, vid_0, val_0), (e_1, vid_1, val_1), ..., (e_k, vid_k, val_k)]_i$. To compare text values, we use formula $Sim(val_0, val_1) = [1 - LevDist(val_0, val_1)]/len(val_0)$ as a measure of similarity. Term $LevDist(val_0, val_1)$ is the Levenshtein distance (a measure of string edit distance) between values val_0 and val_1. Term $len(val_0)$ is the length (number of characters) of string val_0.

4.4.3.2 Step 2: Identifying Candidate Set

This is data pruning step. Input to this step is the list of object signature sets and output is a set of duplicity candidates for that object. Algorithm used for identifying set of duplicate candidates is listed as Algorithm 5.

Algorithm 5 takes list of signature set (S) as input. It compares signature set of all objects with each other (Line 6). To compare two tuples, we follow the following protocol: (1) Match element id (e) of tuples, (2) Match content id (vid), and (3) Match actual text content (vid) using formula Sim(.) mentioned in the initialization step.

Every tuple of set s_i is compared with every tuple of set s_j and Cand_Sim_Score is calculated (Line 7). A match at any stage results in assigning a new Cand_Sim_Score, and further matching operations are not carried out for the tuples. If Cand_Sim_Score exceeds the Candidate Similarity Threshold (θ), object o_j is added to C_i, the candidate set of i^{th} object (Lines 8-9).

Algorithm 2 IdentifyCandidateSet(S)

Input: S = (s_0, s_1, ..., s_N), a list of signature set of N-1 objects.
Output: C = (C_0, C_1, ..., C_N), a list of duplicate candidates of N-1 objects.
C_i: Candidate set for the i^{th} object (i < N).
Cand_Sim_Score: Candidate Similarity Score.
θ_{cand}: Candidate Similarity Threshold.

```
1: for every s_i in S s.t. i ∈ [0, N) do
2:   for every s_j in S s.t. j ∈ [0, N) do
3:     if i ≠ j then
4:       C_i ← ∅
5:       Cand_Sim_Score ← 0
6:       Compare s_i and s_j tuple-wise
7:       Assign Cand_Sim_Score
8:       if Cand_Sim_Score > θ_cand then
9:         C_i ← C_i ∪ o_j
10:      end if
11:    end if
12:  end for
13: end for
```

Algorithm 3 IdentifyDuplicates(C)

Input: C = (C_0, C_1, ..., C_N), a list of duplicate candidates of N-1 objects.
Output: D = (D_0, D_1, ..., D_N), a list of duplicates of N-1 objects.
D_i: Duplicate set for the i^{th} object (i < N).
Dup_Sim_Score: Duplicate Similarity Score.
θ_{dup}: Duplicate Similarity Threshold.

```
1: for every o_i in InputData s.t. i ∈ [0, N) do
2:   for every o_j in C_i do
3:     D_i ← ∅
4:     Dup_Sim_Score ← 0
5:     Compare o_i and o_j: TreeCompare(o_i, o_j)
6:     Assign Dup_Sim_Score
7:     if Dup_Sim_Score > θ_dup then
8:       D_i ← D_i ∪ o_j
9:     end if
10:  end for
11: end for
```

4.4.3.3 Step 3: Detecting Duplicates

After Step 2, we obtain a set of duplicity candidates for every document. In this step, we refine the set of duplicity candidates for every object. Specifically, we carry out intensive pairwise comparison and identify the duplicates. Input to this step is the list of duplicity candidate set (C) and output is a list of duplicate set. Algorithm used for detecting the set of duplicates is listed as Algorithm 3.

Algorithm 3 takes list of candidate set (C) as input and produces a list of duplicates for objects. Objects are compared using TreeCompare(.) algorithm (Line 5). TreeCompare(.) compares all the nodes of o_i with all the nodes of o_j. Depending on the outcome of TreeCompare(.), Dup_Sim_Score is calculated (Line 6). If Dup_Sim_Score exceeds the value of Duplicate Similarity Threshold (θ_{dup}), object o_j is added to D_i, the duplicate set of i^{th} object (Lines 7-8).

4.5 Shaping Tree Data for Parallel Architecture

In this section, we address the second aspect of our context-aware duplicate detection system i.e. parallelization of the computation. We use a data shaping technique for this purpose. Objective of this data shaping technique is to exploit the structured, rigid architecture of SIMTs such as GPGPU. The data shaping technique prepares semi-structured input data in order to exploit compute, memory resources available on SIMTs.

4.5.1 Encoding Tree Data

We describe how we encode the tree Data for GPU-like machines. We use Pre-Pre-Post Tree Sequencing algorithm to encode tree objects for detecting duplicates using GPUs. The Pre-Pre-Post Tree Sequencing algorithm encodes a given tree object using the following steps: (1) Encode START tag in Pre-order, (2) Encode node content in Pre-order, and (3) Encode END tag in Post-order.

Figure 4.7 illustrates Pre-Pre-Post encoding. It depicts in (a) a tree T1 (b) XML encoding of T1 and (c) Pre-Pre-Post encoding of T1. Note that the Pre-Pre-Post encoding is similar to object serialization adopted widely in XML. In Figure 4.7 (a), the nodes also carry contents.

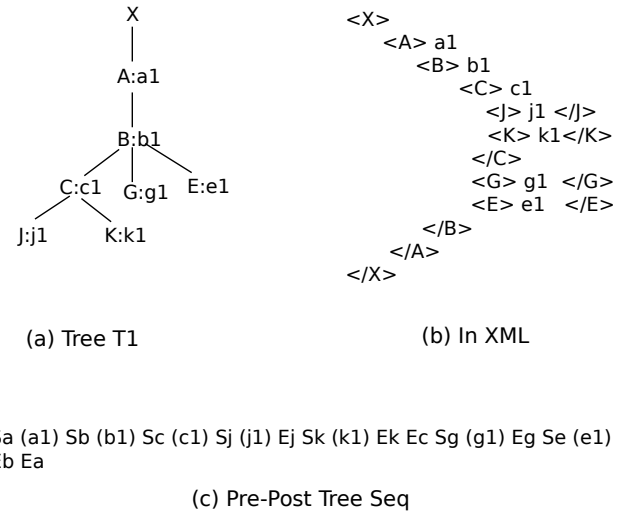


Figure 4.7: (a) A tree T1, (b) XML encoding of the T1, and (c) Pre-Pre-Post encoding of T1.

For example, contents on nodes A, B, C, G, E, J, and K are a1, b1, c1, g1, e1, j1, and k1 respectively.

String to Numeric Conversion. Cost prohibitive string comparison operations are avoided. We convert string values such as tags, content, etc., into numeric codes. To this end we maintain a mapping table for tags and contents. Each START (or END) tag is assigned a two byte unique numeric code. For example, Start_symb = 80 [most significant byte] + ID_of_tag(tag) [least significant byte]. End_symb = C0 [most significant byte] + ID_of_tag(tag) [least significant byte]. Each content is also assigned a two byte unique numeric code. Estimated size of tag table and content table are ∼20 entries and ∼2000 entries, respectively.

4.5.2 Tree Comparison

In the following, we discuss the CompareTreeSeq Algorithm in Section 4.5.2.1, its important properties, and their implications for streaming data scenario. Next, we explain the application of CompareTreeSeq Algorithm for managing computations in context of semi-structured data streams. The encoded trees (or tree sequences) are compared using CompareTreeSeq Algorithm. Algorithm 4 shows the outline sketch of CompareTreeSeq. We discuss two implementations of CompareTreeSeq Algorithm shortly in Section 4.5.4.

4.5.2.1 CompareTreeSeq Algorithm

The CompareTreeSeq Algorithm takes the start and end indices of the reference tree and candidate tree as input. Reference tree (RefTree) and candidate tree (CandTree) are obtained using the start and end indices of reference and candidate trees. Another input to algorithm is maximum number of iterations referred as "MaxIter" allowed for comparison.

Reference tree (RefTree) and candidate tree (CandTree) are compared byte-wise using XOR operation (Line 2). Matching parts of RefTree and CandTree are dropped (Line 3) and size of the data structure is updated (Line 4). These steps are repeated for "MaxIter" times.

Space Complexity. Let the size of reference tree and candidate tree be m and n, respectively, where m and n are number of nodes in the respective trees. Since encoding of every node (and its content) appears twice in encoded tree sequence, the size of RefTree[ ] and CandTree[ ] are O(n) and O(m), respectively. Size of OutXOR[ ] is O(n) as well.

The CompareTreeSeq Algorithm primarily works with three array data structures namely RefTree[ ], CandTree[ ], and OurXOR[ ]. The space complexity of the CompareTreeSeq Algorithm is O(max(m, n)).

Time Complexity. The time complexity of CompareTreeSeq depends upon type of data set. In other words, the time overhead incurred is different for ordered and unordered trees.

Consider a dataset comprising ordered trees. The worst case time complexity of CompareTreeSeq Algorithm is O(n′). For unordered trees, the worst case time complexity is quadratic or O(n′×n′) where n′ = max(m, n).

We also observe that theoretically, the best case time complexity of CompareTreeSeq Algorithm is equal for both ordered and unordered trees. The best case time complexity of CompareTreeSeq Algorithm can be O(1), resource permitting.

4.5.2.2 Applying CompareTreeSeq

Algorithm CompareTreeSeq posses several key properties: (1) *Alignment Property* – Time complexity of comparing two tree objects is inversely (directly) proportional to the similarity (edit distance) between the given objects. This is because for similar trees the CompareTreeSeq

Algorithm 4 CompareTreeSeq(RefTree[ ], CandTree[ ], MaxIter)

Input: RefTree[ ], CandTree[ ], MaxIter
Output: TreeMatched
OutXOR[sizeof(RefTree[ ])] ← 0;
NumIter ← 0;
TreeMatched ← 0;

```
1: while (NumIter++ < MaxIter) do
2:    OutXOR[ ] ← (RefTree[ ]) XOR (CandTree[ ])
3:    Drop matching parts of RefTree[ ] and CandTree[ ]
4:    Update size of RefTree[ ] and CandTree[ ]
5:    if (!sizeof(CandTree[ ])) then
6:       TreeMatched ← 1;
7:       EXIT
8:    end if
9: end while
10: return TreeMatched
```

Algorithm produces identical sequences. (2) *Size Invariance Property* – Time complexity of comparing two tree objects is independent of the size of given trees constrained by the number of available processors. (3) *Architecture Neutral Property* – CompareTreeSeq Algorithm is portable and executes independently of the underlying architecture of the computer system.

Streaming data scenario require the following to hold. (1) Realistic, bounded time complexity, (2) Ability to make quick and appropriate decisions timely, and (3) Ability to analyze high-value data streams more intensely. We can exploit the properties of CompareTreeSeq Algorithm and address the requirements associated with streaming data.

Better resource allocation helps in predicting and achieving a bounded time complexity which is realistic. Similarly, futile computations i.e. computations involving trees which are not likely to result in desired level of similarity can be abandoned at earlier stages. Such a timely decision can be significant from the perspective of time efficiency. Finally, a high-value stream can be analyzed more intensely by offering a higher grade of service to such streams. A higher grade of service can be realized by exploiting a-priori knowledge about data streams, and then dedicating additional compute and memory resources.

4.5.3 CompareTreeSeq vs. Edit Distance

When we consider the edit distance based algorithms, two questions are important: (1) Why not use just the tree edit distance algorithm for comparing trees? (2) Why not to use string edit distance algorithm for comparing the sequences produced by CompareTreeSeq Algorithm?

The fundamental philosophy behind tree edit distance and string edit distance algorithms is same. The string edit distance (SED) approach is compute intensive when compared to CompareTreeSeq approach. Time overhead incurred in computing string edit distance for given two trees is cost prohibitive. There are primarily two factors which makes such computations cost prohibitive: (1) A SED based approach first computes the edit distance and then declares a measure of similarity between the given pair of trees. (2) Computing edit distance between two trees (or strings) itself has a quadratic time complexity in terms of the size of trees (strings) i.e. number of nodes (alphabets) in the respective trees (strings).

The edit distance based approaches are therefore not appropriate for time critical applications. When operating under conditions, such as load shedding, the edit distance based approaches can lead to undesired outcomes. The CompareTreeSeq approach is time-aware and appropriate for streaming scenario.

4.5.4 Computation Model

We explain the computation model used for execution using an example. The example works with a dataset comprising eight tree objects. The GPGPU comprises several streaming multiprocessors (SMs) and each SM consists of four streaming processors (SPs). The size of a thread block (B) is four.

Let there be N objects in the dataset. Total number of grids depends upon the number of objects in the dataset which is N. Each grid has $\lceil N/B \rceil$ blocks. Figure 4.8 depicts an example of the computation model for N = 8. We explore the design space for orchestrating computations on a parallel architecture such as a GPGPU. We present two computation strategies: (1) Thread-Sequential and (2) Thread-Parallel.

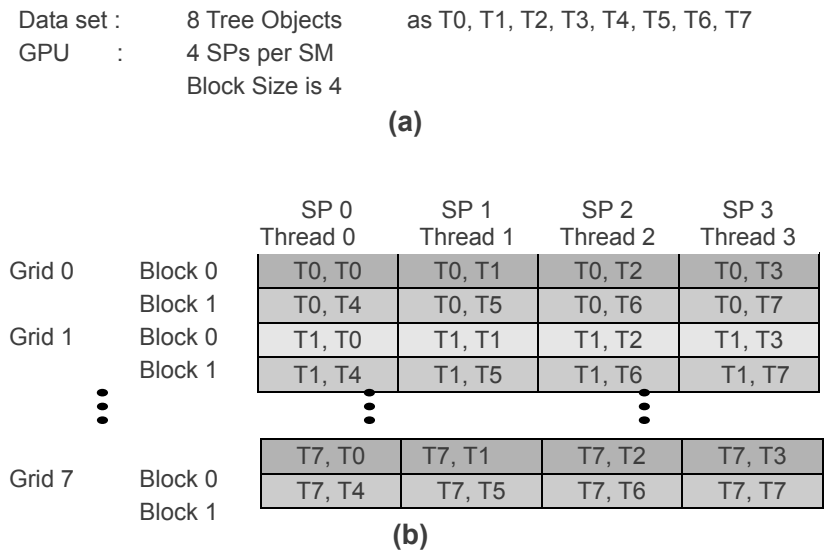


Figure 4.8: (a) Workload and GPGPU configuration. (b) Baseline computation model. (SP is streaming processor, SM is streaming multiprocessor).

4.6 Record Linkage Using GPGPU

This section describes the process of record linkage in semistructured data sets on a GPGPU or GPGPU-like parallel hardware. We adopt a data-shaping-based approach, applying appropriate transformations to data and algorithm(s) involved in parallelizing the computations. Parallelization of computations involved helps exploit the structured and rigid architecture of SIMT/SIMD-like parallel machines in an efficient manner.

The proposed record-linking solution operates in three stages: (1) preprocessing, (2) identification of candidate sets, and (3) linking records. Figure 4.9 illustrates the process flow.

4.6.1 Stage 1: Preprocessing

The terms that occur too frequently in the data sets do not help in identifying the linked records as it increases the probability of false positive. Similarly, infrequently occurring terms can increase the probability of false negatives in identification of linked records. Hence, it is prudent to use terms that are neither too frequent nor too rare when looking for linked records.

Document frequency (DF), a robust searching technique from information theory, helps in defining relative occurrence of terms. We make use of DF to select useful terms, which later on helps prune the not-likely record candidates. Specifically, we use terms that fall in the pre-specified range (<MIN_DF, MAX_DF>) for further search. Input to the preprocessing stage is an XML-encoded data set (comprising tree model objects). Output of the pre-processing stage is a record-wise signature set. Briefly, the preprocessing stage comprises the following.

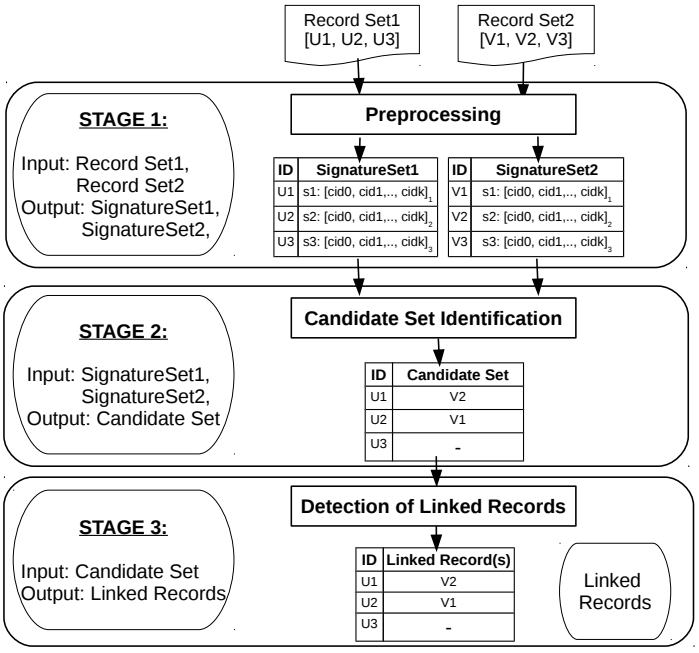


Figure 4.9: Framework for record linkage process. The figure shows the process using two sets of records: Record Set1 and Record Set2.

4.6.1.1 Data Encoding and DF Update

Input data set is parsed using an appropriate parser (e.g., expat parser for XML-encoded data set). On encountering a leaf node, we update its DF. We do not consider the intermediate nodes (tags and textual values) when developing the signature sets for object.

4.6.1.2 Forming Signature Sets

For every object, select text terms which fall in the pre-specified range (<MIN_DF, MAX_DF>), and refer this as signature set of that object. A signature set tuple comprises ContentID.

ID	Signature Set
U1	(The Matrix), (L. Fishburne)
U2	(Matrix), (Keanu Reeves)
U3	(Lord Of The Rings), (Peter Jackson)

Figure 4.10: Example of signature sets for three tree objects namely U1, U2, and U3. (Assume U2 and U3 similar to U1 shown above.) Note that actual signature set contains hashed values instead of text values.

For example, signature set of the i^{th} object comprising k-tuples is represented as $s_i = [cid_0, cid_1, \ldots, cid_k]_i$. Refer Figure 4.10.

We implement Stage 1, that is, preprocessing stage using a hashing approach. Hashing approach helps avoid serious processing bottleneck. For example, with data encoding, there is no need for the costly string comparison-based search operations through the table storing the string terms and the numeric codes assigned to them. Moreover, using numeric codes instead of string values helps in storage and access as well. Preprocessing stage that leads to overall efficient processing is the focus of Section 4.7 and Section 4.8

4.6.2 Stage 2: Identifying Candidate Records

This is data reduction stage. Input to this stage is the list of object signature sets, and output is a set of candidate-linked records for that object. Algorithm used for identifying a set of candidate-linked records is listed as Algorithm 5.

Algorithm 5 takes a list of signature sets (S) as input. It compares signature set of all objects with each other (Line 6). Every tuple of set s_i is compared with every tuple of set s_j, and Cand_Sim_Score (indicating candidate similarity score) is calculated (Line 7). A match at any stage results in assigning a new Cand_Sim_Score, and further matching operations are not carried out for the tuples. If Cand_Sim_Score exceeds the candidate similarity threshold denoted by (θ), object o_j is added to C_i, the candidate set of i^{th} object (Lines 8-9). Stage 2 is amenable to parallel processing. This stage involves identification of candidate set for all

the objects of Set1. Identification is single-process multiple-data kind of processing. Stage 2, therefore, can be and is executed on GPU.

Algorithm 5 IdentifyCandidateSet(S)

Input: S = (s_0, s_1, ..., s_{N-1}), signature set of N-1 objects.
Output: C = (C_0, C_1, ..., C_{N-1}), duplicate candidates for N objects.
C_i: Candidate set for the i^{th} object (0<= i <N).
Cand_Sim_Score: Candidate Similarity Score.
θ_{cand}: Candidate Similarity Threshold.

```
1: for every s_i in S s.t. i ∈ [0, N] do
2:   for every s_j in S s.t. j ∈ [0, N] do
3:     if i ≠ j then
4:       C_i ← ∅
5:       Cand_Sim_Score ← 0
6:       Compare s_i and s_j tuple-wise
7:       Assign Cand_Sim_Score
8:       if Cand_Sim_Score > θ_cand then
9:         C_i ← C_i ∪ o_j
10:      end if
11:    end if
12:  end for
13: end for
```

4.6.3 Stage 3: Linking Records

After Stage 2, for every object in Set1, we have identified a set of candidates belonging to Set2. In this stage, we refine the set of candidates for every object. Specifically, we carry out intensive pairwise comparison and identify the linked records. Input to this stage is the list of candidate set C, and output is a list of linked sets. The algorithm takes the list of candidate set C as input and produces a list of linked records for each object. We compare all nodes of o_i with all nodes of o_j, and LR_Sim_Score is calculated. If LR_Sim_Score exceeds the value of linked record similarity threshold denoted by (θ_{LR}), object o_j is added to LR_i, the linked record set of i^{th} object. Stage 3 is also amenable to parallel processing as it also involves a task that is essentially single process multiple data in nature. Stage 3 is also executed on GPU.

4.7 Signature Selection

In this section, we first review the concept of hash table data structure. Then, we discuss a naive approach based on hash table. Then, we describe our efficient hash-based approach for data encoding and candidate selection, the design rationales, and working. We also provide a discussion on the proposed hashing-based data structure.

4.7.1 Baseline Hashing-based Approach

Hash table is a popular data structure. Typically, a hash table supports the basic dictionary operations such as insert, find, and delete. A given entry is inserted in the following way: obtain a hash value of the given entry using an appropriate hash function, and write the given entry at the address given by hash function. In the context of hash table, three design decisions are important: size of the hash table, hash function, and collision resolution. The literature suggests several approaches to handle collisions in hash table. These approaches are classified into open addressing and closed addressing [28], [85]. An example of open addressing is separate chaining. Examples of closed addressing are linear probing, quadratic probing, double hashing, perfect hashing [81], cuckoo hashing [107], etc.

A naive approach to apply hashing mechanisms for pre-processing stage, that is, Stage 1 is shown in Algorithm 6.2. The basic idea is as follows. Given a term t, obtain hash h of t. If h is outside the range of the hash table, then take a MOD of h and assign it to h. Update the DF count at address h in hash table. Update the document ID list.

The baseline approach discussed above has a serious limitation. Memory footprint of the hash table is prohibitively large and potentially a memory bottleneck. Recall that the record linkage problem in the era of Big Data must leverage the compute power of SIMT machines to address the processing time challenges. However, SIMT machines are limited in terms of memory. Thus, reducing the memory requirements of data structures and algorithms used to this end is critical to take advantage of SIMTs.

Reducing the size of hash table is an option, but it results in false collision leading to an artificial rise in the DF count of some terms. This effect, that is, the artificial rise in the DF

count of some terms, can adversely affect the record linkage task by reducing the accuracy of records linked.

