Parallelism in Querying and Storage for Large XML and Graph Databases

Shnaiderman Lila
Parallelism in Querying and Storage for Large XML and Graph Databases

Research thesis

In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Shnaiderman Lila

Submitted to the Senate of the Technion - Israel Institute of Technology

Sivan, 5775 Haifa May, 2015
This Research Thesis Was Done Under the Supervision of Prof. Oded Shmueli
in the department of Computer Science.

I would like to express my deepest gratitude to my advisor, Prof. Oded Shmueli, for his invaluable support and guidance throughout the research. He has been my mentor during the whole PhD thesis and supported me throughout all the stages of the thesis with his long experience in research and an endless patience. He invested in me immeasurable amount of time. I am indebted to him more than he knows.

I would also like to thank my husband, Arie to which this work is dedicated, for his love, support and understanding. Arie stood by me every step of the way and encouraged me continuously during these years.

Furthermore, I would like to thank to my parents, that gave me their support each time I needed it, and to my kids that encouraged me to continue with the hard work.

The Generous Financial Help Of the Technion Is Gratefully Acknowledged
# Table of Contents

Table of Contents  
Abbreviations  
Abstract  
  1 Introduction  
    1.1 First Phase of the Research  
    1.2 Second Phase of the Research  
    1.3 Third Phase of the Research  
    1.4 Contributions  
    1.5 Related Work  
    1.6 Thesis Organization  
  2 Background  
    2.1 Approaches for Storing and Managing XML Documents  
    2.2 Twig Pattern Matching  
      2.2.1 XML L-Stream Storage Schemes  
      2.2.2 Basic XML L-Stream Storage Scheme  
      2.2.3 XML (Extended) L-Stream Scheme  
    2.3 Approaches for Partitioning a Single XPath Query  
    2.4 Overview of Parallel Computation Complexity  
    2.5 Complexity of Processing XML Documents  
    2.6 PathStack and TwigStack  
    2.7 Graphics Processors (GPUs)
List of Figures

1.1 Example of a twig pattern ........................................... 6
1.2 Example of a TPQ expressed pattern ............................... 16
2.1 Example of twig expression ......................................... 34
2.2 L-Stream storage scheme example ................................. 63
2.3 Core XPath Grammar .................................................. 64
2.4 Wadler Fragment, changed and new rules .......................... 64
2.5 pWF changed and new rules ....................................... 64
2.6 Combined complexity of XPath. An arrow from $L_1$ to $L_2$ means that language $L_1$ is a fragment of language $L_2$. Taken from [43] .............................................. 65
2.7 Compact encoding of answers using stacks ........................ 66
2.8 The PathStack Algorithm Sketch .................................... 66
2.9 The TwigStack Algorithm Sketch ................................... 67
2.10 GPU architecture model ............................................. 67
3.1 Building the Full Tree from 3 Subtrees (with roots $rt_1, rt_2, rt_3$) 72
3.2 Source Tree ............................................................ 73
4.1 Results of PPS with one node solutions on small documents with sizes 10MB, 20MB, 30MB for path patterns 1 and 3 .... 96
4.2 Results of PPS with one node solutions on larger documents with sizes 112MB, 203MB, 524MB for path patterns 1 and 3 96
4.3 Results of PTS with full path solutions on documents with sizes 10MB, 20MB, 30MB, 112MB, 204MB, 524MB for twig pattern 1 ....................................................... 97
4.4 Results of PTS with one node solutions on documents with sizes 10MB, 20MB, 30MB for twig patterns 1 and 2 ............ 97
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Results of PTS with one node solutions on documents with sizes 112MB, 204MB for twig patterns 1 and 2</td>
</tr>
<tr>
<td>4.6</td>
<td>Results of PTS with one node solutions on 524MB document for twig pattern 2, while changing the <code>numOfTasks</code> parameter</td>
</tr>
<tr>
<td>4.7</td>
<td>Experiments with the DBLP database (PTS)</td>
</tr>
<tr>
<td>4.8</td>
<td>Experiments with the DBLP database (PPS)</td>
</tr>
<tr>
<td>4.9</td>
<td>Experiments with TSGeneric+ as the base search algorithm</td>
</tr>
<tr>
<td>4.10</td>
<td>Experiments with Treebank documents for PTS. X MB-Y chart line indicates that the experiment was performed with query number Y for the X-sized document</td>
</tr>
<tr>
<td>4.11</td>
<td>NMMR scheme experiments with very large documents for PTS with Twig Pattern 1. X GB chart line name means that the experiment was performed for the X GB-sized document</td>
</tr>
<tr>
<td>4.12</td>
<td>NMMR scheme experiments with very large documents for PTS with Twig Pattern 2. X GB chart line name means that the experiment was performed for the X GB-sized document</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of a query tree $qTree$ and a data tree $dTree$. All edges in $qTree$ are ancestor–descendant. $qTree$ nodes are indexed 1...6</td>
</tr>
<tr>
<td>5.2</td>
<td>The first phase of the GPU-Twig Algorithm</td>
</tr>
<tr>
<td>5.3</td>
<td>The second phase of the GPU-Twig Algorithm</td>
</tr>
<tr>
<td>5.4</td>
<td>The state of the $dTree$ after processing the first phase of the algorithm for all the leaves of $qTree$. For example, bit 6 in node $(13:20,2)$ is set to 1, i.e., node $(13:20,2)$ is subtree-correct w.r.t. the query node with index 6</td>
</tr>
<tr>
<td>5.5</td>
<td>Results of GPU-Twig on documents with different sizes, for twig pattern 1</td>
</tr>
<tr>
<td>5.6</td>
<td>Results of GPU-Twig on documents with different sizes, for twig pattern 3</td>
</tr>
<tr>
<td>5.7</td>
<td>Results of GPU-Twig on documents with different sizes, for twig pattern 5</td>
</tr>
<tr>
<td>5.8</td>
<td>Results of GPU-Twig on document with size 524MB, for different twig patterns</td>
</tr>
<tr>
<td>5.9</td>
<td>Results of GPU-Twig on document with size 1GB, for different twig patterns</td>
</tr>
<tr>
<td>5.10</td>
<td>Results of GPU-Twig on document with size 2GB, for different twig patterns</td>
</tr>
</tbody>
</table>
5.11 Results of GPU-Twig on the DBLP database document with size 670MB, for different twig patterns . . . . . . . . . . . . . 148
5.12 Summary of GPU-Twig speedup in comparison to PTS with 12 threads, for different twig patterns . . . . . . . . . . . . . 148

6.1 Example of a query tree \textit{qTree} and a data graph \textit{dGraph} . . . 156
6.2 The base GGQ algorithm . . . . . . . . . . . . . . . . . . . 160
6.3 The first extension of the GGQ algorithm . . . . . . . . . . 164
6.4 Example of limited Query . . . . . . . . . . . . . . . . . . . 168
6.5 The second extension of the GGQ algorithm . . . . . . . . . 175
6.6 Query phase ordering algorithm . . . . . . . . . . . . . . . . 176
6.7 Outline of Representative TPQ examples . . . . . . . . . . . 179
6.8 GLS database example . . . . . . . . . . . . . . . . . . . . . . 190
## List of Tables

4.1 Parsing time and size in main memory of the tested documents 107

6.1 Results of GGQ on a document with size 125MB (GRR dataset),
for path queries with 4 nodes (query 1 and query 2). The right
two columns contain the speedup of GGQ run in comparison
to a Gremlin run ............................... 184

6.2 Results of GGQ on a document with size 600MB (GRR dataset),
for a path query with 4 nodes (query 3) and a tree query with
5 nodes (query 6). The right two columns contain the speedup
of GGQ run in comparison to a Gremlin run .......... 184

6.3 Results of GGQ on a document with size 40MB (Census dataset),
for a path query with 4 nodes (query 4) and a tree query with
5 nodes (query 5). The right two columns contain the speedup
of GGQ run in comparison to a Gremlin run .......... 184

6.4 Results of GGQ on a document with size 180MB (Geospecies
data set), for a path query with 4 nodes (query 7) and a tree
query with 6 nodes (query 8). The right two columns contain
the speedup of GGQ run in comparison to a Gremlin run . . . 185

6.5 Results of GGQ on a document with size 180MB (Geospecies
data set), for two different tree queries with 11 nodes (query 9
and query 10). The right two columns contain the speedup of
GGQ run in comparison to a Gremlin run .......... 185

6.6 Results of GGQ on small documents (with size 4.5MB and
1.6MB from Geospecies dataset), for two different tree queries
with 11 nodes (query 9 and query 10). The right two columns
contain the speedup of GGQ run in comparison to a Gremlin
run ............................................. 185
6.7 Results of a $GPU - full$ GGQ run on documents with different sizes (from Geospecies dataset), for two different tree queries with 11 nodes (query 9 and query 10), while stopping right after finding the first answer . . . . . . . . . . . . . . . . . . . . 186
Abbreviations

DS Document Size
DTD Document Type Definition
OODB Object Oriented Data Base
SQL Structured Query Language
TPA Time Proximity Algorithm
XML Extensible Markup Language
DBMS Data Base Management System
Abstract

XML is based on a tree-structured data model. Naturally, the most popular XML querying language (XPath) uses patterns of selection predicates on multiple elements related by a tree structure. These are abstracted by *twig patterns*. Finding all occurrences of such a twig pattern in an XML database is a core operation for XML query processing.

Lately, large amounts of data are modeled and stored as graphs in order to express complex data relationships and for allowing machine learning methods to easily use the stored information. As a result, query processing on graph structures is becoming an important component in real-world applications. The most commonly used query format is that of tree pattern queries.

Due to the continuous growth of data, the efficiency of database operations becomes of paramount importance. Parallelism is the approach most commonly used to improve efficiency. Parallelism may be realized by using Multi-core-based processing. GPGPU (General Purpose Graphics Processing Unit) computing is an innovative approach that may be applied as well for realizing massive Single Instruction Multi-Data (SIMD) parallelism.
The subject of this research is parallelism in querying and storage for large native XML databases and Graph databases using different parallel platforms (multi-core CPUs and GPUs). In this research I have designed and implemented four different algorithms.

The first two algorithms are Parallel Path Stack algorithm (PPS) and Parallel Twig Stack algorithm (PTS). PPS and PTS are efficient algorithms for matching XML query twig patterns in a parallel multi-threaded computing platform. PPS and PTS are based on the PathStack and TwigStack algorithms [26]. These algorithms employ a sophisticated search technique for limiting stream processing to specific subtrees. I have also designed and implemented a novel parallel scheme for running PPS and PTS on Non-Main-Memory-Resident (NMMR) documents.

The third algorithm is GPU-Twig, for matching twig patterns in large XML documents, using a GPU. GPU-Twig uses the data and task parallelism of the GPU to perform memory-intensive tasks whereas the CPU is used to perform I/O and resource management. GPU-Twig works in two phases. In the first phase it goes over the relevant (to the input query) data of the data tree, and builds some helper information for each node. In the second phase it exploits the helper information that was created in the first step to detect the answers to the input query.

The fourth algorithm is GGQ (GPU Graph Data Base Query), for processing tree pattern queries on graph databases, using a GPU. As in GPU-Twig, GGQ uses the GPU data and task parallelism to perform memory-intensive
tasks whereas the CPU is used to perform I/O and resource management. GGQ is novel in that processors identifiers (IDs) are used to determine the choices made in matching the tree pattern to actual database graph nodes and edges. As the space of possibilities that can be represented by an ID is limited, methods are presented to practically increase this space.

I conducted extensive experimentation with all these algorithms using known and especially constructed benchmarks. I compared PPS and PTS to the standard (sequential) PathStack and Twig Stack algorithms in terms of run-time (to completion). I present the results of an extensive experimentation of the GPU-Twig algorithm on large XML documents using the DBLP and XMark benchmarks. I experimented with the GGQ algorithm on different graph databases. The experiments stress-test the cases in which the ID space is limited. Experimental results indicate that all the above algorithms reduce the running time of queries in comparison to other relevant algorithms under various settings and scenarios.
Chapter 1

Introduction

XPath, the XML Path Language [31], is a query language for selecting nodes from an XML [98] document. The XPath language is based on a tree representation of the XML document, and provides the ability to navigate within the tree, selecting nodes by a variety of patterns. For example, consider the following XPath expression: "movie[name = 'Harry Potter']//actor[first='Daniel' AND last = 'Radcliffe']". This expression matches 'actor' elements that (i) have a child element named 'first' with content 'Daniel', (ii) have a child element 'last' with content 'Radcliffe', and (iii) are descendants of 'movie' elements that have a child 'name' element with content 'Harry Potter'. This expression can be represented as a node-labelled tree called a twig pattern. The result of the above matching between the given twig pattern and the XML document is the set of all the elements in the XML document with an 'actor' label which satisfy the matching described above
Finding all occurrences of a twig pattern in an XML database is a basic operation in XML query processing. Twig patterns represent a subset of the XPath language (see Chapter 2.2).