Algorithm 6 BaselineApproach(t)

T: A hash table of size S

1: $h \leftarrow Hash(t)$
2: **if** $(h > S)$ **then**
3: $h \leftarrow (\ h\ \%\ S)$
4: **end if**
5: Update the DF at address h in the T
6: Update document ID list

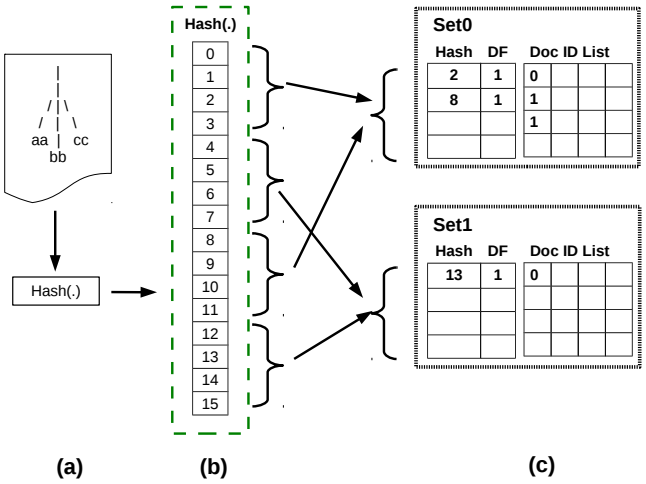


Figure 4.11: Signature Selector data structure. (a) A document and hashing of the leaf node terms. (b) The mapping of hash values to Signature Selector data structure (without hash table). (c) The organization Signature Selector into sets Set0 and Set1.

4.7.2 Signature Selector Data Structure: A Novel Approach

While a hash table is appropriate for dictionary operations *(find, insert, and delete)*, the requirements of data encoding and candidate selection are different. For example, the Signature Selector data structure (SignatureSel) need not perform any deletions. It only needs to carry out *find* and *insert* operations. The hash value collisions (or true collisions) affect candidate selection process by artificially inflating the DF count of few terms. We provision an auxiliary data structure, which records the IDs of the documents (trees) in which the terms occur.

Figure 4.11 depicts the design of our Signature Selector and its operation. Figure 4.11(a) shows that terms accompanying leaf nodes are considered for the signatures. It also shows that a hash value of such terms is generated using Hash() function. The hash value thus obtained is used as an entry in Signature Selector data structure. Since the size of Signature Selector is much smaller than the range of hash values obtained from Hash(), we need to map h to an appropriate line (and hash bucket) in Signature Selector. Mapping scheme is shown in Figure 4.11(b).

Figure 4.11(c) shows the Signature Selector data structure. Signature Selector has three fields: (1) Hash which holds the hash of a given term, (2) DF which is the DF of the given term, and (3) List which contains all the documents in which the given terms occur. Note that, the size of the list, which contains document IDs, can become very large if the given term occurs in a large amount of documents. This is a potential problem. We address this issue in an intelligent manner. We conjecture that if a given term occurs in too many documents, its discerning power is limited, and hence its effectiveness to help identify potential candidate records (and potential linked records) is also very limited. Thus, it is prudent not to use such terms when identifying candidate records. Consequently, such terms have no relevance in the Signature Selector data structure. Hence, there is no need to maintain a list of documents for such terms. Formally, if the DF of a given term exceeds the MAX DF threshold, we do not maintain the list of documents in which the given term occurs.

Algorithm 7 describes the mapping and update procedure. For a given term t, we obtain hash h of t in Line 1. In Line 2, we derive the ID of hash bucket, $bucketID$, to which h is written into. In Line 3, we derive address (in $bucketID$) at which h is to be written. In Line 5, DF of h is incremented. If DF is below DF_MAX threshold, the document ID list is updated (Lines 6 and 7).

4.7.3 Forming Signature Sets

For every object, we select text terms that fall in the pre-specified range (<MIN_DF, MAX_DF>), and we refer this as the signature set of that object. A signature set tuple comprises ContentID. For example, signature set of the i^{th} object comprising k-tuples is rep-

resented as s_i = [cid_0, cid_1, ... , cid_k]$_i$. Algorithm 6 depicts this process, which operates in a sequential manner on CPU. We discuss a parallel implementation of this process on GPU in Section 4.8

Algorithm 7 updateSignatureSel(t)

1: $h \leftarrow Hash(t)$
2: $bucketID \leftarrow$ FLOOR($\frac{h}{numBuckets}$)
3: **if** ($bucketID > numBuckets$) **then**
4: $\quad bucketID \leftarrow bucketID \ \% \ numBuckets$
5: **end if**
6: $lineID \leftarrow h \ \% \ sizeBucket$
7: Write h at line $lineID$ in bucket $bucketID$
8: Increment DF
9: **if** ($DF < MAX\_DF$) **then**
10: $\quad$ Update document ID list
11: **end if**

Algorithm 8 buildSignaturesSeq(t)

T: A hash table of size S
1: $h \leftarrow Hash(t)$
2: **if** ($h > S$) **then**
3: $\quad h \leftarrow h \ \% \ S$
4: **end if**
5: Update the DF at address h in the T
6: Update document ID list

4.7.4 Discussion

While designing the Signature Selector data structure, our goal is to enable a memory-efficient implementation. We do not consider linked list-based implementation because such an implementation is not efficient for GPGPUs. Moreover, our design favors efficient memory access while performing operations on Signature Selector data structure. Specifically, our design considers the following aspects: (1) parallelization, (2) scalability, and (3) collision management.

4.7.4.1 Parallelization

Signature Selector data structure is amenable to parallel processing. The task of identifying relevant critical terms, that is, terms that belong to the range (<MIN_DF, MAX_DF>), can be

done in parallel. All the hash buckets can be processed concurrently, thus utilizing the parallel processing capabilities of modern computing systems such as GPGPUs and MICs.

4.7.4.2 Scalability

Our design is scalable in terms of memory footprint. The hash-based candidate selector data structure reduces the size of memory footprint to a factor of $1/k$. Hence, the size of data structure grows sub-linearly in terms of the size of hash table.

4.7.4.3 Collision Management

Note that, the hash tables are subject to collisions. A collision occurs when the hash function produces the same hash value for two given distinct terms. We refer to such type of collision as a true collision. We surmise that such collisions do not have adverse effect on the record linkage process. Collision can also be a result of folding (or aliasing) effect. Folding (or aliasing) occurs when the hash value of a term is outside the range of the hash table used and an MOD of the hash value needs to be used. We refer to such a collision as false collision. Since we use a hash range, which is large enough, we avoid aliasing effects.

4.8 Parallel Algorithms for Signature Selection on GPGPU

We discuss a set of parallel algorithms to extract signatures on GPU. Specifically, we discuss two approaches for extraction of signatures: (1) lock-based approach followed by a reduction to combine the results and (2) a lock-free approach.

4.8.1 Signature Set Selection Using a Lock-Based Approach

While designing a parallel solution to this process, we consider that the following constraint limits the memory footprint to the size of hash buckets used (refer to Figure 4.11) in the worst case. The process of building signature set on a parallel system, as described in Algorithm 8, can be decomposed into two phases: (1) extraction phase, and (2) combine phase. In extraction phase, $ContentIDs$ (i.e., hash value) present in hash buckets are arranged document wise in a parallel manner. For example, one SM extracts $ContentIDs$ from one (or more) hash bucket(s)

and writes into the corresponding list of the documents to which it belongs. This enables task parallelization. However, in this process, multiple SMs (or Blocks) have partial copies of the document list. These multiple copies need to be combined into a single copy in the combine phase. Algorithm 9 lists the lock-based approach. The algorithm operates on multiple hash buckets in parallel. The algorithm operates in two phases described below.

4.8.1.1 Extract Phase

Note that in Algorithm 9, it is likely that more than one thread may be trying to write to update the list of a document simultaneously. Reason being that in a given step two (or more) hash values have same document IDs in the DocIDList. Lines 7 and 8 are the critical region of code and must be protected by appropriate locking mechanisms to avoid race conditions and guarantee correctness of the result.

4.8.1.2 Combine Phase

Note that Signature Selector data structure is processed in parallel by several blocks of threads on a GPU. For example, consider the following assignment: N sets namely, Set1, Set2, ... , Set(N-1) are assigned onto N GPU blocks, BLOCK0, BLOCK1, ... , BLOCK(N-1) of GPU, respectively. This is a one-to-one mapping, that is, one set is assigned to one GPU block. Other mappings are also possible, for example, 2-to-1, 4-to-1, and 8-to-1 configurations wherein multiple sets are assigned to each block.

Each block produces a local version of signature set of all the records. These local versions contain partial signatures. These local versions need to be combined into a single copy using reduction. Two versions are combined into single one by a GPU block at each step. For example, in Stage 1, local versions produced by BLOCK0 and BLOCK1 are combined by BLOCK0. Similarly, versions of BLOCK2 and BLOCK3 are combined by BLOCK2. Finally, the local versions of BLOCK(N-2) and BLOCK(N-1) are combined by BLOCK(N-2). Note that, half of the blocks such as BLOCK1, BLOCK3, ... , BLOCK(N-1) are not assigned any work in Stage 1. The reduction process terminates after log(N) = M stages producing a complete signature set of all the records.

The lock-based approach (discussed above) has drawbacks in the form of locking, prohibitive memory footprint, and finally a reduction phase. We do not pursue this approach. We explore design of an alternative approach for selecting signatures described below.

Algorithm 9 extractSignaturesPar(B)

B: hash bucket of size S
i: Integer

```
1:  if (blockIdx.x < NumBlocks) then
2:    if (threadIdx.x < threadsPerBlock) then
3:      if (hashDF[threadID] > MIN_DF) && (hashDF[threadID] <= MAX_DF) then
4:        if (i < doc_count) then
5:          h ← hash[i]
6:          docid ← hashDocList[i]
7:          CRITIAL SECTION START:
8:          DocList[docid].[DocList[docid].count] ← h
9:          (DocList[docid].count) + +
10:         CRITIAL SECTION END:
11:       end if
12:     end if
13:   end if
14: end if
```

4.8.2 Lock-Free Signature Selection

Algorithm 10 presents an algorithm based on the principle of output data ownership. This is a lock-free algorithm wherein there is no need to protect critical sections via locking mechanism. Basic idea is that the each thread takes ownership of collecting its signatures from all the hash buckets. Note that in that each thread operates on the all hash buckets (Line 2). Lines 3-5 depict the looping constructs. The DF count of a hash value is obtained. in Lines 6 and 7. Lines 8 and 9 determine if the DF of the hash value lies within the specified range. In Line 10, the hash value is obtained from the hash bucket. A list of documents in which this hash value occurred are accessed in Line 11. Line 12 is provisioned for sanity check. In Line 13, the GPU thread verifies if the kth document is in the list belongs to itself or not. In other words, if the document id and the thread id are a match, then the thread will update its list of signatures (Lines 14-17). Same procedure is repeated for the second data set (not shown in the figure in interest of space).

Algorithm 10 extractSignaturesOutput-wise(B)

```
1: if (threadID < S) then
2:   if (hashDF[threadID] > MIN_DF) && (hashDF[threadID] <= MAX_DF) then
3:     for (i=0; i < NUM_HASH_BUCKETS; i++) do
4:       for (j=0; j < HASH_BUCKET_SIZE; j++)  do
5:         for (k=0; k<MAX_DF; k++) do
6:           lineid_hash_bucket=(i×HASH_BUCKET_SZ) + j
7:           DF_count = DF_count1_1d[lineid_hash_bucket]
8:           if (DF_count>=MIN_DF)  then
9:             if (DF_count < MAX_DF)  then
10:              hash_val = Hash_Bucket1_1d[lineid_hash_bucket
11:              t_doc_id = List_DOC_ID1_1d[(i×HASH_BUCKET_SZ×MAX_DF)+(j×MAX_DF)+k]

12:              if (t_doc_id>0) then
13:                if (t_doc_id==((block:Idx.x)×(threadsPerBlock)+threadldx.x)) then
14:                  t_cnt=doc1_list_sel_list_count[(blockldx.x)×(threadsPerBlock)+
                     (threadldx.x)]
15:                  if if (t_cnt¡MAX_CAND) then
16:                    (doc1_list_sel_content_id[((blockldx.x)×(threadsPerBlock)+
                       (threadldx.x))×(MAX_CAND)+t_cnt]=hash_val)
17:                    (doc1_list_sel_list_count[(blockldx.x)×(threadsPerBlock)+
                       (threadldx.x)])++
18:                  end if
19:                end if
20:              end if
21:            end if
22:          end if
23:        end for
24:      end for
25:    end for
26:  end if
27: end if
28: __syncthreads()
29: //Process for second dataset
30: __syncthreads()
```

4.9 Experimental Methodology

4.9.1 Datasets

All data sets used in our experiments are in XML format. The datasets used for duplicate detection experiments were obtained from [2]. The Cora data set referred as *Cora* contains bibliographical information, extracted from citations in scientific publications. The average number of duplicates per object is approximately 9.15. Each object contains the following attributes: author, venue name, volume, and date.

The Country data set referred as *Country* contains 260 objects representing various countries. Each object contains details such as name of country, name of continent, provinces, cities, languages, religions, etc. All the objects are unique in this data set. We replicated 260 objects of initial *Country* data set and added them to original set. We call this new data set as *Country*2*. The *Country*2* dataset now comprises 520 objects and each object as a duplicate.

For record linkage experiments we used Nasa and Swissprot datasets which are also real datasets in XML format. The Nasa datasets contains geographic data, and Swissprot dataset contains bioinformatics data. These data sets are obtained from UW XML database and are described in Table 6.2. From these real data sets, we construct following data sets for the purpose of record linkage experiments: (1) Nasa1K+1K and Nasa2K+2K and (2) Swissprot1K+1K and Swissprot2K+2K. Nomenclature is explained here: Nasa1K+1K indicates that records from Nasa data set with Set1 and Set2 each having 1K records. Similarly, Nasa2K+2K indicates that records from Nasa data set with Set1 and Set2 each having 2K records. Same holds for Swissprot.

4.9.2 Metrics

We measure the effectiveness of algorithms proposed in this chapter using three metrics: precision, recall [16], and F-measures. Precision measures the percentage of correctly identified duplicates over the total set of objects determined as duplicates by the system. Recall measures the percentage of duplicates correctly identified by the system over the total set of duplicate objects.

Table 4.2: Data sets used in duplicate detection experiments.

Data set	Num. candidates	Num. objects	size (in KB)
Cora	1,878	1,693	928.1
Country	260	0	338.0
Country*2	260×2	260	676.0
Restraunt	864	—	69.5

Table 4.3: Data Sets used in record linkage experiments.

Data set	Number of records in (Set1, set2)	size (in KB)
Nasa	1K-2K, 1K-2K	11.5
Swissprot	1K-2K, 1K-2K	12.39

Precision = TP ÷ (TP + FP) = TP ÷ (declared duplicates). Recall = TP ÷ (TP + FN) = TP ÷ (true duplicates). F-measure is defined as harmonic mean of precision and recall. F-measure = (2×recall×precision) ÷ (recall + precision). We measure the execution time elapsed using Linux "time" command.

4.9.3 Experimental Platform

For experiments, we used a CPU/GPGPU based platform. CPU is a Intel (R) Xeon (R) CPU X5650 @ 2.67GHz, a six core machine with 24 GB of RAM and running GNU/Linux 2.6.18. The GPU used is a Tesla C2070 device comprising 448 streaming processors (SPs). All the algorithms are implemented in C (for CPU) and Cuda version 4.1 (for GPGPU). We also used is a Kepler K20 GPU. The K20 comprises 2496 Cuda cores (or SPs) @ 706 MHz and is equipped with 5 GB GDDR5 on-board memory. All the algorithms are implemented in C (for CPU) and Cuda version 7.5 (for GPGPU).

4.10 Experimental Evaluation Results

4.10.1 Accuracy and Efficiency

Figure 4.12 depicts accuracy of the context-aware duplicate detection system on *Cora* data set. The Figure shows that the proposed duplicate detection system achieves precision and recall above eighty percent for $\theta_{string} > 0.5$. For $\theta_{string} > 0.8$, the value of F-measure is more than ninety percent [130].

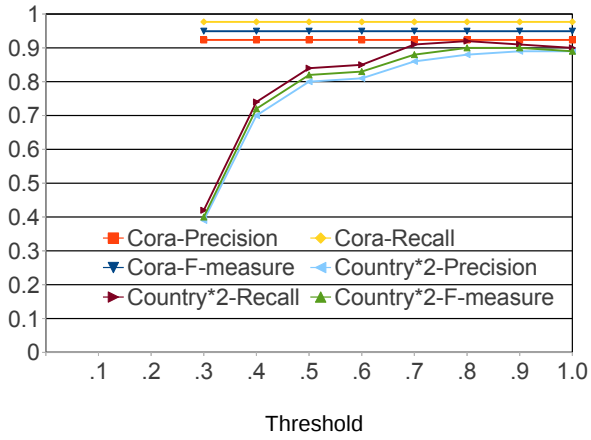


Figure 4.12: Precision, recall, and F-measure obtained for *Cora* and *Country*2*.

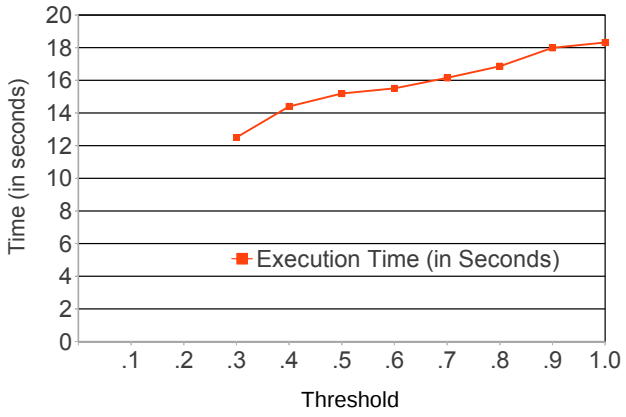


Figure 4.13: Execution time elapsed in duplicate detection over *Cora*.

Figure 4.13 shows time elapsed in duplicate detection over Cora data set for various values of θ_{string}. From this Figure, we observe that for $\theta_{string} > 0.7$, time elapsed is 16 seconds yielding a duplicate detection throughput of 58 KBytes per second (kbps).

We worked with *Country* dataset and verified the accuracy our context-aware duplicate detection system. The detection system detected only nine false positives and missed out some objects (when $\theta_{string} = 0.7$). The false negatives were encountered because some duplicate objects have very small number of attributes when compared to their original counterparts. This set of experiment confirmed that our duplicate detection system performs as desired when there are no duplicates in a given dataset. The duplicate detection throughput obtained is 1690 kbps.

We also worked with *Country*2* data set. The results obtained for *Country*2* data set are also shown in Figure 4.12. From Figure 4.12 we notice that the proposed context-aware detection system achieves precision and recall above ninety two percent and ninety seven percent, respectively for $\theta_{string} > 0.3$. During experiments with *Country*2* data set, we also observed that the context-aware duplicate detection system identifies all but six of the duplicates. And the system identifies twenty one objects as false positives. We observed that most of the false negatives were result of either missing data, or insufficient data i.e. only two or three attributes were present.

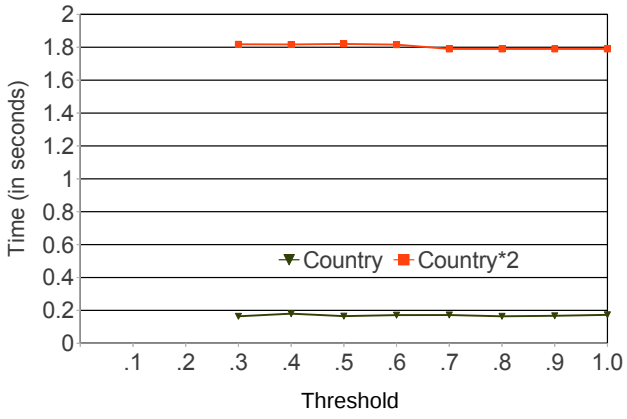


Figure 4.14: Execution time elapsed in duplicate detection over *Country* and *Country*2*.

Table 4.4: Number of Comparisons performed in DF-based ($\theta_{string} = 0.7$) and XMLDup [75] (for best pruning factor (pf).

Data set	DF-based Context aware Dup. Det.	XMLDup[75]
Cora	32,145,032 (Step 1) + 247,681 (Step 2)	60,414,848
Country	257,073 (Step 1) + 140,797 (Step 2)	1,451,553
Country*2	6,712,128 (Step 1) + 133,642 (Step 2)	–

Figure 4.14 shows the time elapsed in detecting duplicates for various values of θ_{string}. From this Figure, we observe that for $\theta_{string} > 0.3$, the execution time elapsed is approximately 1.8 seconds. And the duplicate detection throughput obtained is 375 kbps. Figure 4.14 also shows that the proposed technique yields precision and recall above eighty percent for $\theta_{string} > 0.5$. For $\theta_{string} > 0.8$, the value of F-measure is more than ninety percent.

While experimenting, we observed that quality of the data set used plays a key role in determining the accuracy and efficiency of a duplicate detection system. Typically, the quality of data set used affects accuracy of duplicate detection system adversely. And same is true for our context-aware duplicate detection system which operates in schema-oblivious manner.

4.10.2 Comparison with XMLDup

Here, we compare our duplication detection solution with XMLDup proposal.

4.10.2.1 Execution Time

A recent study presented an improved version of XMLDup duplicate detection system [75]. In [75] experimental platform used is like this: A Ubuntu Linux system running on Intel two core CPU at 2.53 GHz and 4 GB of RAM, and algorithm was implemented in Java, using the DOM API. Their study reports that XMLDup spends minutes and 4 seconds (refer [75](Table 4)) in detecting duplicates over *Cora* data set. The study also reports that XMLDup takes ∼9 seconds for detecting duplicates over *Country* data set (refer [75](Table 2)).

When compared with the state-of-the-art in duplicate detection in semi-structured data, our context-aware duplicate detection results in approximately 8X speedup. We use DF-based

heuristic (described in Section 3). We discount the performance difference between java and C languages) over the XMLDup [75]. We sacrifice an accuracy of less than eight percent. The loss in accuracy is primarily because we use signature set for identifying the candidate set. Note that signature set comprises only a small number of nodes (and attributes) from the tree object. As a result, we miss out some true candidates (which are potential duplicates) when forming candidate set leading to a slight loss in accuracy.

4.10.2.2 Reduction in Comparison Operations

Table 4.4 lists the number of string comparison operations performed for duplicate detection when using our DF-based heuristic and XMLDup. From this Table we observe that for *Cora* data set, our context-aware duplicate detection system when using DF-based heuristic needs less than 33 million string comparisons. The XMLDup requires over 60 million string comparisons [75](Table 5, ROW: "best pf"). For *Country* data set, our DF-heuristic based system requires less than 0.4 million comparisons whereas the XMLDup requires a little over 1.4 million string comparisons. For *Country*2* data set, our DF-heuristic based system requires about 7 million string comparisons.

4.10.3 Effect of Data Shaping

We discuss the speedup obtained in duplicate detection throughput as a result of data shaping. We compare the execution time of the CompareTreeSeq Algorithm obtained on CPU and GPU for various scenarios ranging from best case to worst case.

Table 4.5 lists the details of the execution time obtained for all the five scenarios. The best case result indicates an empirical lower bound on time elapsed on detecting duplicates. Similarly, the worst case gives an empirical upper bound on the time elapsed in detection process. Also, note that the execution time we report is the time elapsed in performing comparisons of quadratic order. From Table 4.5, we observe that data shaping improves the detection throughput over state-of-the-art, i.e., XMLDup by approximately two orders of magnitude.

Figure 4.15 shows the effect of data shaping on speedup obtained in duplicate detection throughput. The data shaping results in a speed up of up to two orders of magnitude for

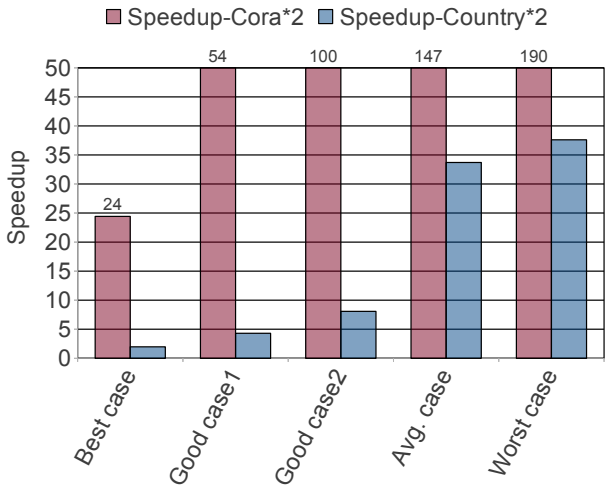


Figure 4.15: Speedup in execution time elapsed for Cora*2 and Country*2 data sets. Speedup is ratio of Execution time on CPU (CPUTime) vs. Execution time on GPU (GPUTime).

Table 4.5: Execution Time elapsed on CPU and GPU for Cora*2 and Country*2 data sets. ($R = RefTreeSize$; $Iter = Iterations$).

Case	**Iter**	**Cora*2 CPUTime**	**Cora*2 GPUTime**	**Country*2 CPUTime**	**Country*2 GPUTime**
Best case	1	0m:14.14s	0m:0.58s	0m:0.89s	0m:0.45s
Good case 1	4	0m:35.55s	0m:0.66s	0m:2.04s	0m:0.47s
Good case 2	10	1m:18.18s	0m:0.78s	0m:4.30s	0m:0.53s
Avg. case	R/2	2m:34.21s	0m:1.05s	1m:15.99s	0m:2.26s
Worst case	R	5m:07.67s	0m:1.62s	2m:31.90s	0m:4.04s

*Cora*2* and *Country*2*. The best case scenario occurs when expecting exact duplicates instead of approximate (fuzzy) ones, or while detecting duplicates in a time constraint environment.

4.10.4 Record Linkage Results

We study the following: performance of hash-based approach for data encoding, effect of the size of hash table on accuracy of linking records, performance of signature selec- tor-based data pruning, tradeoff between the F-measure, and metric of overall accuracy, versus memory requirement of the hash-based methods, overall impact of using GPU implementation of our hash-based approach. Now, we discuss the results obtained.

4.10.4.1 Performance of Hash-Based Approach for Data Encoding

Figure 4.16 depicts the implication of using a hash-based approach for assigning numeric codes to string terms. X-axis is the number of words whose hash value is calculated. Y-axis is the time elapsed in hashing step. The figure shows a linear relationship between the elapsed time in hashing and the number of words hashed as expected.

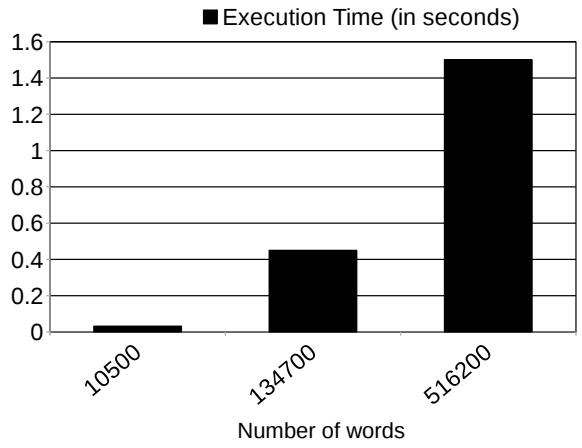


Figure 4.16: Runtime of hash functions.

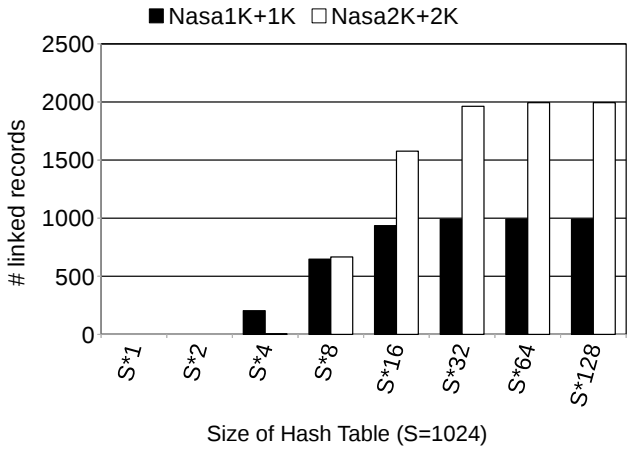


Figure 4.17: Effect of the size of hash table on accuracy for Nasa data set.

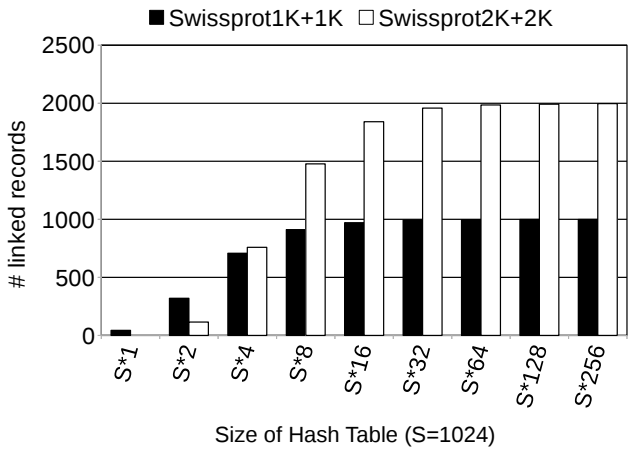


Figure 4.18: Effect of the size of hash table on accuracy for Swissprot data set.

4.10.4.2 Effect of Hash Table Size

Figure 4.17 depicts the effect of varying the size of hash table used in the record linkage process for Nasa. X-axis shows the size of hash table used in multiples of 1K (1024). Y-axis shows the total number of records identified. The figure shows that hash table of 64K is sufficient for finding the linked records for data sets comprising 1K+1K records. The figure also shows that hash table of 64K entries is sufficient for finding the linked records for data sets comprising 2K+2K records.

Figure 4.18 depicts the effect of varying the size of hash table used in the record linkage process for Swissprot. X-axis shows the size of hash table used in multiples of 1K (1024). Y-axis shows the total number of records identified. The figure shows that hash table of 128K is sufficient for finding the linked records for data sets comprising 1K+1K records [129]. The figure also shows that hash table of 256K is sufficient for finding the linked records for data sets comprising 2K+2K records.

4.10.4.3 Performance of Signature Selector-Based Data Pruning

Now, we discuss effect of varying the number of hash buckets on the accuracy in linking records. Figure 6.8 plots the effect of the number of hash buckets on the accuracy of record linkage for Nasa1K+1K data set. X-axis shows the number of hash buckets each of size 64

entries varying from 32 to 2048 sizes. Y-axis shows the total number of records identified. From this figure, we observe that a 128 hash buckets each of size 64 yield an acceptable accuracy for the record linkage problem. With 128 hash buckets, we observe an insignificant number of false positives, 0.81%. Increasing the number of buckets to 256 or more yields approximately similar performance. Figure 4.19 also depicts the effect of varying the number of hash buckets for Nasa2K+2K data set. From this figure, we observe that a 256 hash buckets each of size 64 yield an acceptable accuracy for the record linkage problem. Using 256 hash buckets yields 1.61% false positives, which is insignificant in the context.

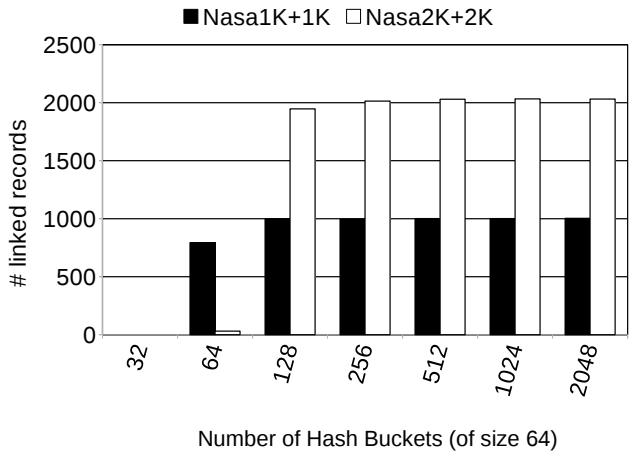


Figure 4.19: Effect of the number of hash buckets on the accuracy of record linkage for Nasa data set.

4.10.4.4 F-Measure versus Memory Requirement

Figure 4.20 compares the effect of memory needs on the accuracy of baseline approach and our signature selector data structure. The figure depicts the effect of memory usage on F-measure. X-axis shows the number of memory requirement in terms of entries in hash table of signature selection data structure varying from 1024 to 1024 256. Y-axis shows the F-measure obtained. Note that F-measure is the harmonic mean of precision and recall metrics.

Figure 4.20, we note that for Nasa1K+1K data set, the baseline hash-based approach (Section 4.7.1) requires a hash table of size 32K for an acceptable accuracy level (i.e., for an accuracy

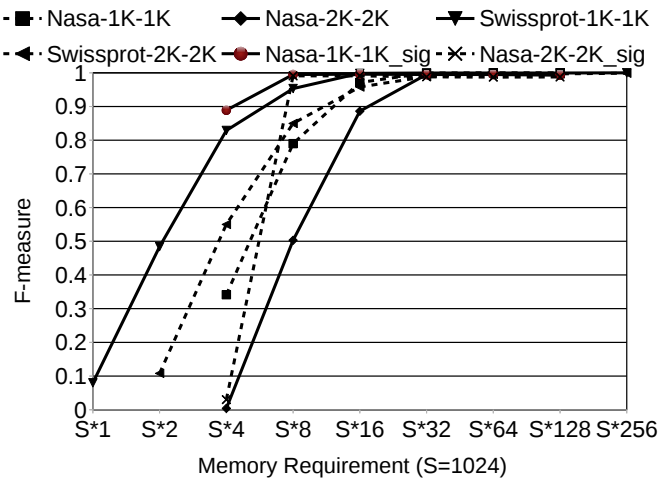


Figure 4.20: Effect of the size of hash table on the accuracy for Nasa data set.

level more than 99 percent). Our optimized signature selector-based approach (Section 4.7.2) requires 128 hash buckets of 64 entries (equivalent to 1024 × 8 entries), each are sufficient to keep false-positive rate within an acceptable limit. This indicates that the use of Signature Selector data structure results in a reduction of 4X in memory requirement at the preprocessing stage.

Similarly, for Nasa2K+2K data set, the baseline approach, hash-based approach, requires a hash table of size 32K, and our optimized Signature Selector-based approach requires 128 hash buckets (of 64 entries each) for an acceptable accuracy level. This indicates that the use of Signature Selector data structure reduces memory requirement of preprocessing stage by 4X.

4.10.4.5 Overall Performance Comparison

A table-based implementation to identify signature sets in the context of duplication detection problem is discussed in [130]. Here, we discuss the overall performances of record linkage implementation utilizing the table-based implementation for CPU against our novel hashing-based implementation for GPU, which leverages lock-free signature selection algorithm.

Figure 4.21 depicts a comparison of overall execution time of a CPU implementation of a table-based record linkage method based on [130], depicted as Baseline_ CPU, versus the hashing-based method on GPU, depicted as Hash_GPU. X-axis shows the type of data set used

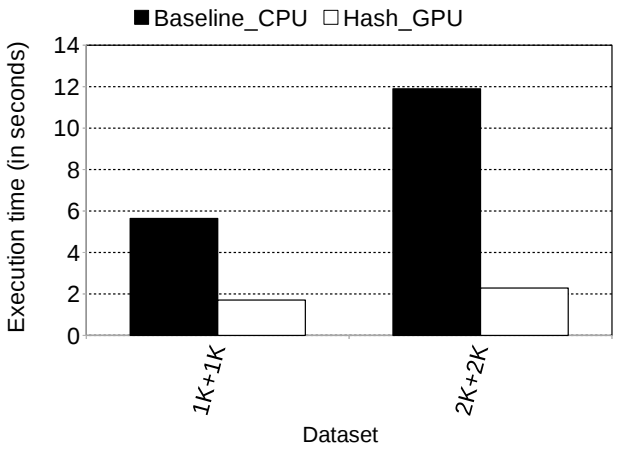


Figure 4.21: Performance of baseline record linkage method on CPU (depicted as Baseline_CPU) vs. hash-based method on GPU (depicted as Hash_GPU).

in measurement, specifically, the Nasa1K+1K and Nasa2K+2K data sets. Y-axis shows the end-to-end time, in seconds, elapsed in execution of the algorithms on the respective platforms. For GPU implementation, we used the following configuration parameters: threadsPerBlock = 128, HASH_BUCKET_SZ = 64, and NUM_HASH_BUCKETS = 128.

Figure 4.21, we note that for Nasa1K+1K data set, table-based approach (presented in [130]) takes, approximately, 5.64 seconds. Our hash-based approach (Section 4.7.2) requires, approximately, 1.707 seconds to find the linked records. This indicates that the use of Signature Selector data structure when implemented using lock-free approach results in a speedup of $\sim$ 3.3X.

Similarly, for Nasa2K+2K data set, the table-based approach (Section 6.4) takes, approximately, 11.898 seconds. Our hash-based method approach (Section 6.2) requires, approximately, 2.286 seconds to produce the linked records, resulting in a speedup of $\sim$ 5.2X. Note that despite the increase in speedup from 3.3X to 5.2X, the record linkage workload has been doubled. This indicates that our hash-based algorithms designed for GPUs are scalable.

4.11 Conclusion

State-of-the-art in duplicate detection in semi-structured data obtains significant improvement by exploiting the schema-related knowledge. Such schema-bound duplicate detection approach, however, has severe limitations when dealing with multi-sourced, heterogeneous, high velocity data streams.

In this chapter, we developed a novel context-aware duplicate detection system which is workload- and complexity-aware, and adaptable to the underlying computing platform. The proposed system operates in schema-oblivious manner, and relies upon information theory based heuristic and data shaping technique for efficient and scalable duplicate detection in multi-sourced, heterogeneous data sets. Experiments with real world data sets show speed up of up to 8X over state-of-the-art, while sacrificing accuracy of less than eight percent.

In addition, our data shaping technique for GPGPU processing speeds up the duplicate detection throughput by up to two orders of magnitude.

CHAPTER 5. TREE MATCHING USING DATA SHAPING ON PARALLEL HARDWARE

Summary. Real time big data analytics has become important to meet the business as well as other decision making needs in many complex applications. A significant portion of such data is available and stored in semi-structured form. A tree-based organization is commonly used in such cases. Tree matching is a core component for many applications such as fraud detection, spam filtering, information visualization and extraction, user authentication, natural language processing, XML databases, bioinformatics, etc. Comparing ordered (unordered) trees is compute-intensive, in particular for Big Data. To facilitate comparison of ordered trees, in this chapter, we address the problem of shaping the semi-structured data to enable time efficient processing on contemporary hardware such as a GPGPU (General Purpose Graphics Processing Unit) and INTEL MIC (a multi-core processors). Specifically, our data shaping approach enables pre-computation of partial edit distance values in parallel. We evaluate our work using real world data sets. Our experimental results show that our SIMT-based PTED-GPU (Parallel Tree Edit Distance using GPU) implementation shows speedup of up to 12X when compared to the state-of-the-art in tree edit distance (TED) computation.

Keywords: Big Data; Data Analytics; Tree Matching; Tree Edit Distance; Parallel Processing; GPGPU; Data Shaping

5.1 Introduction and Motivation

Data deluge or Big Data phenomenon has led to an ever increasing volumes of data of various forms such as structured, semi-structured, and unstructured. A compelling dimension of Big Data is velocity. Rapid and timely analytics on Big Data offers significant advantage to

business and scientific communities. A study predicts that in future, organizations have to deal with more and more unstructured and semi-structured data generated from sensors and devices [29]. The same study also predicts that in future a majority of workload in organizations will comprise the real-time analytics.

A significant portion of such data is available and stored in semi-structured form. A tree-based organization is commonly used in such cases. Several data analytic pipelines involve tree comparison operations as their core components. For example, tree comparison between ordered (or unordered) trees, finding the edit distance between a given pair of trees (i.e. tree edit distance), etc. Finding tree edit distance in near-real-time is critical for various applications such as fraud detection [154], spam filtering [97], and others. Fast tree edit distance solution is key to tasks such as information visualization [8], [9], mining web applications [18], automatic news extraction [121], and visual password based authentication [104]. Tree edit distance is also used in recognizing textual entailment between syntactic trees of texts [67], [11] and sentence ranking [157] for the purpose of question answering.

The tree comparison problem has quadratic complexity in terms of the number of nodes in the given trees. Therefore, the structure of tree structured data presents processing time challenges. As volume of semi-structured data sets (e.g. XML, JSON, etc.) increases and real-time data analytic over such data sets become indispensable, there is a need to leverage parallel processing platforms effectively. For example, we can accelerate the tree matching component using parallel hardware such as a General Purpose Graphics Processing Units (GPUs) effectively.

General purpose graphics processing units (GPUs) have become mainstay for high performance computing. Accelerators such as GPUs, not only provide thousands of parallel cores and special purpose software managed memories, they also offer significant cost and energy advantages over their multi-core CPU counterparts if the computing needs match the architecture of GPU. For example, when compared to latest quad-core CPUs, Tesla GPUs (C2050/C2070) deliver a matching supercomputing performance at 1/10th of the cost and 1/20th of the power consumption [101]. Five out of top ten supercomputers from Top500 listing are powered by accelerators such as Intel Xeon Phi or NVIDIA K20/40 [3]. For example, Tianhe-2 (NSCC,

China), listed on the top, comprises Intel Xeon Phi and Titan (ORNL, TN, USA), number two in the list, comprises NVIDIA K20 accelerators. We at ISU have a large number of nodes including both of these platforms in addition for multicore nodes. It is important to explore applicability of SIMT machines for not only to meet the high performance computing needs but also from the cost and energy efficiency point of view.

In context of tree matching problem using GPU, we observe that this problem is compute- as well as data-intensive. Finding tree edit distance between two trees has a time complexity of $O(n^3)$. Given that GPUs are rich in compute resources, they can be good fit to process tree matching problem faster. However, the compute units of GPUs must be presented data for processing in a timely and appropriate manner to gain any significant speedup. GPUs also have memory hierarchy limitations. The compute and memory resources of a GPU need to be used effectively. Moreover, substructures (e.g. subtrees) contained within the semi-structured data object are critical for parallelizing computation. These substructures can be used to perform partial computations independently (on any type of hardware). Partial computations using these substructure can be used to derive the overall outcome. Thus, it is important to identify the substructure(s) contained in semi-structured data objects and reorganize the tree structured data appropriately.

In this chapter, we address the problem of shaping the semi-structured data to enable time efficient processing on a SIMT hardware. Specifically, we propose a novel data shaping technique to identify relevant substructures from a tree object. We also propose a novel approach to pre-compute the partial edit distance between trees using the substructures identified. The computations involving substructures offer a high level of parallelism and are independent of each other, and can be realized on a parallel hardware. Our data shaping technique allows a high degree of *data reuse* using the available on-chip memories effectively. This reduces the traffic to off-chip memory. Our low level approach is applicable to the data structure used in the existing algorithms. We develop a general framework to speedup the data analytic pipeline in the context of semi-structured datasets. Our framework is applicable to several scenarios such as acceleration of key kernels of tree edit distance based algorithms and to calculate approximate distance between two trees, etc.

Contributions This chapter makes following contributions:

- A novel data shaping algorithm to identify substructures (subtrees) within a tree which enables parallel processing of trees.
- A novel approach to pre-compute the partial edit distance between trees, using the substructures identified, in parallel.
- A technique for computing tree edit distance on parallel hardware.

The rest of the chapter is organized as follows: In Section 5.2, we discuss the background for the work and challenges involved. Next, we discuss our approach towards developing an efficient solution. Section 5.3 describes the tree matching process by exploiting substructure and efficient computations on GPU. Section 5.4 presents optimizations. Section 5.5 and 5.6 present evaluation methodology adopted and the results obtained, respectively. Related work is discussed in Section 5.7. Sections 5.8 and **??** provide our conclusions and future directions, respectively.

5.2 Background and Challenges

5.2.1 Background

We first provide an overview of the tree edit distance work and GPU and introduce some of the terminology used in the context of GPU. For more details see [103].

5.2.1.1 Tree Edit Distance

Tree edit distance (TED) between two trees is described in terms of the minimum work needed to transform one tree into another tree by applying a set of edit operations. These edit operations are deletion, insertion, and substitution [20]. Figure 5.1 shows an example of tree edit distance using three trees S0, S1, and S2. Consider Trees S0 and S1. Deletion of node "v5" from S1 results in S0. Hence, the tree edit distance between S0 and S1 is 1. Similarly, tree edit distance between S1 and S2 is 2. The reason being that the two operations on S1,

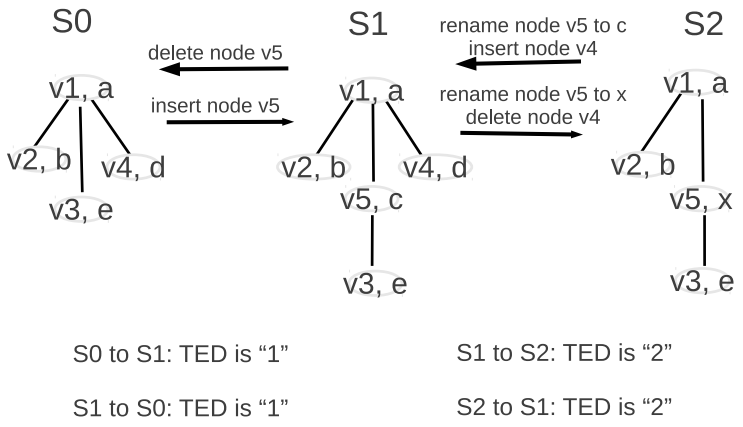


Figure 5.1: Edit operations and tree edit distance (TED).

namely (1) substitution (renaming) of node “v5” to “c”, and (2) deletion of node “v4” results in S2.