Figure 1.1: Example of a twig pattern

There have been many research efforts aimed at parallelizing relational DBMS query processors [51, 83, 81, 28]. This is due to the emergence of multi-core processors. One example is InfiniteDB, which is a PC-Cluster-based Parallel Massive Database Management System [59].

A multi-core processor (or chip-level multiprocessor, CMP) combines two or more independent cores into a single package composed of a single integrated circuit (IC), or additional IC’s packaged together. A system with ’n’ cores is effective when it is presented with ’n’ or more threads concurrently [94, 20, 71].

The performance gained by the use of a multi-core processor depends on the
problem being solved and the algorithms used, as well as their software implementation. That is, the software has to be designed to take advantage of the available parallelism. If the software is not so designed, multi-core hardware may not contribute any significant speed-up. Most application software is not programmed to intensively take advantage of multiple concurrent threads because of the many challenges involved. Programming truly multithreaded applications often requires complex coordination of threads which can easily introduce subtle and difficult-to-find bugs. These bugs are usually related to the interleaved processing of data that is shared among threads.

As XPath is a critical component in many XML-based applications, it is essential to improve its processing performance as much as possible. There are many works dealing with optimizing the sequential performance of a single query [40]. Parallel query processors have become relevant and important as most modern computers have multi-core processors. Currently, most modern computers support two or more quad-core processors, i.e., eight cores, each of which can potentially run two hardware threads. Most XPath processors, for example Apache’s Xalan, support numerous XPath queries which are processed concurrently by different threads against the same query processor instance. Still, each XPath query is executed *serially*. By processing XPath queries concurrently, the overall latency is improved, but to enhance the overall performance one needs to increase the speed of processing a single XPath query via parallelization. This is particularly important for large documents. Parallelization of XPath queries is also fundamental for parallelizing various
host languages, such as XSL [100] and XQuery [99].

1.1 First Phase of the Research

The problem I address first is how to speedup the processing of twig queries, an important subset of the XPath language, within a multi-core architecture. Unlike much prior work, I deal with parallelizing the execution of a single XPath query. The space of possible algorithms and associated storage structures, and indices, is enormous. To start off effectively, I focus on a very prominent scheme for XML document representation, namely L-Streams (described in Chapter 2.2.1) and two basic algorithms (PathStack and TwigStack [26]), and I base my approach on data partitioning, a fundamental approach to parallelization. L-Streams here are sequences of XML elements having the same label, and there is such a sequence per each label of the document. L-Streams should not be confused with general data streams. There are many published variations and improvements on PathStack and TwigStack. However, as I demonstrate, any of these variations is ”pluggable” to my scheme, taken more or less as a black box. So, if an improved version of TwigStack is used, the performance of my algorithm will also improve. Therefore, for testing, I used the original TwigStack (the most accessible version). For validation, I also implemented one of the known improvements over TwigStack, TSGeneric+, and ran tests using it. Any other variation would have done as well.
I present the Parallel Path Stack algorithm (PPS) and the Parallel Twig Stack algorithm (PTS), novel parallel algorithms for processing a single twig pattern in a L-Stream environment. PPS and PTS are based on the PathStack and TwigStack algorithms [26]. TwigStack is a basic algorithm for XML documents that utilizes the L-Stream representation. The PathStack and TwigStack algorithms are designed so that no large intermediate results are created. The reason I chose these particular algorithms is that they are very well understood and effective, and therefore can provide a good vehicle to demonstrate the power of parallelism in the context of XML querying.

My basic idea for parallelizing twig pattern query processing is based on processing different subtrees of the document tree in parallel. However, to effectively parallelize PathStack/TwigStack I need to somehow know how the L-Streams are distributed among the different subtrees of the document tree. If I know the L-Streams distribution among the different subtrees of the entire document tree, I can evenly divide the work, where each thread will process a distinct subtree (or subtrees) of the entire tree. For that aim, I create a virtual partition of the document tree to subtrees (without changing the data storage placement). Then, while processing a query \( q \) on some subtree \( st \), for each L-Stream \( s \) that ”participates” in \( q \), I compute the start and end indices in L-Stream \( s \) of the sub-L-Stream of \( s \) that consists of nodes belonging to subtree \( st \) (see Chapter 3).

The parallel algorithm is based on a sequential algorithm (the base algorithm) and different base algorithms are ”pluggable”. In the parallel run, the base
algorithm is run on each subtree separately. Interestingly, it turns out that on trees in which there are either few query answers or in which answers are distributed non-uniformly, the single thread parallel algorithm is often much more efficient than the base algorithm run sequentially over the whole data tree. This has to do with skipping over substantial portions of L-Streams that occur "naturally" in the parallel algorithm.

TwigStack is I/O and runtime optimal among all sequential algorithms that read the entire input. It is linear in the sum of the sizes of the input lists and the final result list (see [26]). The sizes of intermediate results are also linear in the sum of sizes of the input list (i.e., the sizes of the relevant L-Streams) and the final result list. Recall that PTS and PPS are based on a method of partitioning the document tree to subtrees, so that each thread runs on a different subtree. The work performed by each thread preserves the complexity characteristics of TwigStack, and subject to uniform distribution assumptions, PTS and PPS are optimal among all parallel algorithms in terms of I/O and parallel runtime.

I conducted extensive experimentation with PPS and PTS. For that, I constructed a simple experimental system. This system contains a document parser (based on SAX [16]), and a query processor that is based on the PTS and PPS algorithms. The parser parses the document into an extended L-Stream representation. I compared PPS and PTS to the standard (sequential) PathStack and TwigStack algorithms in terms of runtime (to completion). I examined their performance for varying numbers of threads. Exper-
imental results indicate that using PPS and PTS significantly reduces the respective runtime of queries in comparison with the PathStack/TwigStack algorithms.

Next, I propose a novel scheme for working with Non-main-memory-resident (NMMR) documents. I implemented the scheme, and performed extensive experimentation. I found that this scheme provides significant improvement over the sequential naive algorithm. This scheme may be applied to very large, Terabyte sized, documents, an important special case of NMMR documents.

1.2 Second Phase of the Research

The important problem that I addressed in the second phase of my research is how to use GPUs to speedup twig query processing of tree-structured databases. Lately, there has been much research on using GPUs to speedup database operations [44]. The original use of GPUs, also known as video cards, was to render graphical information. GPUs are a cheap and ubiquitous source of processing power, as at least one GPU can be found in almost any computer. This has led to growing interest in finding methods to apply this power to non-graphical tasks through application programming frameworks, such as NVIDIA’s CUDA (Compute Unified Device Architecture). NVIDIA’s CUDA allows programmers to easily express and execute general purpose programs on NVIDIA GPUs. As a result, GPUs emerged as powerful co-
processors used, for example, in scientific computing [45], databases [52, 72] and other applications.