Zhang and Sasha proposed a recursion based algorithm for calculating tree edit distance between two ordered trees [160]. Their algorithm recurs on the rightmost root and operates in two stages: (1) The first stage comprises (a) naming the nodes in post order, (b) identifying keyroots, and (c) identifying leftmost descendants. (2) The second stage calculates the edit distance between the subtrees at keyroots and updates the tree edit distance matrix. At the end of this stage, edit distance between two trees is obtained. For details, refer to [160]. Number of relevant subproblems or the time complexity involved is $O(n^2 \times m^2)$ (which can be up to $O(n^4)$ when $m \approx n$) where “m” and “n” are the number of nodes in Trees F and G, respectively.

Klein et. al proposed recursion on a light child. This modification reduces the time complexity to $O(n^3 \times \log(n))$.

Demiane et. al improved upon the time complexity of tree edit distance computation [33]. They propose to consider the size of input trees. Their basic idea is to compute d(F, G) recursively on smaller subtrees and use Klein’s strategy recursively. The time complexity of algorithm by Demaine et. al is $O((n \times m)^{3/2}) = O(n^3)$. A survey on tree edit distance and related problems is provided in [20].

A robust tree edit distance (RTED) algorithm is presented in [111]. Authors introduce a class of LRH (left-right-heavy) algorithms which includes RTED and the fastest tree edit distance algorithm presented in the literature. Basic idea behind RTED is the dynamic composition strategy which recursively decomposes the input trees into forests by removing nodes. State-of-the-art in tree edit distance for ordered tree has a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$, where "n" is the maximum of the number of nodes in the two input trees.

5.2.1.2 GPU

GPGPUs are highly parallel architecture SIMT machines comprising thousands of cores or streaming processors (SPs). For details refer Section 2.3.1. In this chapter, we refer GPGPUs and Single Instruction Multiple Thread (SIMT) interchangeably.

5.2.2 Challenges

As noted earlier, tree matching problem is compute intensive involving quadratic order pairwise comparison operations. Parallel machines like GPU offer tremendous processing power. High computational complexity of the problem motivates us to use highly parallel machines like General Purpose Graphics Processing Unit (GPU) for such compute intensive problems.

A key observation here is that the data items are required to be present close to the cores (streaming processors) working on those data items. If this can be achieved then additional computations can be launched for almost free, leading to increase in the data reuse which results in a higher Compute to Global Memory Access (CGMA). The required memory bandwidth and system wide data movement are further constrained by energy consumption and power dissipation limits. Reduced system wide data movement allows increased power allocation to computation cycles. However, increasing data reuse factor necessitates reconsideration of the algorithm to be used. GPUs possess several architectural characteristics, listed below, features which make tree processing challenging.

(1) Lock-step execution of the streaming processors (SPs) in streaming multiprocessor (SM).

(2) High memory access latency associated with device memory (or main memory on a GPU).

(3) Limitations of memory hierarchy of the GPUs, for e.g., data access constraints associated with constant memory.

To work with these architectural features effectively several issues need to be addressed that include:

(1) Size of the computation executed in parallel,

(2) Size of the data item to be processed on parallel, and

(3) Pattern and order of accessing data.

5.3 Tree Matching on Parallel Hardware

Our solution to the tree matching problem operates in three stages.

(A) Subtree Identification: This stage results into a collection of subtrees. This is a data shaping stage.

(B) Computation of partial distances: Compute partial distance values using Subtrees. Also, compute distance between leaf nodes. Record these pre-computed values in TreeDistance Matrix. We use parallel processing to speed up the computation process.

(C) Compute final Tree Edit Distance (TED) using the precomputed edit distance values (from Stage 2). We describe these stages in detail in the following.

5.3.1 Subtree Identification

The basic idea for subtree identification is to organize tree structured data in order to facilitate efficient computations on the GPU and to create opportunities for parallelism and exploiting memory hierarchy. After organizing (shaping) tree data into subtrees (substructures) we can use these substructures for matching a pair of trees. Data shaping also helps exploit the compute and memory resources of the underlying GPU hardware effectively. Subtree Identification Stage is executed on CPU.

Subtree Identification Algorithm We decompose a given tree into subtrees using a subtree encoding algorithm as shown in Algorithm 11. Figure 5.2(a) depicts Tree T1. Definitions of "Left[]" and "Keyroot[]" are similar to prior works [160], [33], etc. Left[i] is the

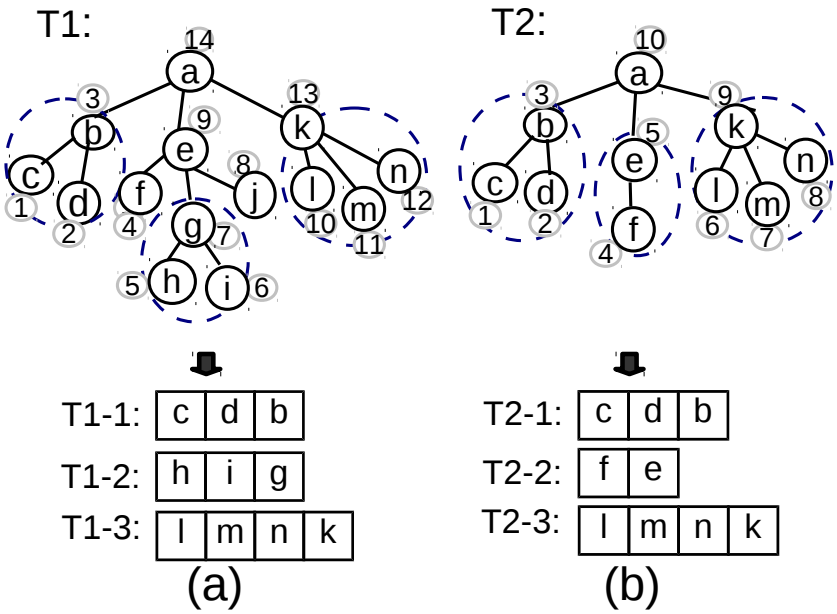


Figure 5.2: (a) Tree T1 and its three subtrees. (b) Tree T2 and its three subtrees. Nodes are recorded in post-order traversal number which are shown in gray circles.

post order traversal number of leftmost leaf descendant of the subtree rooted at index "i". A Keyroot is a node of a given Tree that either has a left sibling or is the root of T. Some nodes in T1 are enclosed in dotted ellipse. These are set of nodes identified by Algorithm 11 as subtrees, namely T1-1, T1-2, and T1-3. Figure 5.2(b) shows Tree T2 and its three subtrees T2-1, T2-2, and T2-3. Next, we explain the subtree encoding process.

Algorithm 11 Identify Subtree

Left[k]: Leftmost leaf descendant of the subtree rooted node at k.
Keyroot[ ]: Array of keyroot nodes.
N: Node currently being visited.

1: Visit Keyroot nodes one by one
2: **for** every N **do**
3: **if** $((depth[N] - depth[left[N]]) == 1)$ **then**
4: Include nodes left[N]...N as a subtree
5: **end if**
6: Mark N as a visited node
7: **end for**

Identification of Subtrees Consider a post-order traversal of Tree T1 as shown in Figure 5.2(a). Apply Subtree Encoding Algorithm to T1. We obtain three subtrees. Nodes *(c, d, b)* constitute first subtree T1-1. Nodes *(h, i, g)* are part of second subtree, i.e. T1-2. Nodes *(l, m, n, k)* form the third subtree T1-3. For Tree T2 (shown in Figure 5.2(b)) subtree identified by tree encoding algorithm are as follows: We obtain three subtrees. Nodes *(c, d, b)* consitute first subtree T2-1. Nodes *(f, e)* are part of subtree T2-2, and nodes *(l, m, n, k)* constitute subtree T2-3.

Memory Requirement Next, we analyze the space complexity of data structures used in Identify Subtree Algorithm. The Subtree Encoding Algorithm stores every node of Tree T in NodeList exactly once. Hence, space complexity of NodeList is $O(n)$. Total size of metadata associated with subtrees (such as start and end indices of subtrees) depends upon the number of subtrees identified. Note that total number of subtrees possible is $\leq O(n)$. Hence, the overall space complexity of Subtree Encoding Algorithm is $O(n)$.

5.3.2 Computing Partial Tree Edit Distance Values using Subtrees

We compare every subtree of T1 with every subtree of T2 and compute the edit distance between them. All of these subtree comparison operations are independent of each other and can be realized in parallel. We use massively parallel compute resources of SIMT machine to compute the edit distance. We record the pre-computed values of subtree edit distances in matrix M[][]. Subtree edit distance is basically the ordered string edit distance, namely Levenshtein Distance. The cost matrix C[][] is also calculated by the GPU. Listing 5.1 depicts the GPU Kernel used to realize this computation. Block "i" obtains the start index of "i"th subtree from a redirection array sub1a[] (see Line 5). Similarly, Thread "j" obtains the start index of "j"th subtree from a redirection array sub2a[] (see Line 8). Start indices of subtrees of T1 and T2 are maintained in array sub1[] and sub2[], respectively. Depending upon the nature of subtrees (i.e. single node/multi node), Threads update the matrix M. We describe this process shortly.

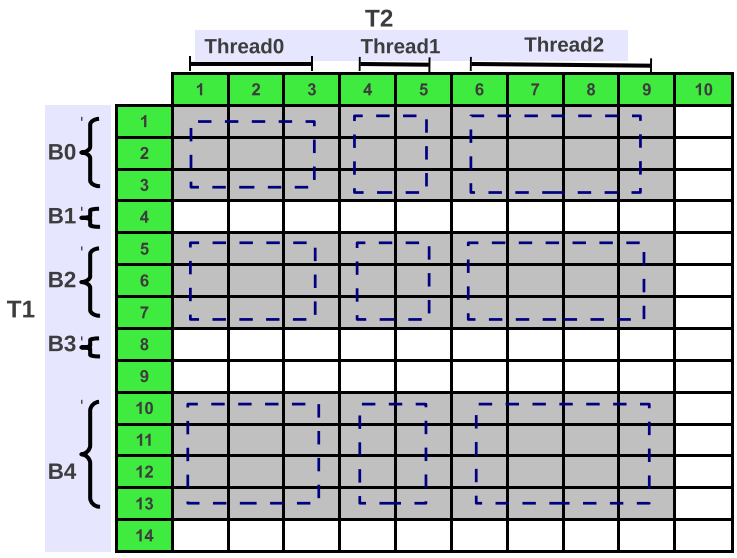


Figure 5.3: Pre-computation of edit distance values using subtrees.

Edit Distance Matrix Figure 5.3 depicts the edit distance matrix, M[][] mentioned in Listing 5.1. M is a 14 × 10 matrix. The number of rows and columns depends on the number of nodes in T1 and T2 i.e. T1 has 14 nodes and T2 has 10 nodes, therefore M is 14 × 10. First row and first column marked in green are indices. Note that some cells in M[][] are marked as gray. These cells correspond to the entries which are calculated using the subtrees identified in Stage A (refer Section 5.3.2). For example, the values in cells M[1:3][1:3] correspond to T1-1 and T2-1. This set of cells are indicated by a top-left box marked by dotted lines. Similarly, values in cells M[1:3][4:5] correspond to T1-1 and T2-2 marked by top-middle box, and values in cells M[1:3][6:9] marked by top-right box correspond to T1-1 and T2-3.

Figure 5.3 also depicts the assignment of computations using subtrees to various Blocks and Threads of GPU. The computation of edit distance values between T1-1 and T2-1 is assigned to Thread0 of Block B0. Computation for T1-1 and T2-2 pair is assigned to Thread1 of Block B0 and computation of T1-1 and T2-3 pair is assigned to Thread2 of Block B0. Similarly, Blocks B2 and B4 are assigned computations involving subtrees T1-2 and T1-3, respectively. In Figure 5.3, also note that Block B1 is assigned edit distance computation between node "f" (of T1) and subtrees (nodes) of T2. Similarly, Block B3 is assigned the distance computation between node "j" (of T1) and subtrees (nodes) of T2.

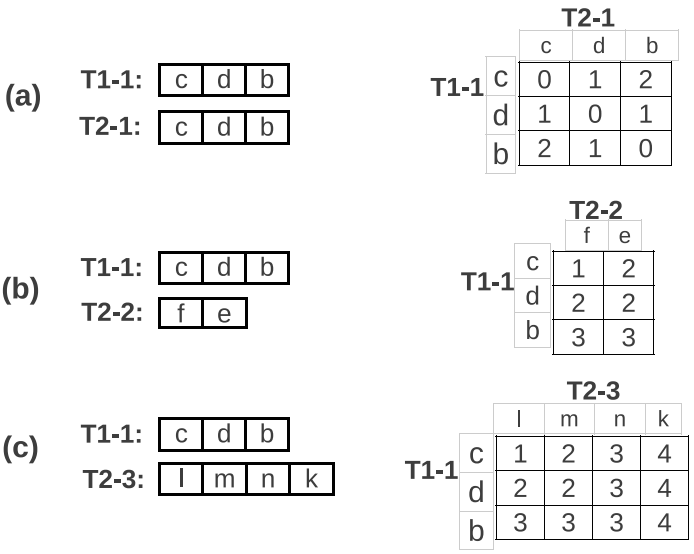


Figure 5.4: Computation of subtree edit distance values. (a) Shows computation of edit distance values between T1-1 and T2-1. (b) and (c) Shows the computation for T1-1 and T2-2 pair, and T1-1 and T2-3 pair, respectively.

Edit Distance Matrix Computations We explain the calculation of edit distance matrix, M[][] using three subtrees of T1 and three subtrees of T2 for comparison. Note that subtree edit distance between pairs T1-1 and T2-1, T1-2 and T2-2, T1-3 and T2-3 is zero. The reason being that the subtrees in these pairs are identical (refer to Figure 5.2(a)-(b)). Edit distance between T1-2 and T2-2 is obtained by replacing first two nodes of T1-2 i.e. nodes *(a, b)* by nodes *(e, f)* of T2-2 and then deleting the remaining nodes of T1-2 i.e. nodes *(c, d)*. Other entries of matrix M can be obtained in the similar way. Figure 5.4 shows the subtree edit distance computation process for T1-1 and three subtree of T2 in Figure 5.4(a)-(c). Figure 5.5 depicts the partially computed M[][].

Listing 5.1: Compute Partial Distance on GPU

```
1  __global__ void ComputePartialDistGPU(...)
2  { //variable declarations ...
3   if(blockIdx.x < m)
4   if(sub1[sub1a[blockIdx.x+1]]>0)
5   { i=sub1a[blockIdx.x+1];
6     if(threadIdx.x < n)
7     if(sub2[subt2a[threadIdx.x+1]]>0)
```

```
8     {j=sub2a[threadIdx.x+1];
9      if(i == sub1[i])//t1 is a single node subtree
10     //Find distance between t1 and t2, Update M.
11     else //t1 is a multinode subtree
12     {if(j == sub2[j])//t2 is a single node subtree
13      //Find distance between t1 and t2,Update M.
14      else //t2 is also a multinode subtree
15      {if((left1[i-1]==left1[sub1[i-1]])
16         &&(left2[j-1]==left2[sub2[j-1]]))
17       {//Case 2: Both are tree
18        M[(i*SZ)+j]=dist2(word1,len1,word2,len2);
19       }
20       else
21       {if((left1[i-1]!=left1[sub1[i-1]])
22            &&(left2[j-1]!=left2[sub2[j-1]]))
23         {//Case 3: One is not a tree
24           M[(i*SZ)+j]=dist3(word1,len1,word2,len2,
25                    sub1[i]-1,sub2[j]-1,SZ,M);
26         }
27       }}}
28      }//End of Thread
29     }//End of Block
30  return;
31 }//end of kernel
```

Listing 5.2: Computation of Final Distance

```
1 TED() {
2   int i1, j1; int i, j;
3   for(i1=1;i1<=total_key_roots1;i1++){
4     for(j1=1;j1<=total_key_roots2;j1++){
5       i = key_roots1[i1]; j = key_roots2[j1];
6       //Check if Treedist value for this //pair of keyroots already exist.
7       if ( M[i][j] > 0 ) { continue; }
8       else {Compute_Fdist(i,j,left1,left2,C);}
9 }
```

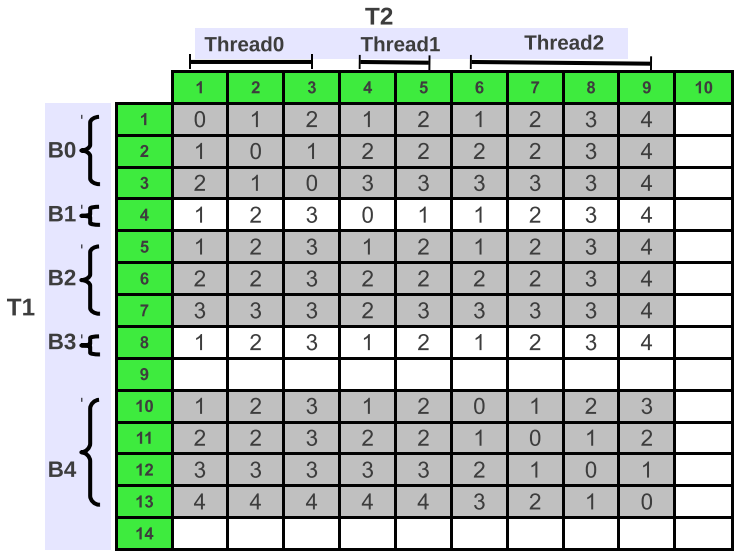


Figure 5.5: Updated state of M[][].

5.3.3 Final Tree Edit Distance

The final tree edit distance calculated using dynamic programming model similar to the one mentioned in earlier works [160], [33], [111]. The subroutines TED() and ComputeFdist() of Listing 5.2 depicts the dynamic programming formulation. We differ from previous studies (namely [160], [33], [111]) in one key aspect – we do not have to compute all the values, since the matrix M is partially filled. This is due to the precomputed values from previous stage (refer Section 5.3.2). Line 8 of subroutine TED() (Listing 5.2) supports this by an "if" condition. For computing distance between two forests, we use the method proposed in prior works [160] and is shown as Listing 5.3. This Stage is executed on CPU.

Listing 5.3: Computation of Forest Distance [160]

```
1  Compute_Fdist(int pos1,int pos2,
2          int left1[],int left2[],int C[][])
3  {int b1 = pos1 - left1[pos1] + 2;
4   int b2 = pos2 - left2[pos2] + 2;
5   int fdist[b1][b2]; int i,j,k,l,m,n;
6
7   for(i=0;i<b1;i++){for(j=0;j<b2;j++){fdist[i][j]=0;}}
8   fdist[0][0] = 0;
```

```
9   for(i=1;i<b1;i++)fdist[i][0]=fdist[i-1][0]+C[i][0];
10  for(i=1;i<b2;i++)fdist[0][i]=fdist[0][i-1]+C[0][i];
11
12  for(k=left1[pos1],i=1;k<=pos1;k++,i++){
13  for(l=left2[pos2],j=1;l<=pos2;l++,j++){
14  //if both are trees, then:
15  if((left1[k]==left1[pos1])
16          &&(left2[l]==left2[pos2]))
17   {fdist[i][j]=MIN(fdist[i-1][j]+C[0][l],
18        fdist[i][j-1]+C[k][0],fdist[i-1][j-1]+C[k][l]);
19       M[k][l] = fdist[i][j];
20   }
21   else{
22    m = left1[k] - left1[pos1];
23    n = left2[l] - left2[pos2];
24    fdist[i][j]=MIN(fdist[i-1][j]+C[0][l],
25          fdist[i][j-1]+C[k][0],fdist[m][n]+M[k][l]);
26  }}}
27  return;
28 }
```

5.4 Optimization

In this section, we discuss optimizations to the algorithms presented in previous: (1) Parallelization of Final Edit Distance Computation, and (2) Parallelization of Forest Distance Computation.

5.4.1 Parallelization of Partial Tree Edit Distance on CPU

For partial tree edit distance computation refer to Listing 5.1. Note that the computations are independent and hence amenable to parallel processing on CPU cores. For example, iterations on outer loop can can be chunked and assigned to OpenMP thread such that each OpenMP thread runs on a single core of the 16-core or 32-core CPU.

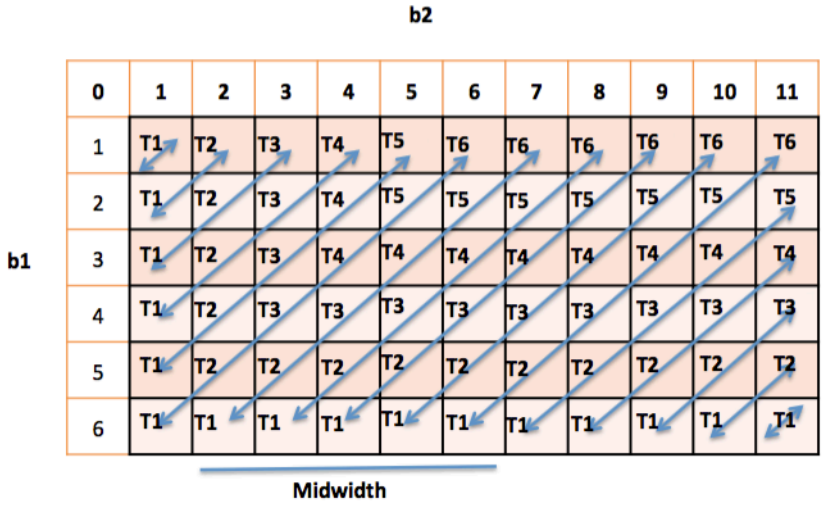


Figure 5.6: Parallelization of forest distance computation: Mapping of computations to threads.

5.4.2 Parallelization of Forest Distance Computation

Refer to Lisiting 5.3 for forest distance computations. We parallelize the forest distance computations using wavefront-based approach.

Figure 5.6 depicts the parallelization strategy. Basic idea is that computations across the diagonal are independent and hence the computations in diagonal set can be executed in parallel. For example, the double edged arrows represent the set of computations which are executed in parallel. This process moves from left to right. To facilitate computations, we need to calculate a set of indices. This indices calculation is an overhead.

5.5 Experimental Setup

5.5.1 Data sets and Platform

We use three popular xml data sets in our experiments namely nasa, SwissProt, and tree_bank_e [141]. "Nasa" is an astronomical dataset having average depth and maximum depth of trees as 5.5 and 8, respectively. "Swissprot", a curated protein sequence database, has average depth and the maximum depth of trees as 3.5 and 5, respectively. "Treebank" is a

dataset of english sentences, tagged with parts of speech and is partially encrypted, with the average depth and the maximum depth of trees in dataset as 7.8 and 36, respectively.

We select tree pairs at rational size intervals and measure the cycles consumed and the time elapsed on GPU. For a given tree size n we pick the two trees in the dataset that are closest to n. The value of size used in the graphs is the average size of the two trees.

For experiments we used a CPU/GPU based platform. CPU is a Linux Machine running Intel(R) Xeon(R) CPU X5650 @2.67GHz with 24GB RAM. The GPU used is a Tesla C2070 device having 448 Streaming Processors (SPs). OpenMP version 4.0 is used. We used Kepler K20 GPUs for some of the experiments. The K20 GPU comprises 2496 Cuda cores @ 746 MhZ. Stage A runs on CPU. Stage B and Stage C runs on CPU/GPU.

We compare our work to RTED. We obtained a copy of RTED implementation from [112] and run on CPU. The RTED implementation require the trees to be represented in bracket notation. For example, bracket representation of Tree T2 (shown in Figure 5.2(b)) is as follows: {a{b{c}{d}}{e{f}}{k{l}{m}{n}}}. The trees encoded in bracket notation are kept in separate files. We use the default strategy i.e. RTED algorithm by Pawlik and Augsten [111]. We run the RTED implementation from command line (java -jar RTED.jar -f FILE1 FILE2) where FILE1 and FILE2 contain bracket representation of Trees T1 and T2 . The execution time is recorded using "time" command.

5.6 Experimental Evaluation Results

In this section, we discuss the results of the experiments for Swissprot, Treebank, and Nasa datasets. Specifically, we discuss the performance of different stages of PTED-GPU and speedup of PTED-GPU implementation to RTED.

5.6.1 Performance of PTED-GPU

Total execution time is sum of three components namely, (1) XML parsing and Data Shaping time i.e. Stage A (referred as Data Shaping Time), (2) Overhead associated with GPU specific system calls (referred as GPU overhead) (3) partial edit distance computation on GPU (Stage B) and Final Tree Edit Distance computation (Stage C). We do not consider GPU specific

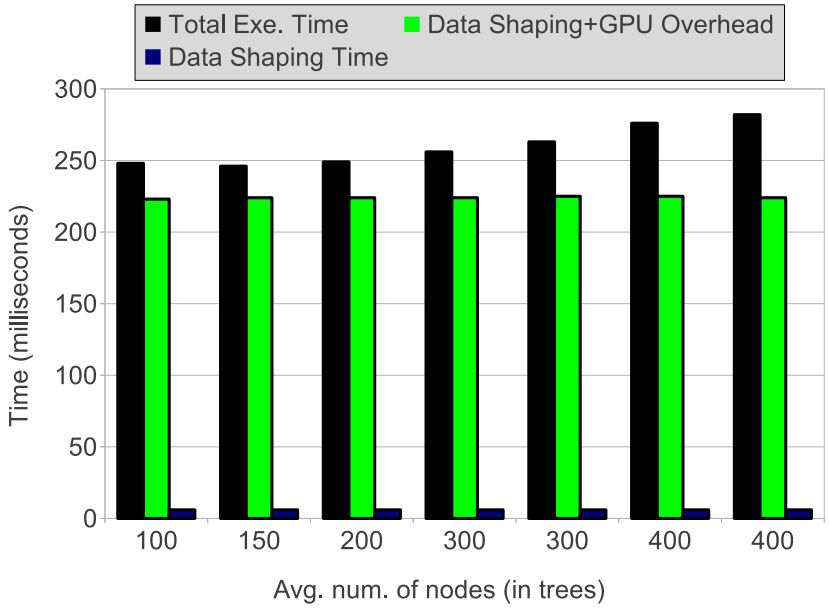


Figure 5.7: Time elapsed in various stages of PTED-GPU for "nasa".

overhead in our performance analysis since it is not due to our PTED-GPU solution. Moreover, GPU specific overhead can be ignored when several tree matching operations are pipelined.

We calculate actual execution time as: actual execution time = total execution time − GPU specific overhead. We use actual execution time (often referred as execution time) in analysis.

Figure 5.7 depicts the breakdown of time spent in various stages for "nasa" dataset. From Figure 5.7, we observe that XML parsing and data shaping take 6 milliseconds on an average. Overhead associated with GPU specific system calls (referred as GPU overhead) is around ≈218 milliseconds. Finally, actual execution time is 25, 22, and 25 milliseconds, respectively, for tree pairs having 100, 150, and 200 as the average number of nodes [131]. The execution time elapsed for tree pairs with 300 nodes as average number of nodes is around 35 milliseconds. For tree pairs with 400 average nodes, the execution time is around 55 milliseconds.

Figure 5.8 shows breakdown of time spent in various stages of PTED-GPU for "Swissprot" dataset. From Figure 5.8, we observe that XML parsing and data shaping take 7 milliseconds on average for tree pairs having 100 to 800 as the average number of nodes. The time elapsed in parsing and data shaping for tree pairs having 1000 and 1100 as average number of nodes is ∼18 milliseconds. Overhead associated with GPU specific system calls is ∼218 milliseconds.

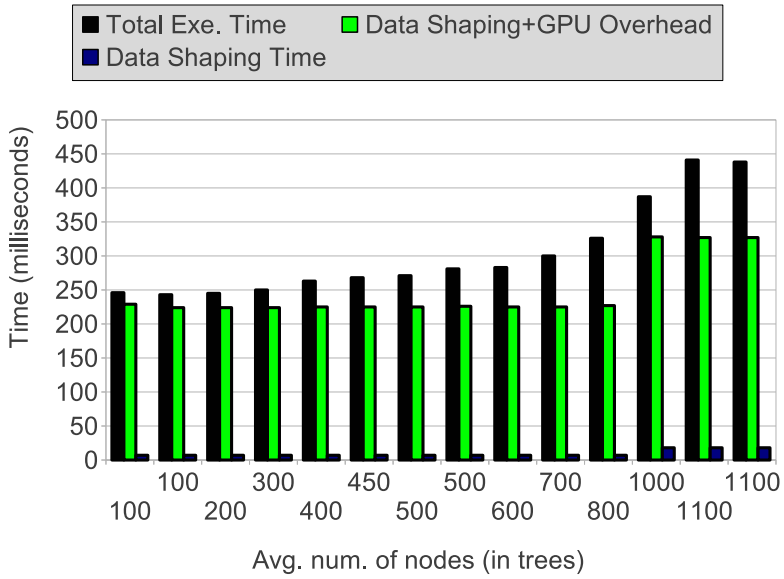


Figure 5.8: Time elapsed in various stages of PTED-GPU for "Swissprot".

Figure 5.9 shows breakdown of time spent in various stages of PTED-GPU for "Treebank" dataset. From Figure 5.9, we observe that XML parsing and data shaping take 6 milliseconds on average. Overhead associated with GPU specific system calls is ∼219 milliseconds.

5.6.2 Scalability of PTED-GPU

Figure 5.15 plots the comparison of rise in execution time vs. rise in average number of nodes for "nasa", "Swissprot", and "Treebank" datasets. From this Figure we note that

(1) for "nasa" dataset, increasing average number of nodes in tree pairs by 4X results in 2.3X increase in execution time. This indicates a sub-linear relationship between increase in execution time vs. increase in average number of nodes in tree pairs.

(2) for "Swissprot" dataset, increasing average number of nodes in tree pairs by 11X results in 9.25X increase in execution time indicating a sub-linear relationship between the execution time vs. average number of nodes in tree pairs.

(3) for "Treebank" dataset, increasing average number of nodes in tree pairs by 3X results in 1.2X increase in processing time, indicating a sub-linear relationship between the execution time vs. average number of nodes in tree pairs.

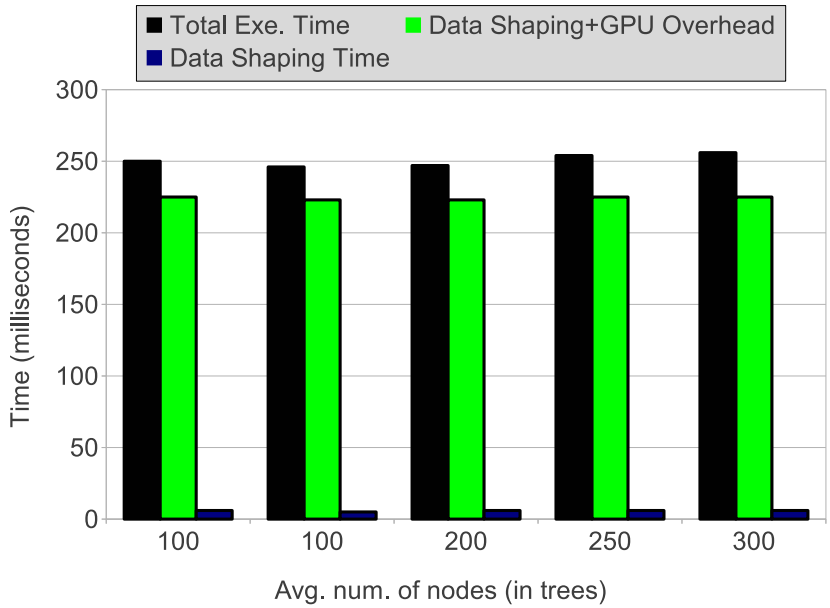


Figure 5.9: Time elapsed in various stages of PTED-GPU for "Treebank".

These observations reveal that our PTED-GPU implementation is scalable for a wide set of problem sizes.

5.6.3 Speedup: PTED-GPU vs. RTED

Our experiments show that, in general, PTED-GPU achieves a speedup of up to 12X over the RTED. Note that we do not consider the GPU specific overhead while comparing performance of PTED-GPU to PTED. Figure 5.11, 5.12, and 5.13 depicts the performance improvement of PTED-GPU implementation over the RTED (ignoring GPU specific overhead) for "nasa", "Swissprot", and "Treebank" datasets, respectively. From these figures we observe that

(1) for "nasa" dataset, the speedup obtained varies from 7X to 12X (approx.).

(2) for "Swissprot" dataset, the speedup obtained varies from 6X to 11X (approx.). We also observe that for tree pairs having average number of nodes in 100-500 range, the speedup in performance is ∼10X.

(3) for "Treebank" dataset, the speedup obtained is up to ∼11X. We also observe that for tree pairs having average number of nodes in 200-300 range, the speedup in performance is more than ∼10X.

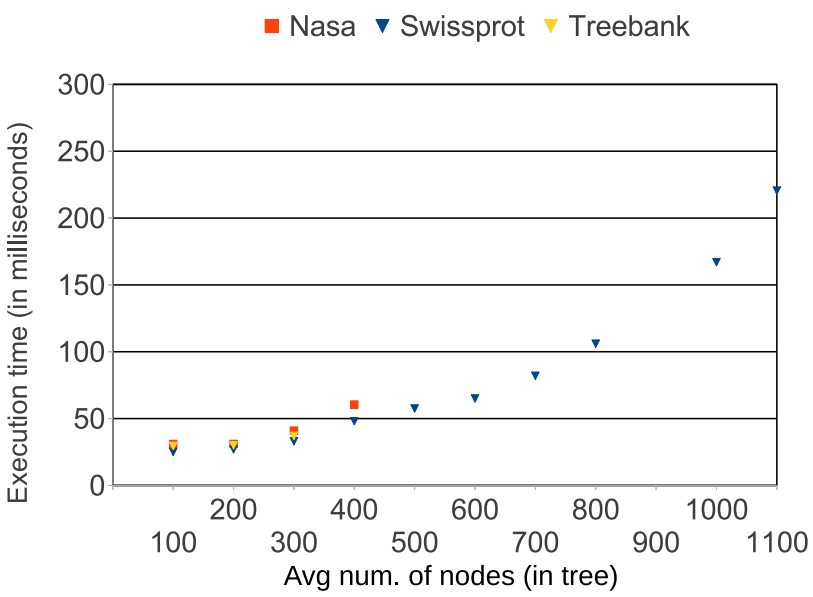


Figure 5.10: Processing time (Data Shaping time + Execution time) of PTED-GPU for nasa, SwissProt, and Treebank.

5.6.4 Performance of Partial Tree Edit Distance on CPU

We used one OpenMP thread per CPU core. We do not use hyper-threading. We set the AFFINITY=COMPACT to make sure that consecutive OpenMP threads bind to consecutive cores on the CPU.

Figure 5.15 depicts the performance on partial edit distance computations on CPU cores. X-axis represents number of OpenMP threads. Y-axis represents time elapsed in execution of 1000 iterations of a partial edit distance computations for a 380 $\times$ 418 node trees.

From this figure we observe that increase in number of OpenMP threads does reduce the time elapsed in execution. For example, by increasing number of OpenMP threads from 1 to 4, the time elapsed decreases by 3.3X. In other words, the performance increases by $\sim$3.3X. By increasing number of OpenMP threads from 4 to 8, we observe decrease in time elapsed by $\sim$1.5X. Further increase in number of OpenMP threads, i.e. from 8 to 16 or 32, shows little gain in performance.

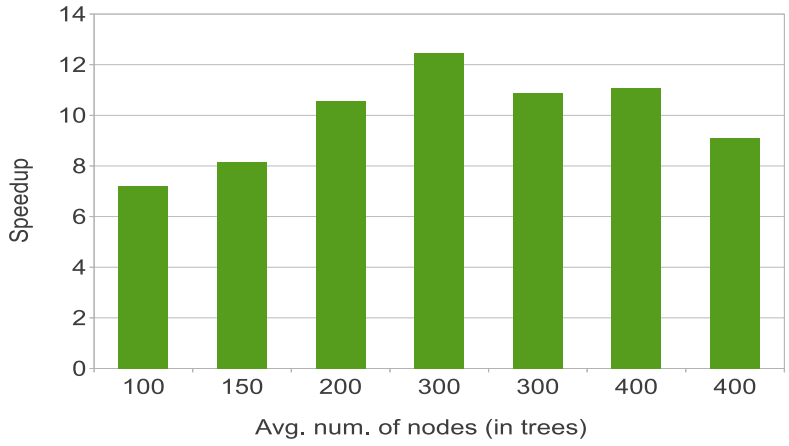


Figure 5.11: Speedup of PTED-GPU over RTED for "nasa" dataset.

5.6.5 Performance of Parallel Forest Distance Computation on CPU

In these experiments as well we used one OpenMP thread per CPU core, did not used hyper-threading, and set the AFFINITY=COMPACT (to make sure that consecutive OpenMP threads bind to consecutive cores on the CPU).

Figure 5.15 depicts the time elapsed in the computation of the largest forest denoted by "Largest". The largest forest computation involves all the nodes in both trees and represents the maximum degree of parallelism generated via our parallel version of forest distance algorithm. The figure also plots total time execution time incurred denoted by "Total_time". X-axis represents average number of nodes in trees involved. Y-axis represents time elapsed in milliseconds.

From this figure we observe that the time elapsed in forest distance computation accounts for nearly 35% to 66% of the total forest distance computation time. We also observe that if the two trees involved are similar in size then the time elapsed in forest distance computation is nearly one-third of the total forest distance computation time.

From this figure we also that nature of tress affects the forest distance computation time. Specifically, equivalence (or otherwise) in the number nodes. For example, compare the tree sizes listed as entries 5, 6, and 7 in Table 5.1. Note that average number of nodes in these three

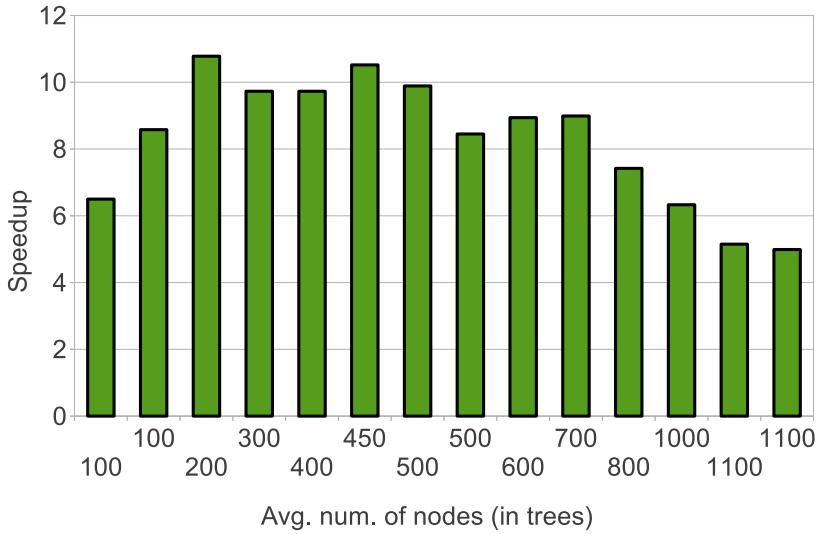


Figure 5.12: Speedup of PTED-GPU over RTED for "Swissprot" dataset.

entries is ∼400. However, the Total_time and time incurred in computation of largest forest distance is not comparable. We observe that Total_time for Tree(192, 591) is ∼ 9.8 milliseconds whereas Total_time for Tree(380, 418) is 25.3 milliseconds, which is ∼2.5X higher. Similarly, time incurred in computation of largest forest distance for Tree(192, 591) and Tree(380, 418) are ∼4.7 milliseconds and ∼9 milliseconds, respectively, which is a difference of ∼1.9X.

5.7 Related Work

Several tree mining methods have been proposed for finding similarities in context of ordered and unordered trees [20].

Typically, the similarity-based methods compare a pair of trees using nodes and paths of two trees, and estimate the similarity between them [126]. A similarity based approach exploiting the level information of tree nodes is proposed [100]. The basic idea is to consider common nodes (of the trees) in the corresponding levels and assigning different weight to different levels [100]. A Tensor Space Model (TSM) is used for representing tree data and finding similarities between a pair of trees [22]. Techniques such as TSM has drawbacks such as high dimensionality and complexity problems [68].

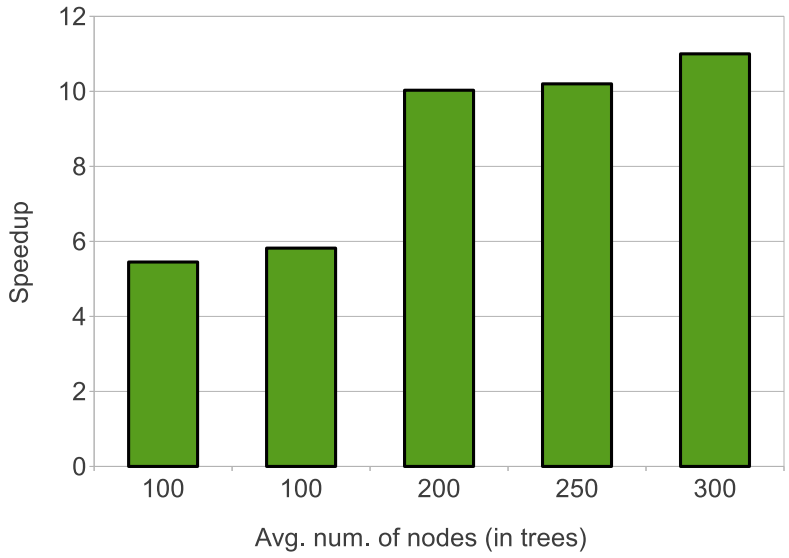


Figure 5.13: Speedup of PTED-GPU over RTED for "Treebank" dataset.

The tree edit distance (TED) based methods have also been proposed in context of unordered trees. These methods exhibit NP-hard complexity [161], [54]. Some studies address the complexity issue involved with computing tree edit distance for unordered tress. [41], [94] reduce the tree edit distance problem to the maximum clique problem. And a variant of the tree edit distance problem is proposed [136]. However, these methods are not appropriate for large unordered trees, as they suffer from high complexity [41].

There also exist other methods for matching unordered trees such as maximum agreement subtree [132], [10], [25], largest common subtree [6], and smallest common supertree [49]. These methods also suffer from the high computational complexity. For more related work see Section Background.

5.8 Conclusion

Big Data analytics in near real-time is becoming important for several objectives. Tree matching is a core component for many applications such as fraud detection, spam filtering, information visualization and extraction, user authentication, natural language processing, XML databases, bioinformatics, etc. Comparing ordered (unordered) trees is compute-intensive, in

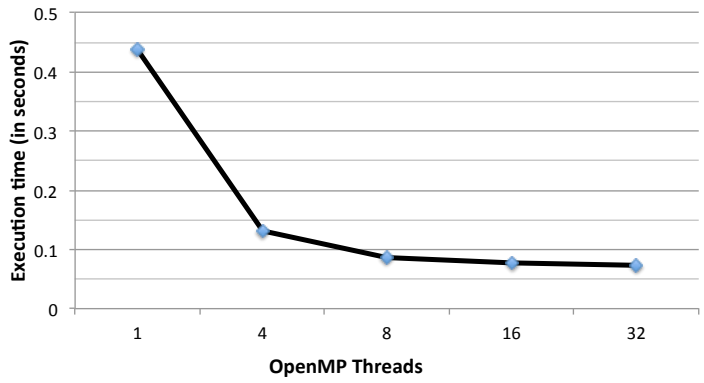


Figure 5.14: Scalability of partial edit distance computation on CPU: Time elapsed in partial edit computations for a 380 × 418 node tree. We report time incurred in 1000 iterations.

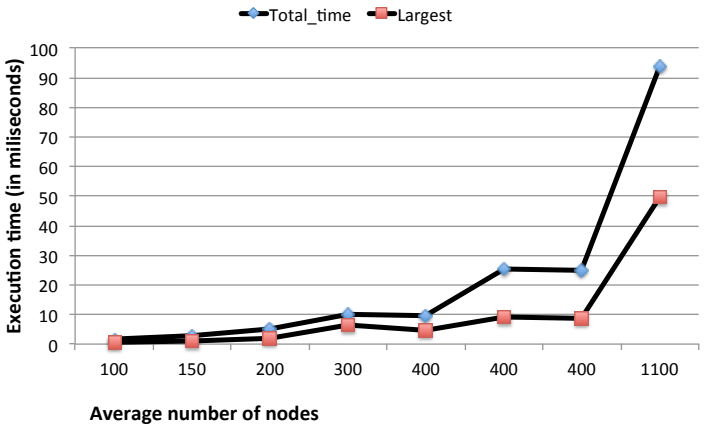


Figure 5.15: Scalability of forest distance computation on CPU.

particular for Big Data. Comparing ordered (unordered) trees is both compute- and data-intensive.

In order to facilitate comparison of ordered trees, we have addressed the following problem: ***How to shape the semi-structured data to enable time efficient processing on a parallel hardware?*** Specifically, our data shaping approach enables pre-computation of partial edit distance values in parallel. We evaluate our work using real world data sets. Our experimental results show that our SIMT-based PTED-GPU implementation shows speedup of up to 12X when compared with the state-of-the-art in tree edit distance (TED) computation.

From this work, we learned that data shaping technique helps when there is a higher

Table 5.1: Characteristics of trees used in parallel forest distance experiment (Section 5.6.5)

Index	Nodes in T1	Nodes in T2	Average	Source Dataset
1	117	119	118	Nasa
2	119	192	155	Nasa
3	192	203	197	Nasa
4	192	380	286	Nasa
5	192	591	391	Nasa
6	380	418	399	Nasa
7	311	514	412	Nasa
8	1130	1076	1103	Swissprot

proportion of leaf nodes in the tree. One limitation of our approach is that a post-processing step is needed.