There are two factors that complicate GPU-based algorithms design. The first is that the GPU is a co-processor of the CPU and not a "first class citizen" processor. Thus, when running a task on the GPU, the data that is needed for processing the task has to be transferred from the CPU to the GPU and the results have to be transferred back to the CPU. These transfers are costly operations. The second is that parallel algorithms need be designed differently for GPUs than for multi-core systems. The main difference between these two platforms is that GPUs follow a SIMD (Single Instruction, Multiple Data, or as SIMT, T stands for thread) architecture, while multi-core systems follow a MIMD (Multiple Instructions, Multiple Data) architecture. In SIMD, multiple processing elements perform the same operation on multiple data elements, simultaneously. As any algorithm for GPUs has to fit the SIMD scheme, an original CPU (or multi-core) algorithm almost always should not be run as is on a GPU. If run as is, it will very likely be extremely inefficient.

The third algorithm that I designed and implemented, during the second phase of my research, is the GPU-Twig algorithm. The main idea underlying this algorithm is to copy the relevant parts of the document, according to the input query, to the GPU, to process the input query on the GPU, and to copy the query results back to the CPU. The key to parallelizing query processing, while using a stream representation scheme, is in the ability to
expose some of the structural characteristics of the document tree within
the streams. This is achieved by broadening the "classic" stream scheme,
such that each node in a L-Stream contains additional information about its
ancestors.

The core GPU-specific features that are used are as follows. The first is
the massive parallelization (a few orders of magnitude in terms of number of
processors). The second is utilizing the GPU’s high bandwidth in transferring
data from its global memory to the processors’ local memories, which is
more efficient than data transfer from main memory to the CPU. Previous
work dealt with possible strategies for parallelizing XPath queries [24]. To
the best of my knowledge, GPU-Twig is the first parallel algorithm that
exploits GPUs to accelerate the processing of a single query using the L-
Stream representation of documents.

I conducted extensive experimentation of the GPU-Twig algorithm. For
experimenting with this algorithm, I have used the simple experimental
document parser (based on SAX) to parse the document into an extended
L-Stream representation. I have implemented a query processor that runs
partly on the CPU and partly on the GPU. I compared GPU-Twig to the
Xalan [8] processor and to the sequential TwigStack algorithm [26] in terms
of runtime (to completion). The PTS algorithm, a parallel version of the
TwigStack algorithm for a multi-core CPU (that was designed during first
phase of my research), was also compared. Experimental results indicate
that using the GPU-Twig algorithm significantly reduces the running time.
It speeds up by up to 60 times in comparison with the sequential TwigStack algorithm and by up to 25 times in comparison to the PTS algorithm (not counting the copying times to and from the GPU). In experiments against Xalan, I found that using the same queries that I used in experiments with GPU-Twig, for documents larger than 50MB, Xalan runtime for processing one query, can not practically be measured: it did not finish (after executing for an hour, I stopped the run).

There exist technical limitations to current GPU technology that affect its usefulness for performing database functions. The two most relevant limitations are the GPU limited memory size and the host CPU to GPU global memory transfer time. Though each new GPU version, and future GPUs, will almost certainly have a larger memory, current NVIDIA cards have a maximum of 12GB, a fraction of the size of many databases. Transferring data blocks between the CPU and the GPU remains costly (on PCI Express 16x version 3, the bandwidth is 16 GB/sec). Despite these constraints, the actual query execution can be run concurrently over the GPU’s highly parallel organization, thus outperforming CPU query execution significantly. According to my experiments, even if the time of transferring data to and from the GPU (for documents of any size) is taken into account, GPU-Twig outperforms the best known parallel algorithm (PTS) [89, 87].

For databases that can fully reside in the global memory, I gain improvement of 16.5 times on average, in comparison to the PTS algorithm (counting the time of copying the results from the GPU to the CPU). Note that once
a document is loaded to the GPU global memory, a few queries on this document can be gathered together and processed one after the other, thus eliminating the need to separately copy the document, for each query, from the CPU to the GPU. For databases that can not fully reside in the GPU global memory, I still gain a significant improvement of 6.6 times on average in comparison to the PTS algorithm (counting the time of copying the data from the CPU to the GPU and the time of copying the results from the GPU to the CPU).

1.3 Third Phase of the Research

The third phase of my research deals with queries over Graph Databases. Graph Databases are widespread in many areas, including the semantic web and social/biological networks, as a graph is a more flexible and expressive structure than a tree. One of the most common ways of encoding and exchanging graph structures is that of using XML documents. XML documents model graphs by using ID/IDREFs. Another area that widely models its data as graphs is the semantic web. The languages that are used in encoding semantic web data are mainly RDF and RDFS. One of the most important and practically most interesting query formats for graph databases is a tree pattern query. I designed a simple language for expressing this queries, TPQs - Tree Pattern Queries. For example, consider the following TPQ expression: "movie {drama}* [{name} Miracle] {actor}* {name} Yosef Danovich".
I indicate an edge by using ‘{}’ and ‘*’ is used to denote an (un-named) element. The string that is written inside ‘{}’ denotes an edge label. I indicate a branch by using ‘[]’. The expression written inside ‘[]’ defines a tree branch. The above expression matches ‘movie’ labeled elements that have an edge labeled ‘drama’ that points to some node (name it x), such that x (i) has an edge labeled ‘name’ that points to some node with data ‘Miracle’, (ii) has an edge labeled ‘actor’ that points to some node (say y), and y has an edge labeled ‘name’ that points to some element with data ‘Yosef Danovich’. This expression can be represented as a node-labeled tree, as shown in Figure 1.2. The result of the matching between this TPQ and a graph database document is the set of all elements in the document with a ‘movie’ label which satisfy the matching condition described above. Finding all occurrences matching a TPQ, or specific data nodes corresponding to a specific TPQ node, in a graph document, are fundamental operations in graph query processing.

Figure 1.2: Example of a TPQ expressed pattern
In most known query languages for XML and RDF [96] (such as XQUERY and SPARQL [97]), many queries can be treated as TPQs on graphs. Basically, there are two common strategies for the abstraction and storage of XML documents, one is the relational method and the other is native XML. In the first method (relational graph), the document is represented as tables (relations). Then, a query is run on these tables by using logical operations [102]. The second method, on which I focus (native graph), stores the graph by using standard data structures for graph representation. In this case, the natural way to model queries is as tree (graphical) structures as well. Then, the problem of processing a query in native graph databases is that of pattern matching between the query that is represented as a tree, and the database that is represented as a (general directed) graph.

I focus on processing TPQ queries in native graph databases and not on general graph structured queries as, in practice, TPQ queries seem to be the most frequently used type. Also, TPQ queries can serve as building blocks for solving more general graph structured queries (as shown in Chapter 6). Although TPQs have been widely studied for many years [88], few of the proposed processing algorithms can be used to efficiently evaluate such TPQs over general graphs [102]. For example, techniques critical for the efficiency of processing TPQs, such as stack encoding and node skipping, appear mainly in the context of processing tree-structured data [26]. Few research projects have addressed parallelizing query processing over graph databases. The main idea has been to use the data partitioning strategy, i.e., methods to
partition the data between many computing elements, for example see [18]. To the best of my knowledge, there have been no attempts to parallelize the query processing of a single TPQ query, or to design a parallel algorithm that exploits GPUs (or any other SIMD-based device) to accelerate the processing of a single TPQ query.

The last outcome of the last phase of my research, is the GGQ algorithm (GPU-Graph database Query). The problem I address is how to use GPUs to accelerate the processing of a single TPQ query. The main idea underlying GGQ is to copy the relevant parts of the graph document, according to the input query, to the GPU global memory, to process the query using all the processors of the GPU in parallel, and to copy the query results back to the CPU memory. The key to parallelizing the query processing is in the ability to efficiently coordinate the query processing tasks between thousands of working units. In GGQ, each thread checks a different potential matching between the TPQ pattern and the data graph. In case that the checked potential matching actually exists, the thread reports this matching as one of the answers to the query.

GGQ is novel in that processor identifiers (IDs) are used to determine the choices made in attempting to match the tree pattern to actual database graph nodes and edges. As the space of possibilities that can be represented by an ID is limited, methods are presented to practically increase this space. To minimize the amount of data that has to be copied to the GPU for a particular query execution, I designed a new graph lists storage scheme,
GLS, that is based on a XML L-Stream representation scheme [26]. There is one node list for all data graph nodes that are not leaves, a different node list for leaf nodes of the graph, and a dedicated edge list for each edge label in the document. Each edge list contains data about the target nodes of the graph edges whose label matches the particular edge label of the list. For details see Chapter 6.7.

The main GPU-specific features that are utilized are as follows. The first is the massive parallelization (the number of processing units is a few orders of magnitude larger than in CPU-based systems). The second is using the GPU’s high bandwidth in transferring data from the global GPU memory to the local thread memories, which is more efficient than data transfer from main memory (RAM) to CPU processors.

Similarly to GPU-Twig, I conducted extensive experimentation with GGQ. I constructed a simple experimental system containing a graph document parser (based on SAX), and a query processor that runs partly on the CPU and partly on the GPU. The parser parses the document into a graph lists storage scheme representation. I compared GGQ to Gremlin [84] in terms of runtime (to completion). While Gremlin is not necessarily the most efficient tool in existence, Gremlin is the only query processor available for public use that I was aware of at the time of this effort that uses the native graph approach and that supports XPath-style queries over graph documents. Using Gremlin’s query language, one can easily express TPQs. This situation is expected to change as more work is done in this area.
The limitations to current GPU technology that affect its usefulness for performing database functions also affect this algorithm (the GPU limited memory size and the host CPU to GPU global memory transfer time). Overall, despite these limitations, the GPU execution significantly outperforms the CPU query execution.

For documents that can fully reside in the global memory, I gain speedup of up to 1000 times in comparison to Gremlin (counting the time of copying the results from the GPU to the CPU but not counting copying from the CPU to the GPU). If a whole document is loaded to the GPU, many queries on this document can be processed one after the other, thus eliminating the need to copy the document, for each query, from the CPU to the GPU. For documents that can not fully reside in the GPU global memory, according to my experiments, I still gain a significant improvement, up to 100 times, in comparison to Gremlin (counting the time of copying the data from the CPU to the GPU and the time of copying the results from the GPU to the CPU). In experiments with an extra-large query, I obtain speedup of up to 50 times in comparison to Gremlin (while counting just the copying time of the results from the GPU to CPU), and up to 35 times in comparison to Gremlin (while counting the time of copying the data from the CPU to the GPU and the time of copying the results from the GPU to the CPU).

As a future research direction, it will be interesting to compare GPU-Twig and GGQ on XML data.
1.4 Contributions

The initial objective of this work was to design fast parallel algorithms for querying XML and Graph database systems. This led to the PPS, PTS, GPU-Twig and GGQ algorithms. Some of the main contributions are as follows:

- I designed the PPS and PTS algorithms which, to the best of my knowledge, were the first algorithms for parallelizing the processing of a single query using the PathStack and TwigStack approaches.
- I implemented, and performed extensive experimentation with the PPS and PTS algorithms. My experimental evaluation demonstrates the effectiveness of PPS and PTS.
- I analyze PTS and show that I/O and runtime complexities of each thread run by PTS is optimal among all algorithms that read the entire input. Also, I showed that based on the assumption that the nodes in streams are distributed uniformly in the document tree, PTS is an optimal parallel algorithm.
- I designed a new scheme for working with very large, NMMR, files. For that aim I use a novel storage method, called Extended XML streams storage (EXSS).
- I designed GPU-Twig which, to the best of my knowledge, is the first GPU-based algorithm for processing a single XPath query over tree structured database.
• I implemented GPU-Twig, and performed an extensive experimentation of GPU-Twig against different relevant algorithms. Experimental evaluation demonstrates the effectiveness of GPU-Twig.

• I designed GGQ which, to the best of my knowledge, is the first GPU based algorithm for processing a single TPQ query over graph structured databases.

• I implemented GGQ, and performed an extensive experimentation of GGQ against Gremlin. Experimental evaluation demonstrates the effectiveness of GGQ.

• I designed a new graph lists storage scheme, GLS. I have also implemented a simple SAX based parser for RDF documents. The parser parses RDF documents into the GLS storage representation.

1.5 Related Work

The XPath language is a critical component in many XML-based applications, so it is imperative to maximize its performance. There has been extensive work on optimizing performance of a single XPath query by improving its traversal pattern [40, 42, 62].

There have also been research efforts aiming to parallelize relational DBMS query processors [63, 83, 81, 29]. This is mainly due to the emergence of multi-core processors. One example is InfiniteDB, which is a PC-Cluster-based Parallel Massive Database Management System [59]. Another example
is an extensive study of SQL query parallelization in the context of both distributed and centralized repositories [57, 58, 64].