CHAPTER 6. FAST DOCUMENT SIMILARITY COMPUTATIONS USING GPGPU

Summary. Several Big Data problems involve computing similarities between entities, such as records, documents, etc., in timely manner. Recent studies point that similarity-based deduplication techniques are efficient for document databases. Delta encoding-like techniques are commonly leveraged to compute similarities between documents. Operational requirements dictate low latency constraints. The previous researches do not consider parallel computing to deliver low latency delta encoding solutions. This chapter makes two-fold contribution in context of delta encoding problem occurring in document databases: (1) develop a parallel processing-based technique to compute similarities between documents, and (2) design a GPU-based document cache to accelerate the performance of delta encoding pipeline. We experiment with real datasets. Our experiments demonstrate the effectiveness of GPUs in similarity computations. Specifically, we achieve throughput of more than 500 similarity computations per millisecond.

Keywords: Deduplication, semi-structured data, NoSQL, Big Data, parallel processing

6.1 Research Problem and Contribution

Semi-structured data is a prominent component of Big Data and is becoming more important day-by-day. Increasing importance of semi-structured data has generated vast interest in document-oriented databases [1], [2]. The document databases store the semi-structured data, which is hierarchical in nature, in formats such as JSON, Avro, XML, etc. [3], [4]. Semi-structured data is also a part of scientific workflows. Scientific meta data associated with experiments, measurements, etc. are oftentimes in xml or other semi-structured format.

Oftentimes, these document databases are plagued with volume-borne challenges when dealing with data storage and data movement requirements. The scientific workflows require large scale data transfers across geographical locations. Deduplication helps in reducing the amount of data transferred and reduces the bandwidth requirement during transfers. Deduplication techniques such as similarity-based deduplication are deployed to overcome storage- and bandwidth-related issues in the document databases.

Recent studies conclude that delta encoding (compression) based deduplication offers advantages when compared to other earlier data deduplication approaches. This is due to the fact that (1) regions of similarity are small, and (2) such similar regions are scattered in the deduplication stream.

Research Problem. We address the following research problem: How can we leverage GPGPUs effectively to accelerate the delta encoding pipeline? This work has two components: (1) How can we tackle the volume-related challenges associated with processing of Big Data workloads, and (2) How can we design a GPGPU-based solution which alleviates the performance bottlenecks of existing delta encoding solutions?

Our response. Our response to first component is: (1) Design output aware techniques. For instance, computations involving favorable set of inputs must incur lesser time complexity when compared to computing with unfavorable set of inputs. Our response to second component is: (1) Design a parallel processing based solution which circumvents the pathologies of existing solutions? For example, trade the compute power of GPGPUs to avoid auxiliary data structures?

Contributions This work makes the following contributions:

- Develop a novel technique to compute degree of similarity in tree-structured data via identifying the similarity patterns.
- Develop an novel technique to compute similarity between two objects in context of delta encoding problem.

- Design a GPU-based document cache to accelerate the delta encoding pipeline in context of document databases.

The chapter is organized as follows. Section 6.4 describes our data shaping technique, and Section 6.5 describes our similarity computation technique. Section 6.6 discusses our GPU-based document cache framework for delta compression. Section 6.7 describe the experimental methodology and results obtained. Section 6.8 covers the related work and Section 6.9 concludes the chapter.

6.2 Background

Here, we cover background on delta encoding and General Purpose Graphics Processing Units (GPGPUs).

6.2.1 Delta encoding problem

Informally, delta encoding problem can be defined as follows: Given two files, $f1$ and $f2$. Goal is to compute difference file, f_d, such that it is possible to compute $f2$ using $f1$ and f_d.

Delta compression has applications in software revison control systems, data compression at file system level, software distribution for generating patches, visualizing differences between documents, improving HTTP performance, web page storage, etc [134]..

Recently, delta compression has been applied to document deduplication. For example, Xu et al. proposed a delta encoding-based technique to perform similarity-based deduplication in context of document databases.

6.2.2 Rabin-Karp fingerprinting

Rabin-Karp algorithm is basically a string search algorithm that uses hashing to find patterns in strings [59]. The Rabin-Karp algorithm relies on rolling hash technique.

Rabin-Karp fingerprinting is a type of rolling hash function commonly used with Rabin-Karp algorithm. Due to the rolling nature the Rabin-Karp fingerprinting facilitates compu-

tation of the hash value of next substring from previous one by using a constant number of opetations. And the number of operations are independent of the subtring's length.

6.2.3 Murmur Hash

Murmur hash functions are appropriate for hash-based lookups [13]. Murmur hash belongs to non-cryptographic class of hash functions and is not suitable for cryptographic applications. Various versions of Murmur hash exist. MuurmurHash3, the current version, generates a 32-bit or 128-bit hash value. A previous version is MurmurHash2 which generates 32-bit or 64-bit hash value. Like other hashing functions Murmur hash function also generates collisions.

6.2.4 Deduplication process

Here, we review deduplication process, in general. Typically, the dedupication process comprises four stages which are as follows: (1) initialization, (2) hashing, (3) searching for duplicate chunks, and (4) removal of duplicate chunks. Figure 6.1 depicts the general outline of the deduplication process.

6.2.4.1 STAGE 1: Chunking

Initialization stage comprises data chunking. Input data is divided into smaller sized chunks using a chunking method, such as Rabin-Karp fingerprinting. Stage 1 is executed on CPU.

6.2.4.2 STAGE 2: Hashing

The chunks identified in previous STAGE 1 are compared so that the duplicate chunks can be identified. Direct chunk comparison, i.e. comparing the payload of one chunk versus the payload of another chunk, is prohibitive and is typically avoided in the de-duplication solution. Instead, duplicate chunks are identified via comparing the fingerprint or hash value of the chunks. Collision resistant hash functions such as SHA-2, MD-5, etc., can be used for this purpose.

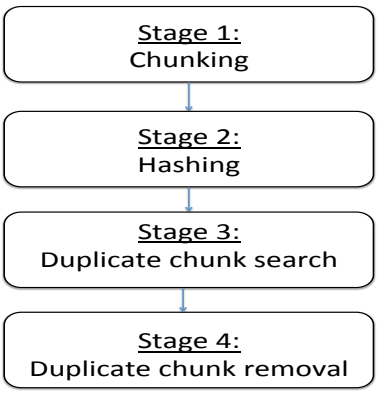


Figure 6.1: Stages in deduplication process.

6.2.4.3 STAGE 3 and 4: Duplicate Chunk Search and elimination

In this stage, hash value of a new chunk is searched against the hash values of the previous chunks seen. Searching through hash values of all the previous chunks is avoided; instead a cache of the chunks, for example, a LRU-based cache is maintained. The duplicate chunks are eliminated. In some deduplication approaches, for example delta encoding, etc., similar chunks are also eliminated as mentioned in Section 6.2.1.

6.3 Motivation

Compression performance of delta encoding techniques is directly related to the quality of candidates (the $f1's$) available. Reason being that the closer $f1$ and $f2$ are, smaller is their difference, d_d, resulting in smaller memory overheads. Better candidates can be retrived when searching through a larger corpus. And larger caches are useful in such scenarios. Larger caches present a quick scanning of larger number of candidates, possibly resulting in a better candidate(s).

We argue that the traditional caching-hashing-based approach has drawbacks in Big Data regime. We make two key observations: (1) Large caches pose performance bottlenecks for in-line low latency operational requirements, and (2) Hash collisions shoot-up when working with larger datasets which is a common case in Big Data scenario.

6.3.1 Issues with large caches

Traditional software-based caching techniques such as LRU, MRU, etc., are primarily sequential in nature. It is possible to perform parallel search/update operations in such caches. For example, we can design caches which have ten's of thousands entries courtesy the size of system memory which is up to 100's GBs in present day's server-class nodes. Using GPUs for larger, parallel caches is also a possibility, wherein we can leverage parallel processing capabilities of GPUs to perform search/update operations. Larger caches could be performance bottleneck for in-line low latency requirments of deduplication process.

6.3.2 High rate of hash collisions

With increase in number of hash values (or the entries in the cache) the probability of hash collison also increases [114]. For example, for a 32-bit hash function, the probability of hash collisions is 0.01% for ∼1,000 hash values. For ∼3,000 and ∼10,000 hash values, the probability of collision is 0.01% and 0.1%, respectively. And for ∼30,000 and ∼77,000 hash values, the probability of hash collision is 10% and 50%, respectively. In other words, if we design a cache for ∼30,000 or ∼77,000 entries, than probability of hash collision is 10% or 50%, respectively. This indicates that cache comprising more than ∼30,000 entries is not good from performance perspective.

6.4 Data Shaping for GPGPUs

We consider semi-structured form data represented in XML, JSON, etc. Figure 6.2(a) depicts one such semi-structured document represented in XML notation. The document is basically an excerpt of *revision* metadata from Mediawiki dataset [5]. Precisely, this revision document relates to one of the revisions made by some wiki user RoseParks hose user id is 99, as shown under the *contributor*. The document contains several other information such as timestamp, comment, model, format, text id, sha1 hash of that update.

A given document is organized as a sequence of objects, wherein an object is marked by the start of a node label. For example, in Figure 6.2(a), XML label *revision* shown as $< revision >$

Figure 6.2: (a) Example of history metadata document. (b) Memory representation of document.

marks the beginning of object *revision*.

Figure 6.2(b) depicts the resulting encoding and its memory representation. The object *revision* starts at byte 384, shown as B'384, in memory.

6.5 Parallel Document Similarity Computation

In this section, we describe the application of our data shaping technique (described above) to compute document similarity metric in a parallel manner.

Specifically, we first describe the similarity computation problem being considered in this chapter in Section 6.5.1. Next, we present an outline of the solution to similarity computation problem in Section 6.5.2. Remaining parts of this section covers the details of solution.

6.5.1 The Similarity Computation Problem

We consider the following problem: Given two rooted, ordered tree objects O1, O2. Also given is a similarity threshold, θ. Compute difference between the O1 and O2.

We make two key observations: (1) Closer are the objects, lesser is the similarity (dissimilarity) computation time. (2) Contiguous dissimilarity is better than disjoint dis-similarity.

Basic idea is that since the pattern of similarity is directly related to the final overhead incurred after delta encoding, it makes sense to track the patterns of similarity between the objects being considered for delta encoding. For example, if the two objects (to be encoded) are completely similar (identical) than the delta encoding overhead in insignificant.

6.5.2 Similarity Computation: An Overview

Apply data shaping technique described in Section 6.4 to obtain T1 and T2 which are linearized memory representation of O1 and O2. Compare linearized tree objects in parallel and record element-wise outcome. Perform data compaction and record dissimilarity pattern P. This is realized through the pseudocode as Listing 6.1. Classify P as a positive or negative pattern by comparing it with a set of template patterns. Specifically, P is classified as a positive pattern when P is closer to pattern(s) favorable to attain the desired δ. If P is classified as a positive pattern, then compute α_i, the similarity metric corresponding to the i^{th} orientation pair. Otherwise, P is classified as a negative pattern.

6.5.3 Similarity patterns

Figure 6.3(a) depicts four patterns of similarity distribution. A shaded box denotes that corresponding entries in the linearized version of the trees are dissimilar. First category of similarity pattern is when both source and sink are unshaded. No migration is needed this case.

Second category of similarity pattern denotes case when source is shaded while sink is unshaded. In this case, migration takes place from source to sink.

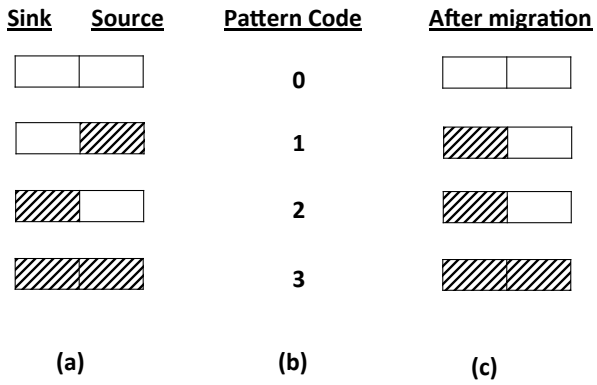


Figure 6.3: (a) Similarity patterns. Dissimilarity migration takes place from source to sink. A shaded box indicates that the corresponding entries in linearized trees are not similar. (b) Pattern codes, and (c) Distribution of similarity after migration.

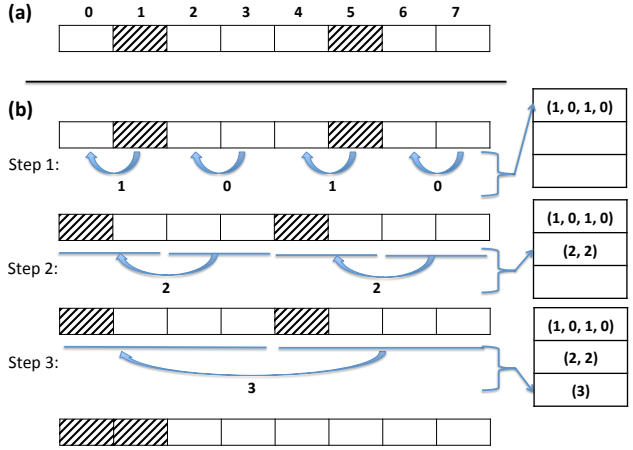


Figure 6.4: (a) Similarity distribution after comparing O1 and O2. (b) Steps to register similarity pattern. Number of steps is $\log_2(\max(O1, O2))$, which is 3 in this example. Similarity pattern code is indicated along the curved arrows. (1) In first step, cell size is 1. (2) In second step, size of cell is 2. (3) In third and final step size of 4 cells is considered. The similarity pattern is denoted as {(1, 0, 1, 0), (2, 2), (3)}. [Comment: add O1 and O2 and show the differences at index 1 and 5. And show how XORMatch is obtained.]

Third category of similarity pattern denotes case when source is unshaded while sink is shaded. No migration takes place in such cases. Fourth category of similarity pattern represents scenario of no similarity. No migration can happen in such cases.

Figure 6.3(b) depicts the pattern code associated with these four categories of similarity distribution patterns. Figure 6.3(c) depicts the distribution after migration.

Listing 6.1: Register-Pattern-GPU

```
__global__ void Register-Pattern-GPU(...)
{
  int step;  int i;  int Start;
  int maxSteps = log(max(O1, O2));
  int maxThreads = max(O1, O2)/2;

  //Implement XOR_Match here:
  if ( blockIdx.x  < num_BLOCKS )
  { if ( threadIdx.x  < maxThreads )
    { for ( i = 0; i < maxSteps; i++)
      { if ( threadIdx.x < maxThreads )
        { TID = threadIdx.x;
          if ( XORMatch[TID + 1] == 0 )
          { //No migration is needed:
            //sizeat[TID] = sizeat[TID];
            if ( XORMatch[TID] == 0 ) { Sim_Pattern[i][TID] = 0; }
            else { Sim_Pattern[i][TID] = 2;}
          }
          else
          { Start = 0;
            if ( XORMatch[TID] == 0 )
            {   Sim_Pattern[i][TID] = 1;
                        Start = 0;
            }
            else
            {  Sim_Pattern[i][TID] = 3;
                Start = sizeAT[TID];
                //Copy from TID+1 to TID:
                for (j=0;j<sizeAT[TID+1];j++) {XORMatch[TID+j] = XORMatch[(TID+1)+j];}
                sizeAT[TID] +=  sizeAT[TID+1];
            }
           }
        }
        maxThreads = maxThreads/2;
        __threadfence_system();
       }
  }} return;
}
```

6.5.4 Registering similarity pattern

Figure 6.4(a) depicts distribution of similarity obtained after comparing O1 and O2. Note that two entries are shaded. The shaded entries signify the the corresponding elements in O1 and O2 are dissimilar. Specifically, when O1 and O2 are compared element-wise, elements 1 and 5 are dissimilar.

Figure 6.4(b) depicts how we record similarity pattern for O1 and O2. The similarity registration process operates in binary reduction fashion. Total number of steps involved in registration process are bounded by $\log_2(\max(O1, O2))$, where $\max(O1, O2)$ denotes the maximum of O1 and O2. In this case, number of steps in reduction process are $\log_2(8) = 3$.

The figure depicts a three step reduction operation. In Step 1, four sets of operations occur in parallel (refer Figure 6.4(b)). Specifically, entries 0 and 1 participate in first set of comparison. Similarity pattern corresponds to type 1, which is marked on the arch from source to sink.

Second set of operation comprises entries 2 and 3. Note that both of the entries are unshaded which indicates a similarity pattern corresponds of type 0, as marked on the arch from source to sink. Third and fourth set of operations comprises entries 4 and 5, and 6 and 7, respectively. And similarity patterns corresponds to type 1 and type 0, respectively. After Step 1, pattern table is updated with similarity pattern (1, 0, 1, 0).

In Step 2, two set of operations occur in parallel. In first set of operation, entries 0-1 form the sink and entries 2-3 form source. Similarity pattern corresponds to type 2, as marked on the arch from source to sink. In second set, the participating entries are 4-7 wherein entries 4-5 form the sink and entries 6-7 form source. The similarity pattern corresponds to type 2 as marked on the arch. After Step 2, pattern table is updated with similarity pattern (2, 2).

In Step 3, one set of operation takes place wherein all the entries participate. Entries 0-3 form the sink and entries 4-7 form source. Similarity pattern corresponds to type 3, which is marked on the arch from source to sink. After Step 3, pattern table is updated with similarity pattern (3). With this step the final pattern table is as follows: {(1, 0, 1, 0), (2, 2), (3) }.

Listing 6.1 depicts the methodology to register the similarity distribution pattern.

6.5.5 Classification of similarity patterns

Next, we discuss some of the common similarity patterns from delta encoding perspective.

6.5.5.1 Favorable patterns

Intuitively, if O1 and O2 are identical, then the similarity pattern comprises of all 0's, and denoted as {(0, 0, 0, 0), (0, 0), (0)}.

If O1 and O2 are completely different, then the similarity pattern comprises all 3's, and denoted as {(3, 3, 3, 3), (3, 3), (3)}.

Consider another similarity pattern {(0, 0, X, X), (0, X), (X)}, where X=(1, 2, 3). This is similarity pattern represents a scenario where first half of O1 is identical to the first half of O2. Note that this is based on the observing similarity pattern emerging out of the first step which is {(0, 0, X, X) }.

Likewise, another similarity pattern {(X, X, 0, 0), (X, 0), (X)}, where X=(1, 2, 3), indicates that the bottom halves of O1 and O2 are identical.

Such patterns indicate that there exist a significant amount of contiguity in the similarity. Contiguous similarity pattern represent scenarios which are highly favorable for delta encoding. Reason being that the overhead resulting from delta encoding would be limited due to the contiguous nature of the similarity (or dissimilarity) in the objects being considered. Such favorable patterns are also referred as positive patterns.

6.5.5.2 Unfavorable patterns

Consider a pattern such as {(Y, Y, Y, Y), (3, 3), (3)}, where Y=(1, 2) indicates that similarity is non-contiguous. Such patterns represent worst case scenarios and are unfavorable for delta encoding from overhead aspect. Such patterns are also referred as negative patterns.

We argue that negative patterns or unfavorable patterns are not likely to benefit the cause of delta encoding. This is due to the diverging nature of the overhead accruing. The task of finding delta encoding for objects, which exhibit such negative pattern similarity, should be abandoned in favor of other more meaningful computations.

Algorithm 12 ClassifySImilarityPattern( )

```
1: while (Level < Thresh) do
2:    Check all the nodes at this level:
3:    if ( NodeValue == 0 )) then
4:       case = BestCase, Exit //This is the best case
5:    end if
6:    if ( NodeValue == 1 )) then
7:       case = HalfIdentical, //This is OK
8:    end if
9:    if ( NodeValue == 2 )) then
10:      case = HalfIdentical, //This is OK
11:   end if
12:   if ( NodeValue == 3 )) then
13:      case = Dissimilar
14:   end if
15: end while
```

6.6 GPU-based In-memory Document Cache for Fast Delta Encoding

In this section, we describe our proposed delta encoding framework. The proposed delta encoding framework comprises (1) similarity computation substrate proposed in previous section and (2) a delta encoding system.

6.6.1 The Framework

We maintain the documents in their native form. In other words, we do not hash the documents; instead we represent them as-they-are. This approach offers several advantages: (1) Representing the documents preserves their structural details whereas those crucial structural details are lost after hashing. (2) Representing documents in their native form also facilitates obtaining their similar segments or the similarity pattern. Note that this similarity pattern information is very critical for determining if the similarity is good enough to produce effective delta compression.

Figure 6.5(a) depicts our GPGPU-based document cache and delta encoding framework. The framework depicts the a set of documents as input, GPU-based document cache, a delta encoder, and output. We consider a First-In-First-Out (FIFO) based document cache.

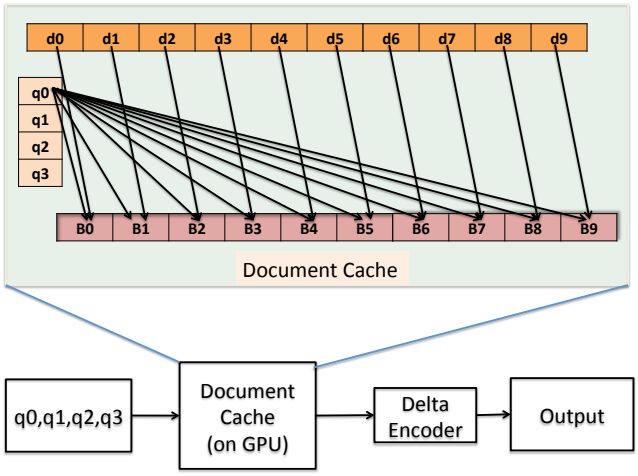


Figure 6.5: Block diagram of the framework comprising GPU-based document cache and delta encoder. The document cache is shown in detail (TOP).

Operation. Now, we describe operational aspects of the framework. The framework operates in batch mode. Input dataset is parsed and documents are organized their id-wise. A set of k input documents are presented to the document cache to obtain their corresponding similar documents. We refer to this set of k documents as query documents. The query documents are searched in document cache concurrently. If the document cache contains document(s) similar to the query document, the identifiers of the similar document(s) are passed to the delta encoder. The identifier comprises ID(s) of the similar document(s) and the similarity pattern.

6.6.2 Computation Mapping on GPU

Let N be the total number of entries in the document cache and let N_Q be the total number of entries in the query set.

Computations are organized into blocks in a 3D manner as follows:

(a) blockIdx.x = N;

(b) blockIdx.y = $max\_ydim\_threadblocks$;

(c) blockIdx.z = $max\_ydim\_threadblocks$ / N_Q, where $max\_ydim\_threadblocks$ is the maximum y- or z-dimension of a grid of thread blocks.

Note that the value of *max_xdim_threadblocks*, the maximum x-dimension of a grid of thread blocks, for Nvidia GPUs with compute capability higher than 3.0 is (2^{31}-1), which is a relatively very high number w.r.t. the size of document cache being considered in this work.

Figure 6.5(TOP) describes the mapping of document similarity computations on to the blocks of GPU using a document cache comprising 10 entries and a four query documents. The figure shows the mapping process of only one query, $q0$, for the sake of clarity. From the figure, we can see that each GPU block numbered as B0, B1, $\cdots$, B9 is assigned a copy of query document $q0$ and entries $d0$, $d1$, $\cdots$, $d9$, respectively. This set of computation forms the blockIdx.y=0. Similarly, blockIdx.y=1 handles the set of ten computations corresponding to query document $q1$. And computations due to query documents $q2$ and $q3$ are handled by blockIdx.y=2 and blockIdx.y=3, respectively.