Work regarding XPath parallelization have started just shortly before I started my PhD research. It was a new research direction as parallelization of SQL queries differs fundamentally from XPath parallelization.

Studies on parallelization of XML began to appear in the last ten years; for example, studies regarding parallel XML parsing [65, 80, 78, 79], parallel data placement in XML databases [93], and parallel query processors for XML databases [39, 66, 95].

Most existing XML processing engines are thread-safe and allow multiple threads to issue concurrent XPath queries against an XML document. Distributed XML processing is discussed in [27], [33]. Parallelization of a single XPath query using the Xalan XPath engine on shared address space multicore systems is presented in [24]. It evaluates opportunities for parallelizing a single XPath query, in a shared-address space environment, on multi-core processors. Given the constraints of the XML execution and data model, it proposes the following strategies for parallelizing individual XPath queries. 

Data partitioning approach: executes the same (sub)query on different sections of the same XML document. Query Partitioning approach: executes different (sub)queries on the same XML dataset. The third approach is the Hybrid Partitioning approach that combines both the data and query partitioning schemes. Continuation work is presented in [23]. It proposes a parallelization framework that determines the optimal way of parallelizing
an XML query. This decision is based on a statistics-based approach that relies both on the query specifics and the data statistics. At each stage of the parallelization process, three alternative approaches, namely, data-, query-, and hybrid- partitioning are evaluated. For a given XPath query, the parallelization algorithm uses XML statistics to estimate the relative efficiencies of these different alternatives and find an optimal parallel XPath processing plan. This research demonstrated that it is possible to accelerate XPath processing using commodity multi-core systems.

Zuo et. al. [103] propose a novel and complete optimization framework on parallelism for XML database multi threaded query processing based on the query partitioning approach for CMP systems (Chip-level Multi Processing). Firstly, a set of algorithms for constructing the parallel sub-query plans and partitioning the XML document to parallel sub-query plans are proposed. In order to reduce cache access conflicts and address the imbalance of threads workload, the granularity of the partitioned XML document is refined and the workload assignment is balanced by organizing working threads in pairs. Finally, by building the execution plan tree of sub-query plans constructed, the partial solution produced by parallel sub-query plans are merged into a final solution.

PPS and PTS follow the Data partitioning approach. PPS and PTS employ for running the same query on different sections of the streams in parallel. The main differences between Bordawekar et.al [23] and the current research on PPS and PTS are as follows.
1. Bordawekar et. al. use the Xalan off-the-shelf processor. I use a new query processor extending PathStack and TwigStack (which are L-Stream based).

2. PPS and PTS are able to "focus" on the relevant part of the streams, namely on the parts that match the subtree being processed by a thread.

3. I create a collection of tasks (subtrees) that are dynamically processed and thus perform dynamic load-balancing and not just a static one, as done in Bordawekar et. al.

4. I used the L-Streams storage scheme, whereas Bordawekar et. al. used a native XML storage scheme.

Machdi et. al. proposed general parallelizing techniques for holistic twig join algorithms to process queries against XML databases on a multi-core system [67, 68, 69]. These techniques are based on the Data partitioning approach and the Task Parallelism approach (i.e., decomposing the main algorithm into a few main tasks, which are then pipelined to realize parallelism). Machdi et. al. adopted a L-Stream based partitioning scheme as the basis of parallelism on multiple CPU cores. The XML data partitioning was performed in two levels. The first level was to form buckets for creating data independence and balancing loads among CPU cores; each bucket was assigned to a CPU core. Within each bucket, the second level of XML data partitioning was performed by creating finer partitions for providing finer
parallelism. Each CPU core performed the holistic twig join algorithm on each of its finer partitions in parallel with other CPU cores. In Task Parallelism, the holistic twig join algorithm was decomposed into two main tasks, which were pipelined to create parallelism. The first task adopted the data parallelism technique and its outputs were transferred to the second task periodically.

One of the basic ideas in [68] is to associate queries to documents. Then, to partition the associated documents and queries between different parallel components according to similarity between documents. The similarity is determined according to the workload in the system and other parameters. Another basic idea is to partition the XML document according to containment between different parts of the streams.

My research on PPS and PTS algorithms differs from Machdi et. al. as follows:

1. I present a different algorithms for partitioning the document and a different parallel running scheme.

2. The partition in my case is virtual, and is done only once while loading the document to the database. My algorithms do not require creating a new partition for each query as required by Machdi et. al.

3. My algorithms do not require dividing query processing into two tasks (such a division causes creation of redundant temporary results), due to the mechanism for solving the breaks some answer apart problem.
4. My algorithms adapt to very long, Terrabyte sized, documents.

5. My result shows better speedup (up to 18) in comparison to Machdi et. al. whose speedup is up to 3.6 \(^1\).

Another related work is [30]. They proposed the *Prefix Path Stream Storage Model*. The main idea of this model is that each L-Stream (in the regular L-Stream scheme) can be partitioned to a few streams according to the path from the root node to the current node. Based on this scheme, they changed the original PathStack/TwigStack algorithms to fit their new storage scheme. In [61], a parallel structural join algorithm (PSJ) for processing XPath queries in parallel is proposed. However, it is not efficient for XML twig queries. So, in [36], Feng el. al. proposed an efficient parallel PathStack algorithm, named P-PathStack, for processing XML twig queries. The algorithm first efficiently partitions the input element lists into multiple buckets, and then processes data in each bucket, in parallel. It is based on two ideas: the *Data partitioning approach*, and the *Task Parallelism* approach. The main difference between Feng et.al [36] and the current research on PPS and PTS is that I propose a scheme that allows to parallelize both algorithms, PathStack and TwigStack, while [36], proposes only parallelization for PathStack. Using PathStack and then merging its results to solve twig queries, causes the creation of large amounts of intermediate results in the PathStack phase that do not always contribute to the final answer. This is

\(^1\)As the environment of [68] is incomparable to our environment, I can only compare the speedup obtained by them
problematic due to the large memory that is required to hold these redundant results. Also, the merge phase processing time is significantly enlarged if these redundant results are part of the output of the PathStack phase.

As general purpose programming on GPUs emerged, they were mainly used to accelerate scientific, geometric, and imaging applications. State-of-the-art General Purpose GPU (GPGPU) techniques can be found in [77]. Lately, there are efforts to use GPUs to improve the performance of DBMSs. For example, in [92] the rendering and search capabilities of GPUs are used for spatial selection and join operations. In [17] GPU-based spatial operations are implemented as external procedures in a commercial DBMS. [47, 46, 44] present novel GPU based algorithms for relational operators. Recently, [60] implemented a similarity join on CUDA. There are also new framework proposals, such as Medusa, a programming framework for parallel graph processing on GPUs. Medusa enables developers to leverage the massive parallelism and other hardware features of GPUs by writing sequential C/C++ code for a small set of APIs. This framework uses known processing algorithms for GPUs, but does not propose algorithms of its own.

Graphs are widely used for representing data, thus a number of query languages for graphs have been proposed over the past few decades. First steps in this direction were taken in the 1980s, in areas such as hypertext systems [34]. In the 1990s, semistructured data [14] and object databases [49] became prominent, these provided fruitful areas for the study of graph models and query languages. In the last decade, the semantic web [82] and
social networks [35] have taken over as key areas amenable to graph-based approaches.

There are a growing number of initiatives to implement and commercialize Graph Databases, such as Neo4j [9], HyperGraphDB [6] and DEX [4] and many RDF solutions such as Jena [7] and AllegroGraph [1]. There are other initiatives to create graph querying languages that enable a simplified user view of querying such as SPARQL [97] and Gremlin [84]. Another initiative for a graph query language is GraphQL that is presented in [53].

In this work, the base node is a graph, so it deals with a graph of graphs, and proposes a flexible language for such a structure. Here, the answer to a query is a set of graphs. This work is not dealing with parallel query processing.

Recent works, [72, 91], propose efficient XML path processing algorithms using GPUs, which deal with path-based patterns. My research, on the other hand, deals with twig patterns, which are more complex query patterns. It also deals with TPQs, which are more complex query patterns applied on more complex (graph) database structures.

1.6 Thesis Organization

This thesis is organized as follows:

- Chapter 2 contains background.

- Chapter 3 describes how to partition L-Streams according to a given XML tree partition.
• Chapter 4 presents the PPS and the PTS algorithms, their experimental evaluation (4.4) and using these algorithms with very large, NMMR documents (4.5).

• Chapter 5 presents the GPU-Twig algorithm, its correctness proof (5.4) and its experimental evaluation (5.5).

• Chapter 6 presents the GGQ algorithm, its two extensions (6.3 and 6.4) and its experimental evaluation (6.6).

• Chapter 7 concludes and suggests future research directions.
Chapter 2

Background

2.1 Approaches for Storing and Managing XML Documents

XML [98] is in widespread use for both data and knowledge encoding. XML is a platform independent standard whose hierarchical nature is useful in modelling the real world, especially in cases where the data is hierarchical by nature. For example, major efforts were invested in migrating today’s genomic data to XML [25]. There are numerous works on languages and systems for efficiently accessing XML data [56, 37, 86, 19, 38].

There are two main approaches for storing and managing XML documents. The first approach maps an XML document to a relational table where each row represents an edge in the document’s XML tree [38]. Relational operators are used for traversing over XML stored documents. The second
approach, Native XML Storage, views an XML document as a tree. The entire XML tree is partitioned into distinct records containing disjoint connected subtrees \[56, 37, 86, 19\]. These records are stored on disk pages, either in an unparsed, textual form, or using some internal representation. In a "classical" relational database the disk organization essentially follows the relations' schemes. Partitioning of data to pages is the result of a data loading algorithms and the disk structures employed; for example, a loading algorithm for a \(B^+\) tree or a hash table. Keys are the main proximity factor affecting physical closeness of records.

In native XML Database systems, navigation over the document is strongly influenced by the document hierarchical structure. The main navigational tool is XPath (which can also use indexes). In addition to the XML augmented tree, there are also indices. Database systems use path indices that reduce the number of navigation steps across stored XML nodes. Thus, XPath processors for disk-resident data employ a mixed, i.e., part navigational, part indexed, processing model.

Lately, database and repository systems increasingly use the Native XML Storage approach. For example, the DB2 Universal Database? system is enhanced with comprehensive Native XML support \[73\]. The Oracle XML DB \[75\] also provides Native XML support. Another example is the effort to enhance SQL, so as to allow it access to natively-stored XML data, through traditional SQL interfaces (Relational Over XML - ROX scenario) \[50\]. One more example is Natix, a database management system, built from scratch,
for storing and processing XML data [37]. The Natix storage scheme also stores XML documents as tree structures.

In this research, when I use the term XML database I mean a forest of rooted, ordered, labelled trees, each node corresponds to an element or a value, and the edges representing (direct) element-sub element or element-value relationships. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes. An XML database is built from one or more XML documents. For simplicity of presentation, in the current work, the database is built out of one XML document. When I use the term document, I mean this XML document.

### 2.2 Twig Pattern Matching

A twig pattern is defined as a rooted, ordered, labelled tree whose nodes’ labels are either element tags or string values, and edges are either parent-child edges (depicted using a single line) or ancestor-descendant edges (depicted using a double line). I use the language of Bruno et.al. which is a subset of XPath.

A Path pattern is a special case of a Twig pattern in which each node has at most one child.

Consider for example, the XPath expression:

`actor[first='Daniel' AND last='Radcliffe']`”.

It matches actor elements that:
1. Have a child element named 'first' with content 'Daniel'.

2. Have a child element named 'last' with content 'Radcliffe'.

It can be represented as the twig pattern in Figure 2.1. Only parent-child edges are used in this example. In the XPath expression presented in the introduction (which is depicted in Figure 1.1), an ancestor-descendant edge is used between the movie element and the actor element.

![Figure 2.1: Example of twig expression](image)

Given a query twig pattern $Q$ and an XML database $D$, a match of $Q$ in $D$ is a mapping from nodes in $Q$ to nodes in $D$, such that:

1. Each query node maps to a distinct XML database node. My algorithms can be easily adjusted to a mode in which a few query nodes can map to the same data node. For clarity, I only present the distinct mode.

2. The structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes.
The answer to query $Q$ with $n$ nodes can be represented as an $n$-ary relation in which each tuple $(d_1, \ldots, d_n)$ consists of the database nodes that together identify a distinct match of $Q$ in $D$.

The Twig Pattern Matching problem is defined as follows. Given a query twig pattern $Q$ and an XML database $D$ that is represented via L-Streams, compute the answer to $Q$ on $D$ by listing all matches of $Q$ in $D$.

### 2.2.1 XML L-Stream Storage Schemes

There is a wide spread scheme to represent XML documents which follows the native XML Database approach, I call it *L-Stream representation*. The L-Stream representation extends the classic inverted index data structure used in information retrieval. Each L-Stream contains the positional representations of the database nodes whose labels are identical. There is a separate L-Stream for each element label in the document.