6.7 Evaluation

This section describes the experimental methodology used and the results obtained.

6.7.1 Experimental Setup

6.7.1.1 Platform

We used Kepler K20 GPGPUs in our experiments. The K20 device comprises 2496 Cuda cores or streaming processors (SPs) @706 MHz, and is equipped with 5 GB GDDR5 on-board memory. The compute platform comes with CPU having following specifications: Intel (R) Xeon (R) CPU E5-2650 @ 2.00 GHz machine running GNU/Linux. Algorithms proposed in this chapter are implemented in C/Cuda7.5.

6.7.1.2 Datasets

We used dumps of Mediawiki revision metadata for our experiments. The dataset is in XML format. We consider each revision metadata as one document. Each document is marked within $<$ *revision* $>$ and $<$ */revision* $>$ tags. Each revision document comprises *contributor* which in turn is composed of *id* and *user name*. Then, each revision document contains several

other information such as timestamp, comment, model, format, text id, sha1 hash of the update made under that revision.

Table 6.1 lists details of the datasets. *Wiki-57MB* dataset is 57 MB in size and contains ∼ 127000 documents. Similarly, *Wiki-107MB* and *Wiki-1.1GB* datasets are 107 MB and 1.1 GB in size, respectively; and each of them comprise 236,000 and 2,364,000 documents, respectively.

Table 6.1: Description of Mediawiki Datasets

Name	Size	Num. of documents	Avg. size of document
Wiki-57MB	57 MB	126,828	471 Bytes
Wiki-107MB	107 MB	236,086	475 Bytes
Wiki-1.1GB	1.1 GB	2,363,912	500 Bytes
Wiki-11GB	11 GB	23,678,264	498 Bytes

6.7.2 Data Shaping Overhead

Input dataset comprising documents is parsed on CPU. Table 6.2 depicts the parsing overhead. From this table we observe that wall-clock time elapsed in data shaping of 107 MB input document is 1.927 seconds. From this table we also observe that for a 10.2X increase in input size, the corresponding increase in parsing time is 10.16X indicating a nearly linear relation between input size and parsing time. The overall data shaping throughput is ∼55 MB per second.

Table 6.2: Data Shaping overhead (on CPU).

Name	Parsing (on CPU)	Parsing Throughput
Wiki-107MB	1.927 seconds	55.52 MB/sec
Wiki-1.1GB	19.596 seconds	57.48 MB/sec

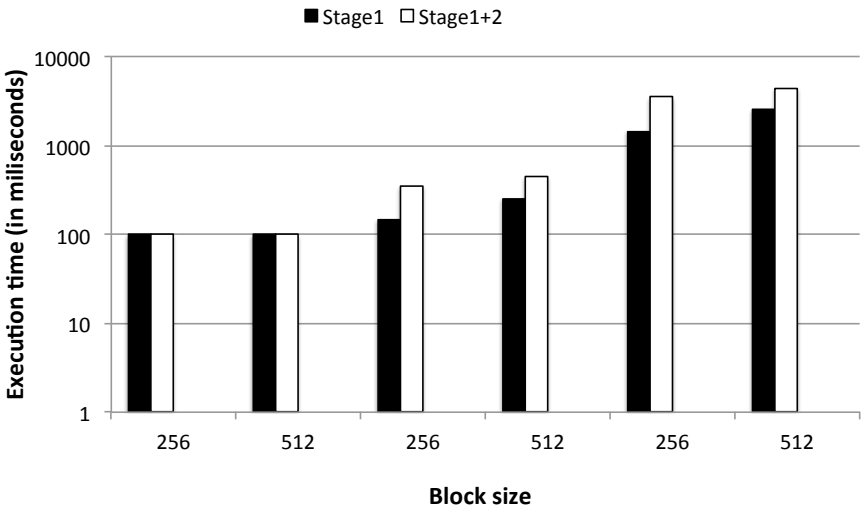


Figure 6.6: Throughput performance of similarity pattern algorithm on GPU.

6.7.3 Similarity Computation Throughput

To measure raw compute throughput of our similarity computation technique described in Section 6.5 on K20 GPU, we compute the similarity of one document from that dataset with all other documents. All of the resulting similarity computations run concurrently on GPU. Consider the 107 MB dataset Wiki-107MB which has ∼236,000 documents. As result of comparing one document against all other documents in that dataset, a total of ∼236,000 concurrent computations are generated.

Figure 6.6 plots the compute throughput of the similarity pattern algorithm for three wiki datasets. Note that the performance reported in the figure is from execution on K-20 GPU having 2496 Cuda cores. X-axis represents three Wiki datasets of 57 MB, 107 MB, and 1.1 GB in sizes. For each dataset, we experiments with GPU block sizes of 256 and 512 threads. Y-axis is log scale and represents the execution time elapsed on GPU as milliseconds.

From Figure 6.6, we observe that for block sizes of 256, time elapsed in Stage 1 for concurrent similarity computation of ∼236,000 is ∼148 milliseconds. And the time elapsed in Stage 1 and Stage 2 combined is ∼354 milliseconds. This indicates an overall raw compute throughput of 666 similarity computations per millisecond. When using a GPU block of 512 threads, the time elapsed on Stage 1 is 257 milliseconds and that elapsed in Stage 1 and Stage 2 combined is

∼450 milliseconds. These execution times yield a raw compute throughput of ∼524 similarity computations per millisecond.

Similarly, for 1.1 GB dataset, the time elapsed on Stage 1 and Stages 1, 2 combined is ∼1462 milliseconds and ∼3524 milliseconds, respectively when using block of 256 threads. This yields a raw compute throughput of ∼670 similarity computations per millisecond. And when using block size of 512, time elapsed on Stages 1 and 2 combined is ∼4478 milliseconds yielding a raw compute throughput of ∼527 similarity computations per millisecond.

6.7.4 Scalability

We determine the size of query documents and document cache empirically. To this end we conduct the following three set of experiments: (1) Small document cache having 1024 entries, (2) Medium document cache with 32K entries, and (3) Large document cache having 1M document entries. For these set of experiments we set the GPU block sizes as 512 threads.

Figure 6.7 plots performance of small document cache on varying query set size from 1 to 1024 in steps of 4X. X-axis represents the configuration of similarity computation. For example, 16*1024 denotes a configuration when query set size is 16 and document cache size is 1024. Note that the configuration also indicates the total number of document similarity computations involved for the document cache and query set being considered. Y-axis is log scale and represents the execution time elapsed on GPU as milliseconds.

From Figure 6.7, we observe that for a document cache of 1024 entries the time elapsed in Stage 1 varies from ∼372 milliseconds to ∼1346 milliseconds when the size of query set is varied from 1 to 1024. From the figure, we also observe that for the same document cache the time elapsed in stages 1 and 2 combined varies from ∼374 milliseconds to ∼2649 milliseconds. These execution time measurements reveal that for our small document cache, an increase in query set size by 1024X results in ∼7X rise in execution time. This indicates that medium document cache is highly scalable.

Figure 6.8 plots performance of medium document cache on varying query set size from 1 to 16*1024 in steps of 4X. X-axis represents the configuration of similarity computation. Y-axis is log scale and represents the execution time elapsed on GPU in milliseconds. From Figure 6.8,

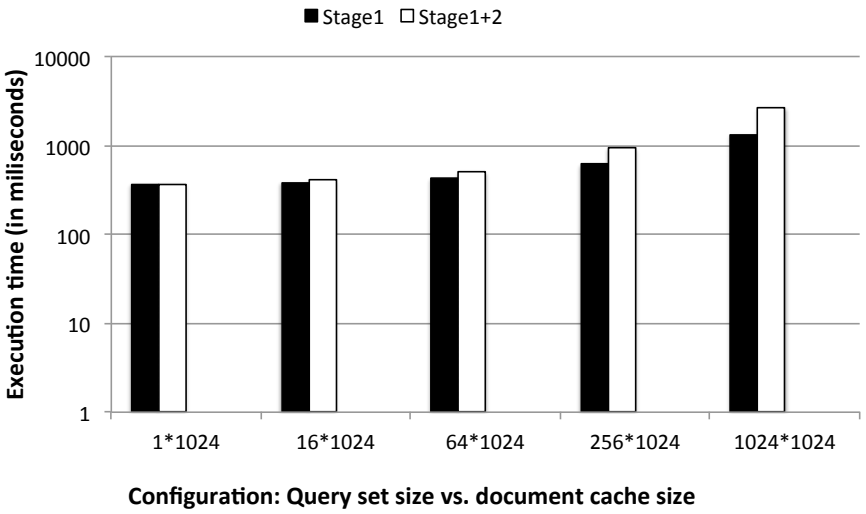


Figure 6.7: Performance of small document cache on varying size of query set.

we observe that for a document cache of 32K entries, where K=1024, the time elapsed in Stage 1 varies from ∼391 milliseconds to ∼125 seconds when the size of query set is varied from 1 to 4K. From the figure, we also observe that for the same document cache the time elapsed in stages 1 and 2 combined varies from ∼430 milliseconds to ∼293 seconds. These measurements reveal that for our medium document cache, an increase in query set size by 4096X results in ∼682X rise in execution time. This indicates that medium document cache is scalable.

Similarly, Figure 6.9 plots performance of our large document cache. The query set of size 1 and 32 are used. X-axis represents the configuration of similarity computation and log-scale Y-axis represents the time in milliseconds. From this figure, we observe that for our large document cache, the time elapsed in execution is ∼2.6 seconds and ∼71.85 seconds for query set of size 1 and 32, respectively.

6.7.5 Comparison with caching-hashing-based approach

6.7.5.1 Performance of caching-hashing approach

Recall that the deduplication process comprises several stages. First stage comprises chunking for which Rabin-Karp fingerprinting is commonly used. Hashing is the second stage. Duplicate chunk search and removal are third and fourth stages, respectively. Refer Section 6.2.4.

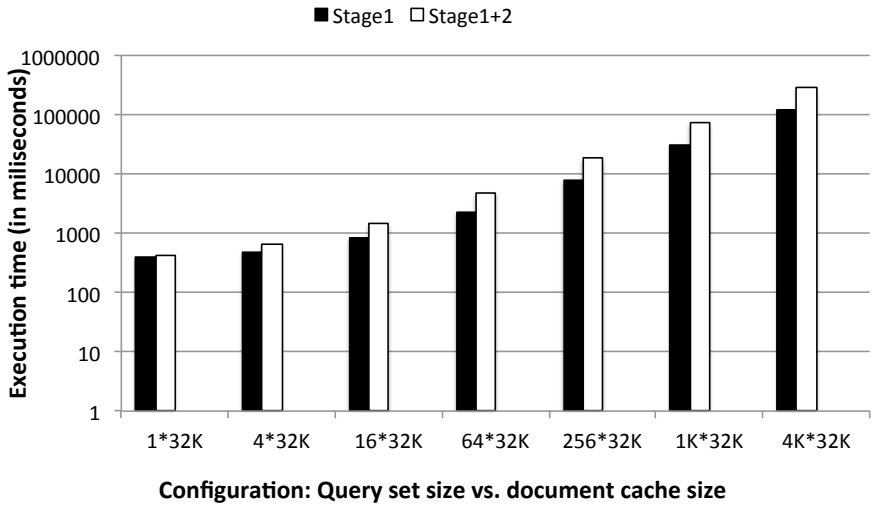


Figure 6.8: Performance of medium document cache on varying size of query set.

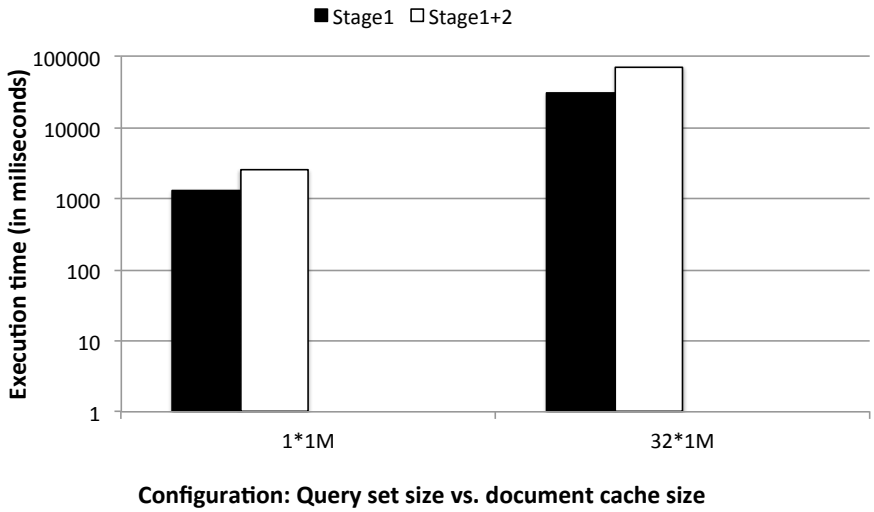


Figure 6.9: Performance of large document cache on varying size of query set.

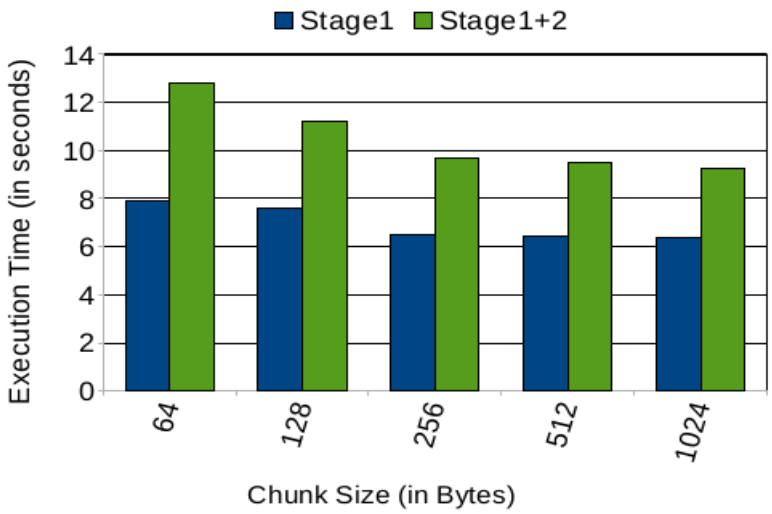


Figure 6.10: Performance of chunking and hashing stages in baseline de-duplication scheme using 600 MB text data from Wikimedia dump.

Figure 6.10 plots the the performance of Rabin-Karp-based chunking (stage 1) and MD-5 hashing (stage 2) of the deduplication process. From this figure, we observe that the time elapsed in chunking and hashing stages reduces gradually as chunks sizes are increased. Chunking time overhead dominates.

Figure 6.11 plots the performance of baseline approach with LRU caching. We used caches of 1K, 2K, 4K, and 8K entries. The experiment reveals that larger caches when used in context of small chunks are prohibitive. For example, when using 64 Byte chunks in combination with cache size 8K, the execution time is in the range of ∼10 minutes. This indicates a throughput performance of 1 MB/sec. For the same chunk size, smaller cache yields a execution time of ∼1 minute, or a throughput performance of ∼10 MB/sec. Working with larger chunks, for example chunk size of 1024 Bytes, yields execution time in the range of 10 seconds to 25 seconds, resulting in a throughput performance in the range of 24-60 MB/sec.

6.8 Related Work

A node encoded tree sequence (NETS) approach is proposed in context of filtering twig queries over XML documents [124]. We build a Pre-Pre-Post encoding algorithm that builds upon the the NETS algorithm. The study in NETS leverages tree sequencing to convert the

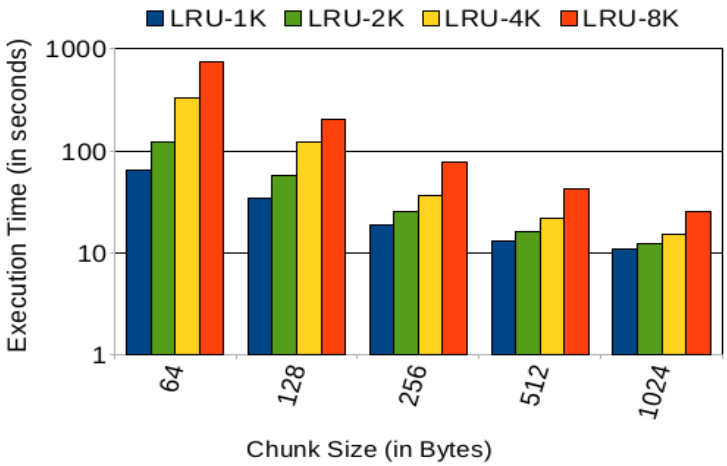


Figure 6.11: Performance of baseline de-duplication using LRU caching with 600 MB text data from Wikimedia dump.

problem of twig filtering into a sub string matching problem. Our work differs from [124] in two aspects: algorithmic level and objective level. (1) We cover the attributes (which are registered in pre-order traversal) while sequencing the tree structure which is ignored by NETS. (2) We use tree sequencing to exploit the architectural features of GPGPU.

A study proposes algorithms for delta compression and remote file synchronization [134]. Mogul et al. studies benefits of delta encoding and data compression for HTTP [91]. File system support for delta compression [80]. Zdelta, a tool for delta compression [138], [139]. A cluster-based delta compression of a collection of files is studied in [106]. A pre-cache similarity-based delta compression for use in a data storage system is studied in [156].

Recently, Shilane et al. focussed on preferential selection of candidates for delta compression [127]. Zhang et al. focussed on reducing solid-state storage device write stress through opportunistic in-place delta compression[162]. Xia et al. proposed DARE which is a A Deduplication-Aware Resemblance Detection and Elimination Scheme for Data Reduction with Low Overheads [152].

Wen et al. proposed edelta which is a word-enlarging based fast delta compression approach [151]. Li et al. explored use of hardware accelerator for similarity based data deduplication[77]. Zhang et al. explored application of delta compression for energy efficient STT-RAM register file on GPGPUs [159].

6.9 Conclusion

Several Big Data problems involve computing similarities between entities, such as records, documents, in timely manner. Recent studies point that similarity-based deduplication techniques are efficient for document databases. Delta encoding-like techniques are commonly leveraged to compute similarities between documents. Operational requirements dictate low latency constraints. The previous researches do not consider parallel computing to deliver low latency delta encoding solutions.

Through this chapter, we made a two-fold contribution in context of delta encoding problem occurring in document databases: (1) developed a parallel processing-based technique to compute similarities between documents, and (2) designed a GPU-based document cache to accelerate the performance of delta encoding pipeline. We experiment with real datasets. Our experiments demonstrate the effectiveness of GPUs in similarity computations. Specifically, we achieve throughput of more than 500 similarity computations per millisecond.

CHAPTER 7. CONCLUSION AND FUTURE OUTLOOK

In this dissertation, we explore application of *platform-centric data shaping approach* to tackle semi-structured data- and GPGPU-specific issues in context of similarity computation problems. Specifically, we address the problem of *shaping* or reorganizing data inorder to exploit parallel processing capabilities of GPGPU, GPGPU-like machines. In this chapter, we provide important conclusions derived and discuss future outlook.

7.1 Conclusion

Data processing pipelines comprise several tasks such as preprocessing, algorithmic operation, computation of metrics, etc. Similarity computation is a fundamental operation common to these tasks (preprocessing, algorithmic operation, computation of metrics). We consider similarity computation problems occurring in context of semi-structured data, for example, matching tree objects. Technology and processor architecture trends indicate that future processors shall have ten's of thousands of cores, and that ratio between on-chip and off-chip memory to core counts is decreasing. Accelerators such as General Purpose Graphics Processors (GPUs) and MICs are promising for high performance as well high throughput computing. However, processing semi-structured form data efficiently using parallel computing machines (e.g. GPUs) is challenging, due to the highly structured nature of the such machines, where several cores (streaming processors) operate in lock-step manner, or they require a high degree of task-level parallelism.

Effective and efficient solutions to similarity computation problems in data processing pipelines need to operate in a synergistic manner with the underlying computing hardware. Semi-structured form input data needs to be shaped or reorganized with the goal to exploit

enormous computing power of *state-of-the-art* highly threaded architectures (Nvidia GPUs, Intel's KNL, KNH, etc.).

Preprocessing is an operation common at initial stages of data processing pipelines. Typically, the preprocessing involves operations such as data extraction, data selection, etc. In context of semi-structured data, twig filtering is used in identifying (and extracting) data of interest. Duplicate detection and record linkage operations are useful in preprocessing tasks such as data cleaning, data fusion, and also useful in data mining, etc., in order to find similar tree objects. Likewise, tree edit is a fundamental metric used in context of tree problems; and similarity computation between trees another key problem in context of Big Data. Next, we discuss key lessons learned.

Twig filtering problem is latency sensitive, and accelerators (GPUs, MICs) comprise numerous cores/threads which operate in parallel. Therefore, we developed a data shaping technique which can harness compute resources available in these parallel machines. Specifically, we first developed a data shaping technique to reorganize semi-structured data in a manner amenable for processing on parallel architecture machines. The data shaping enables large number of threads to operate in parallel, resulting in efficient utilization of compute resources of highly parallel architecture machines. The data shaping-based filtering solution exhibits consistent performance. Variation in performance due to thread scheduling schemes is, less than ten percent, nearly in same ballpark range. Similalry, effect of block sizes on variability in filtering performance when using GPU is less than ten percent again in the same ballpark range. Hence, choice of appropriate runtime parameters (core/thread and GPU block sizes) is not the key factor for performance. Scaling performance is also consistent which help estimate the resource (core/thread count and GPUs) requirement.

Techniques for detecting duplicate and similar records occurring in semi-structured datasets exploit schema-related knowledge for efficiency. However, such schema-bound approaches are not appropriate when dealing with multi-sourced, heterogeneous, high velocity data. We developed a novel context-aware techniques to detect duplicates and similar records. The proposed techniques operates in schema-oblivious manner, and relies upon information theory based heuristic and data shaping technique for efficiency and scalability when dealing with

multi-sourced, heterogeneous datasets. The information theretic heuristic helps exploit the context, i.e., information such as expected number of duplicates, etc., and reduces computational complexity of key intermediate stages. In addition, our data shaping technique for GPGPU processing speeds up the throughput by up to two orders of magnitude.

Research in Chapter Five facilitates comparison of ordered trees by shaping the semi-structured data to enable time efficient processing on a parallel hardware. Specifically, our data shaping approach enables pre-computation of partial edit distance values in parallel. Experiments using real world dataset indicate that GPUs are effective in pre-computation of partial edit distance values which results in overall speedup gains.

From this work, we learned that our technique performs better when there are a higher proportion of leaf nodes in the tree. Speedup due to fine-grained parallel processing are limited by the following factors: (1) average number of nodes in subtrees, (2) branch divergence resulting due to structure of the trees, and (3) average number of nodes in trees. Trees with larger number of nodes such that a distribution of leaves is highly skewed towards the leaf nodes is most suitable for processing on GPU when using our data shaping technique.

Recent studies point that similarity-based deduplication techniques are efficient for document databases. Delta encoding-like techniques are commonly used to compute similarities between the documents. Operational requirements dictate low latency constraints. In this context, the research in Chapter six proposes a GPU-based document cache. We developed a parallel processing-based technique to compute similarities between documents, which is an important feature of the document cache. The GPU-based document cache accelerates performance of delta encoding pipeline. Experiments with real datasets yield a throughput of more than 500 similarity computations per millisecond. With a throughput of more than 500 similarity computations per millisecond, this technique is appropriate for online data processing pipelines and cloud-based settings.

7.2 Future Outlook

The data shaping technique proposed in Chapter Three works on encoded data. The queries and the input documents need to be encoded. The data encoding incurs a finite overhead and

can have implications of performance. Two approaches are possible. One approach is to reduce the overhead associated with data encoding process. Another approach is to avoid data encoding and use the input as it is. Avoiding data encoding has a disadvantage: the footprint of working memory or working set size (WSS) of the application is likely to increase which may have some performance implication when using GPUs.

Study in Chapter Four suggests several directions for research. One of the directions is to explore design space to address the record linkage problem occurring in unstructured data domain leveraging the research discussed in this chapter. Another research direction is to explore the applicability of similar hash-based algorithms for efficient mining of linked records in semistructured and unstructured data sets using parallel architecture-based processors.

The research in Chapter Five points that changes to structure of trees, for example, orientation of subtree, can result in different computation patterns. This idea can be used to design a efficient techniques for tree matching problems in context of unordered trees and for problems involving tree rotations common in bioinformatics.

Deduplication with variable sized chunking (on CPU) incurs a finite overhead which could a bottleneck in performance. The research in Chapter Six suggests a research direction on how to increase throughput of deduplication process. To achieve a high throughput deduplication, for example, wire-rate throughput, we need to address the overhead associated with variables-sized chunking. This can be done designing faster variable-sized chunking algorithms. Another possible approach is to shun the variable-sized chunking and use fized-sized chnking instead. The fixed-sized chunking has one drawback: the fixed-sized chunking techniques lead to reduction in deduplication gains. In order to mitigate the losses incurred in deduplication due to fixed sized chunking we can use smaller chunk sizes.

Bibliography

[1] Expat: The expat xml parser. 2012. URL http://expat.sourceforge.net/.

[2] Hasso plattner institut: Repeatability data sets. August 2012. URL http://www.hpi.uni-potsdam.de/naumann/projekte/repeatability/datasets.html.

[3] Top 500: The list. *top500.org*, November 2015. URL http://www.top500.org/lists/2015/11/.

[4] LuxMark Database: Overall top 20 medium benchmark. 2017. URL http://www.luxrender.net/luxmark/.

[5] Wikimedia downloads. Accessed: 2016-09-30. URL https://dumps.wikimedia.org.

[6] Tatsuya Akutsu and Magnús M Halldórsson. On the approximation of largest common subtrees and largest common point sets. *Theoretical Computer Science*, 233(1):33–50, 2000.

[7] S. Al-Khalifa, HV Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 141–152. IEEE, 2002.

[8] María Alpuente and Daniel Romero. A visual technique for web pages comparison. *Electronic Notes in Theoretical Computer Science*, 235:3–18, 2009.

[9] Maria Alpuente and Daniel Romero. A tool for computing the visual similarity of web pages. In *SAINT*, pages 45–51, 2010.

[10] Amihood Amir and Dmitry Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6): 1656–1669, 1997.

[11] Ion Androutsopoulos and Prodromos Malakasiotis. A survey of paraphrasing and textual entailment methods. *J. of Artificial Intelligence Research.*, pages 135–187, 2010.

[12] Apache. Welcome to apache avro! 2017. URL https://avro.apache.org.

[13] Austin Appleby. Murmurhash 2.0, 2008.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[15] Nikolaus Augsten, Michael Bohlen, Curtis Dyreson, and Johann Gamper. Approximate joins for data-centric xml. In *ICDE 2008*, pages 814–823. IEEE, 2008.

[16] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[17] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th Int. Conf. on*, pages 567–576. IEEE, 2014.

[18] Omer Barkol, Ruth Bergman, and Shahar Golan. Mining web applications, October 11 2011. US Patent App. 13/271,036.

[19] Kirill Belyaev and Indrakshi Ray. Towards efficient dissemination and filtering of xml data streams. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, pages 1870–1877. IEEE, 2015.

[20] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1):217–239, 2005.

[21] David Guy Brizan and Abdullah Uz Tansel. A survey of entity resolution and record linkage methodologies. *Communications of the IIMA*, 6(3):5, 2015.

[22] Deng Cai, Xiaofei He, and Jiawei Han. Tensor space model for document analysis. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 625–626. ACM, 2006.

[23] Saverio Caminiti, Irene Finocchi, and Rossella Petreschi. On coding labeled trees. *Theoretical Computer Science*, 382(2):97–108, 2007.

[24] Cassandra. Accessed: Jan 2014. URL http://cassandra.apache.org/.

[25] Richard Cole, Martin Farach-Colton, Ramesh Hariharan, Teresa Przytycka, and Mikkel Thorup. An o (n log n) algorithm for the maximum agreement subtree problem for binary trees. *SIAM Journal on Computing*, 30(5):1385–1404, 2000.

[26] Mariano P Consens and Tova Milo. Algebras for querying text regions. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–22. ACM, 1995.

[27] International Data Consortium. Idc digital universe study: Big data, bigger digital shadows and biggest growth in the far east. *On Web*, December 2012. URL http://www.sintef.no/en/corporate-news/big-data--for-better-or-worse/.

[28] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[29] Intel Corporation. Big data analytics: Peer research report 2012., August 2012. URL http://www.intel.com/content/dam/www/public/us/en/documents/reports/data-insights-peer-research-report.pdf.

[30] Intel Corporation. Intel atom processor. Accessed: Jan 2014. URL http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html.

[31] Intel Corporation. Intel xeon phi. product family: Product brief. Accessed: Jan 2017. URL http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[32] BaseX: The XML Database. Accessed: Jan 2014. URL http://basex.org/.

[33] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. In *Automata, languages and programming*, pages 146–157. Springer, 2007.

[34] Narsingh Deo and Paulius Micikevicius. A new encoding for labeled trees employing a stack and a queue. *Bulletin of the Institute of Combinatorics and its Applications*, 34: 77–85, 2002.

[35] Y. Diao, P. Fischer, M.J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341–342. IEEE, 2002.

[36] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[37] Ase Dragland. Big data – for better or worse. *SINTEF (Online)*, May 2013. URL http://www.sintef.no/en/corporate-news/big-data--for-better-or-worse/.

[38] Wayne W. Eckerson. Data quality and the bottom line. *The Data Warehousing Institute (TDWI)*, February 2002.

[39] The Economist. Special report on managing information: Data, data everywhere. Accessed: Jan 2014. URL http://www.economist.com/node/15557443/.

[40] Amy Feekin and Zhengxin Chen. Duplicate detection using k-way sorting method. In *Proc. of the ACM symp. on Applied computing-Volume 1*, pages 323–327. ACM, 2000.

[41] Daiji Fukagawa, Takeyuki Tamura, Atsuhiro Takasu, Etsuji Tomita, and Tatsuya Akutsu. A clique-based method for the edit distance between unordered trees and its application to analysis of glycan structures. *BMC bioinformatics*, 12(Suppl 1):S13, 2011.

[42] Phillip B Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[43] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 10. ACM, 2013.

[44] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. *Database TheoryICDT 2003*, pages 173–189, 2002.

[45] T.J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems (TODS)*, 29(4):752–788, 2004.

[46] GREEN500. The green500 list – june 2015. *green500.org*, June 2015. URL http://www.green500.org/lists/green201506.

[47] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[48] Sudipto Guha, HV Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2002.

[49] Arvind Gupta and Naomi Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21(2):183–210, 1998.

[50] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th Int. Conf. on*, pages 363–366. IEEE, 2011.

[51] Oktie Hassanzadeh, Ken Q Pu, Soheil Hassas Yeganeh, Renée J Miller, Lucian Popa, Mauricio A Hernández, and Howard Ho. Discovering linkage points over web data. *Proceedings of the VLDB Endowment*, 6(6):445–456, 2013.

[52] Apache HBase. Accessed: Jan 2014. URL http://hbase.apache.org/.

[53] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD Record*, volume 24, pages 127–138. ACM, 1995.

[54] Kouichi Hirata, Yoshiyuki Yamamoto, and Tetsuji Kuboyama. Improved max snp-hard results for finding an edit distance between unordered trees. In *Combinatorial Pattern Matching*, pages 402–415. Springer, 2011.

[55] HPCWire. Gpus power one-third of top russian supercomputers. Accessed: March 2014. URL http://www.hpcwire.com/2013/10/02/gpus-power-one-third-of-top-russian-supercomputers/.

[56] Joel Hruska. The death of cpu scaling: From one core to many and why we are still stuck. Accessed: Jan 2014. URL http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck/3.

[57] Binary JSON. Accessed: Jan 2014. URL http://bsonspec.org/.

[58] Introducing JSON. Accessed: Jan 2014. URL http://www.json.org/.

[59] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[60] David Kay and Mark van Harmelen: Activity data delivering benefits from the data deluge. Accessed: Jan 2014. URL http://www.jisc.ac.uk/publications/reports/2012/activity-data-delivering-benefits.aspx/.

[61] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and

gpus. In *Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of data*, pages 339–350. ACM, 2010.

[62] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. *ACM Transactions on Database Systems (TODS)*, 36(4):22, 2011.

[63] Hung-sik Kim and Dongwon Lee. Harra: fast iterative hashed record linkage for large-scale data collections. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 525–536. ACM, 2010.

[64] Toralf Kirsten, Lars Kolb, Michael Hartung, Anika Groß, Hanna Köpcke, and Erhard Rahm. Data partitioning for parallel entity matching. *arXiv preprint arXiv:1006.5309*, 2010.

[65] J Steven Kirtzic, Ovidiu Daescu, and TX Richardson. A parallel algorithm development model for the gpu architecture. In *Proc. of Intl Conf. on Parallel and Distributed Processing Techniques and Applications*, 2012.

[66] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[67] Milen Kouylekov and B Magnini. Recognizing textual entailment with tree edit distance. In *Proceedings of the PASCAL RTE Challenge*, pages 17–20, 2005.

[68] Sangeetha Kutty, Richi Nayak, and Yuefeng Li. Xml documents clustering using tensor space model–a preliminary study. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 1167–1173. IEEE, 2010.

[69] J. Kwon, P. Rao, B. Moon, and S. Lee. Fist: scalable xml document filtering by sequencing twig patterns. In *Proceedings of the 31st international conference on Very large data bases*, pages 217–228. VLDB Endowment, 2005.

[70] YAML: YAML Ain't Markup Language. Accessed: Jan 2014. URL http://www.yaml.org/.

[71] Luís Leitão and Pável Calado. Duplicate detection through structure optimization. In *Proc. of ICKM*, pages 443–452. ACM, 2011.

[72] Luís Leitão and Pável Calado. An automatic blocking strategy for xml duplicate detection. *ACM SIGAPP Applied Computing Review*, 13(2):42–53, 2013.

[73] Luís Leitão and Pável Calado. Efficient xml duplicate detection using an adaptive two-level optimization. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 832–837. ACM, 2013.

[74] Luís Leitão, Pável Calado, and Melanie Weis. Structure-based inference of xml similarity for fuzzy duplicate detection. In *Proc. of CIKM*, pages 293–302. ACM, 2007.

[75] Luís Leitão, Pável Calado, and Melanie Herschel. Efficient and effective duplicate detection in hierarchical data. 2012.

[76] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[77] Dongyang Li, Qingbo Wang, Cyril Guyot, Ashwin Narasimha, Dejan Vucinic, Zvonimir Bandic, and Qing Yang. Hardware accelerator for similarity based data dedupe. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 224–232. IEEE, 2015.

[78] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1254–1265. IEEE, 2011.

[79] Xingkong Ma, Yijie Wang, and Xiaoqiang Pei. A scalable and reliable matching service for content-based publish/subscribe systems. *IEEE Transactions on Cloud Computing*, 3(1):1–13, 2015.

[80] Josh MacDonald. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[81] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[82] Abdullah-Al Mamun, Tian Mi, Robert Aseltine, and Sanguthevar Rajasekaran. Efficient sequential and parallel algorithms for record linkage. *Journal of the American Medical Informatics Association*, 21(2):252–262, 2014.

[83] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008. URL http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html.

[84] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[85] Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[86] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

[87] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[88] Paulius Micikevicius, Saverio Caminiti, and Narsingh Deo. Linear-time algorithms for encoding trees as sequences of node labels. *Congressus Numerantium*, 183:65, 2006.

[89] Diego Milano, Monica Scannapieco, and Tiziana Catarci. Structure aware xml object identification. *IEEE Data Eng. Bull.*, 29(2):67–74, 2006.

[90] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras. Boosting xml filtering with a scalable fpga-based architecture. *arXiv preprint arXiv:0909.1781*, 2009.

[91] Jeffrey C Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 181–194. ACM, 1997.

[92] Reza Mokhtari and Michael Stumm. Bigkernel–high performance cpu-gpu communication pipelining for big data-style applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 819–828. IEEE, 2014.

[93] MongoDB. Accessed: Jan 2014. URL http://www.mongodb.org/.

[94] Tomoya Mori, Takeyuki Tamura, Daiji Fukagawa, Atsuhiro Takasu, Etsuji Tomita, and Tatsuya Akutsu. A clique-based method using dynamic programming for computing edit distance between unordered trees. *Journal of computational biology*, 19(10):1089–1104, 2012.

[95] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V.J. Tsotras. Efficient xml path filtering using gpus. 2011.

[96] R. Moussalli, M. Salloum, W. Najjar, and V.J. Tsotras. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 948–959. IEEE, 2011.

[97] Claudiu CN Musat. Layout-based electronic communication filtering systems and methods, May 17 2011. US Patent 7,945,627.

[98] Felix Naumann and Melanie Herschel. An introduction to duplicate detection. *Synthesis Lectures on Data Management*, 2(1):1–87, 2010.

[99] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[100] Richi Nayak. Fast and effective clustering of xml data using structural information. *Knowledge and Information Systems*, 14(2):197–215, 2008.

[101] Nvidia. Supercomputing at 1/10th the cost, July 2010. URL http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[102] Nvidia. Nvidia 16nm pascal based tesla p100 with gp100 gpu unveiled worlds first gpu with hbm2 and 10.6 tflops of compute on a single chip. 2017. URL http://wccftech.com/nvidia-pascal-tesla-p100-gp100-gpu/.

[103] Nvidia. Cuda parallel computing platform, Aa yy. URL http://www.nvidia.com/object/cuda_home_new.html.

[104] Benjamin Okundaye, Sigrid Ewert, and Ian Sanders. A novel approach to visual password schemes using tree picture grammars, Aa yy. URL www.prasa.org/proceedings/2014/prasa2014-43.pdf.

[105] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: insert-friendly xml node labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908. ACM, 2004.

[106] Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Web Information Systems Engineering, 2002. WISE 2002. Proceedings of the Third International Conference on*, pages 257–266. IEEE, 2002.

[107] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[108] Charles C Palmer and Aaron Kershenbaum. Representing trees in genetic algorithms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 379–384. IEEE, 1994.

[109] George Papadakis and Wolfgang Nejdl. Efficient entity resolution methods for heterogeneous information spaces. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 304–307. IEEE, 2011.

[110] Marcus Paradies, Susan Malaika, Jérôme Siméon, Shahan Khatchadourian, and Kai-Uwe Sattler. Entity matching for semistructured data in the cloud. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 453–458. ACM, 2012.

[111] Mateusz Pawlik and Nikolaus Augsten. Rted: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[112] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Manual, 2012. URL http://www.inf.unibz.it/dis/projects/tree-edit-distance/documentation.php.

[113] Sally Picciotto. *How to encode a tree*. PhD thesis, University of California, San Diego, 1999.

[114] Jeff Preshing. Hash collision probabilities. Accessed: March 2017. URL http://preshing.com/20110504/hash-collision-probabilities/.

[115] Heinz Prüfer. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys*, 27: 742–744, 1918.

[116] Sven Puhlmann, Melanie Weis, and Felix Naumann. Xml duplicate detection using sorted neighborhoods. In *Advances in Database Technology-EDBT 2006*, pages 773–791. Springer, 2006.

[117] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[118] Banda Ramadan and Peter Christen. Forest-based dynamic sorted neighborhood indexing for real-time entity resolution. In *Proceedings of the 23rd ACM Int. Conference on Conference on Information and Knowledge Management*, pages 1787–1790. ACM, 2014.

[119] P. Rao and B. Moon. Prix: Indexing and querying xml using prufer sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 288–299. IEEE, 2004.

[120] Antonio Regalado. The data made me do it: The next frontier for big data is the individual. *MIT Technology Review*, May 2013. URL http://www.technologyreview.com/news/514346/the-data-made-me-do-it/.

[121] Davi De Castro Reis, Paulo Braz Golgher, Altigran Soares Silva, and AF Laender. Automatic web news extraction using tree edit distance. In *Proc. of the 13th int. conf. on World Wide Web*, pages 502–511. ACM, 2004.

[122] Pratanu Roy, Jens Teubner, and Gustavo Alonso. Efficient frequent item counting in multi-core hardware. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1451–1459. ACM, 2012.

[123] John Russell. Processor diversity on the rise, reports intersect360. *HPC Wire*, November 2015. URL http://www.hpcwire.com/2015/11/12/processor-diversity-on-the-rise-reports-intersect360/.

[124] M. Salloum and V.J. Tsotras. Efficient and scalable sequence-based xml filtering. *Proc. of 12th WebDB*, 2009.

[125] Ziad Sehili, Lars Kolb, Christian Borgs, Rainer Schnell, and Erhard Rahm. Privacy preserving record linkage with ppjoin. In *BTW*, pages 85–104, 2015.

[126] Dennis Shasha, JT-L Wang, Kaizhong Zhang, and Frank Y Shih. Exact and approximate algorithms for unordered tree matching. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):668–678, 1994.

[127] Philip N Shilane, Grant R Wallace, and Mark L Huang. Preferential selection of candidates for delta compression, February 16 2016. US Patent 9,262,434.

[128] Lila Shnaiderman and Oded Shmueli. A parallel twig join algorithm for xml processing using a gpgpu. In *ADMS@ VLDB*, pages 45–56, 2012.

[129] Parijat Shukla and Arun Somani. A scalable record linkage technique for nosql databases using gpgpu. *in NoSQL: Database for Storage and Retrieval of Data in Cloud*, pages 119–142, 03 2017.

[130] Parijat Shukla and Arun K Somani. Context-aware duplicate detection in semi-structured data streams. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 216–223. IEEE, 2014.

[131] Parijat Shukla and Arun K Somani. Tree matching using data shaping. In *Big Data (BigData Congress), 2015 IEEE International Congress on*, pages 166–173. IEEE, 2015.

[132] Mike Steel and Tandy Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree. *Information Processing Letters*, 48(2):77–82, 1993.

[133] Fabian Suchanek and Gerhard Weikum. Knowledge harvesting in the big-data era. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 933–938. ACM, 2013.

[134] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization, 2002.

[135] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215. ACM, 2002.

[136] Andrea Torsello and Edwin R Hancock. Computing approximate tree edit distance using relaxation labeling. *Pattern Recognition Letters*, 24(8):1089–1097, 2003.

[137] Nam-Luc Tran, Sabri Skhiri, Esteban Zim, et al. Eqs: An elastic and scalable message queue for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 391–398. IEEE, 2011.

[138] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. zdelta: An efficient delta compression tool. 2002.

[139] Dimitre Trendafilov, Dimitre Trendafilov, Nasir Memon, Nasir Memon, Torsten Suel, and Torsten Suel. zdelta: An efficient delta compression tool. Technical report, 2002.

[140] J. D. Ullman. The tree data model. Accessed: June 6, 2016. URL http://infolab.stanford.edu/?ullman/focs/ch05.pdf.

[141] UWXMLRepository. UW XML Data Repository, Accessed: 2013-09-30. URL http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

[142] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[143] Leslie G Valiant. A bridging model for multi-core computing. In *Algorithms-ESA 2008*, pages 13–28. Springer, 2008.

[144] J. Van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. Xml accelerator engine. In *The First Int. Workshop on High Performance XML Processing*, 2004.

[145] Bill Vorhies. 4 things you need to know about the growth of big data. Accessed: Jan 2014. URL http://data-magnum.com/?p=452/.

[146] H. Wang, S. Park, W. Fan, and P.S. Yu. Vist: a dynamic index method for querying xml data by tree structures. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121. ACM, 2003.

[147] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. Entity matching: How similar is similar. *Proceedings of the VLDB Endowment*, 4(10):622–633, 2011.

[148] Melanie Weis and Felix Naumann. Dogmatix tracks down duplicates in xml. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM, 2005.

[149] Christopher G. Willard, Addison Snell, and Michael Feldman. Hpc application support for gpu computing. *Intersect360 Research*, November 2015. URL http://www.intersect360.com/industry/reports.php.

[150] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *ACM SIGPLAN Notices*, volume 48, pages 57–68. ACM, 2013.

[151] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara,

CA, July 2015. USENIX Association. URL https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/xia.

[152] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. Dare: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Transactions on Computers*, 65(6):1692–1705, 2016.

[153] XML. Accessed: Jan 2014. URL http://www.w3.org/XML/.

[154] Jianyun Xu, Andrew H Sung, and Qingzhong Liu. Tree based behavior monitoring for adaptive fraud detection. In *Pattern Recognition, 2006. (ICPR). 18th Int. Conf. on*, volume 1, pages 1208–1211. IEEE, 2006.

[155] Zhiwei Xu and Kai Hwang. Early prediction of mpp performance: the sp2, t3d, and paragon experiences. *Parallel Computing*, 22(7):917–942, 1996.

[156] Qing Yang and Jin Ren. Pre-cache similarity-based delta compression for use in a data storage system, February 6 2012. US Patent App. 13/366,846.

[157] Xuchen Yao, Benjamin Van Durme, Chris Callison-Burch, and Peter Clark. Answer extraction as sequence tagging with tree edit distance. In *HLT-NAACL*, pages 858–867. Citeseer, 2013.

[158] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing, 2011.

[159] Hang Zhang, Xuhao Chen, Nong Xiao, and Fang Liu. Architecting energy-efficient stt-ram based register file on gpgpus via delta compression. In *Proceedings of the 53rd Annual Design Automation Conference*, page 119. ACM, 2016.

[160] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. on Computing*, 18(6):1245–1262, 1989.

[161] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information processing letters*, 42(3):133–139, 1992.

[162] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *14th USENIX Conf. on File and Storage Technologies (FAST 16)*, pages 111–124, 2016.

[163] Canwei Zhuang and Shaorong Feng. Full tree-based encoding technique for dynamic xml labeling schemes. In *Database and Expert Systems Applications*, pages 357–368. Springer, 2012.