### 2.2.2 Basic XML L-Stream Storage Scheme

In the basic XML L-Stream scheme, the position of a string or element occurrence in the XML database is represented as a 3-tuple $(DocId, LeftPos:RightPos, LevelNum)$, where:

1. $DocId$ is the identifier of the document.
2. $LevelNum$ is the nesting depth of the element (or string value) in the document.
3. **LeftPos** and **RightPos** are the open and close indices of the element in the data tree ([26]). The open and close indices are assigned according to an order given by a DFS (Depth First Search) variation, as follows. The tree of the **DocId** document is scanned in DFS order. I use a variable called **counter** initialized to zero. When visiting some element \( n \), I increment **counter**, then assign its value to the **LeftPos** of \( n \), then scan the subtree of \( n \) (in DFS order). After finishing the scan of the subtree of \( n \) I increment **counter** and assign its value to **RightPos** of \( n \).

Let \( \text{lbl} \) denote a label in a database. Associated with each distinct \( \text{lbl} \) there is an **L-Stream** \( S_{\text{lbl}} \). The **L-Stream** \( S_{\text{lbl}} \) is defined as the ordered sequence of the positional representations of the database nodes having the \( \text{lbl} \) label. The nodes in the L-Stream are ordered by their \((\text{DocId}, \text{LeftPos})\) values. This scheme allows to easily deduce the structural relationship between any two nodes. For example, a parent-child relationship between nodes \( a \) (parent) and \( b \) (child) exists if \( b.\text{LeftPos} > a.\text{leftPos AND b.rightPos < a.rightPos AND b.levelNum = a.levelNum + 1} \), and an ancestor-descendant relationship between nodes \( a \) and \( b \) exists if \( b.\text{LeftPos} > a.\text{leftPos AND b.rightPos < a.rightPos} \). In this work I treat a single document, so from now on, I will omit **DocId** and use a 2-tuple \((\text{LeftPos:RightPos, LevelNum})\) instead.

Figure 2.2 presents the L-Stream storage scheme. The upper part of the picture depicts an XML document tree that has to be stored. The lower part shows how this document tree is stored in memory. Observe that each
label of the document nodes has a corresponding L-Stream. For example, the node with \((\text{LeftPos:RightPos, LevelNum})=(16:25,2)\) can be found in the 'actor' L-Stream.

A sequent is defined as a sequence of one element after the other without intervening or missing elements. A sequent sub-sequence of nodes in an L-Stream is said to be a \textit{Sub-L-Stream}. In the L-Stream scheme, each distinct label (i.e., data value) has a separate L-Stream. As there may be numerous leaf labels corresponding to a multitude of string values, this leads to a significant memory overhead. There are standard ways to overcome this problem. For example, to assign a single L-Stream for all leaf labels, called \textit{leaves-L-Stream}. The nodes in this leaves-L-Stream are sorted in a lexicographic order according to their label, and all the nodes with the same label are ordered according to the \textit{LeftPos} value of the node.

\subsection{XML (Extended) L-Stream Scheme}

For GPU-Twig, one of the algorithms that was designed during the research, I extended the L-Stream scheme, with additional structural information whose usefulness is explained in Chapter 5. The first additional field that is added to each node is a list \textit{ancStreamsL}. Let \(n\) be a node in the data tree, its \textit{ancStreamsL} list holds information about all the L-Streams that contain at least one node that belongs to the path between \(n\) and the root of the data tree (i.e., ancestors of \(n\)). The information that is held for each such L-Stream is as follows: the name of the L-Stream (i.e., label), and the index (i.e.,
position number) in the L-Stream of a node that belongs to the path between
\( n \) and the root which is closest to the root. The second additional field that is
added to each node is a Boolean array, \( qArray \) (see definition below). Its size
is equal to the number of nodes in the twig pattern. There is an assumption
that in twig patterns there are rarely more than 32 nodes, so a 32 bits \( qArray \),
which is integrated into the storage structure of the document, is sufficient;
should this not be the case, I can enlarge the \( qArray \) in advance to 64 bits or
more. The \( qArray \) field size is defined prior to starting the document parsing,
based on the predefined knowledge of the maximal possible number of nodes
in the twig patten. Thus, when the twig pattern is changed, I do not have
to change the database structure. For example, the following describes node
\( n \) in L-Stream \( str: "3 : 5, 4, < author : 3, book : 2 >, 0". Here, 3 : 5, 4 is the
original 3-tuple (LeftPos:RightPos, LevelNum), < author : 3, book : 2 > is the
ancStreamsL of node \( n \), which means that \( n \) has two types of ancestors,
the first with label \( author \) where the closest to the root \( author \) ancestor of \( n \)
is at index 3 of the \( author \) L-Stream, and the second with label \( book \) where
the closest to the root \( book \) ancestor of \( n \) is at index 2 of the \( book \) L-Stream.
The last field, 0, is the \( qArray \) field. It means all the bits of \( qArray \) are zero.
The extended L-Stream scheme consumes 4.2 times the amount of space that
is consumed by the basic L-Stream scheme; this is acceptable for real-world
applications as exhibited in the experiments.

An L-Stream is held in memory as an array of \textit{structs}. Each \textit{struct} contains
the above described data. This is a compact way to store information that
allows fast access to each node (especially if its index in the L-Stream is known) thus convenient for working with a GPU.

2.3 Approaches for Partitioning a Single XPath Query

In [24], Bordawekar et al. evaluate opportunities for parallelizing a single XPath query in a shared-address space environment on commodity multi-core processors. They assume that the XML document is prepared and can be concurrently accessed by multiple threads. The key aim of their study was understanding the challenges in parallelizing XPath queries using a real production-grade system.

Three new schemes for parallelizing XPath queries were proposed: (1) Data Partitioning, (2) Query Partitioning, and (3) Hybrid partitioning that combines both the data and query partitioning schemes. These approaches exploit the usually read-only nature of XPath processing. All three approaches achieve parallelism via partitioning traversals over the XML documents. The data partitioning approach executes the same (sub)query on different sections of the same XML document whereas the query partitioning approach executes different (sub)queries on the same XML data.

In the data partitioning approach, the input query is split into serial and parallel sub-queries. The serial part of the XPath query is executed by a single processor on the entire document. The resulting node set is then
distributed across multiple processors, e.g., by using a block-distribution. Each participating processor then uses the locally assigned node set as the context node set and executes the parallel sub-query on every local context node. This approach achieves parallelism by executing the same XPath query concurrently on different sections of the XML document. Thus, this approach follows the data parallel style of parallel processing.

In the query partitioning approach, the input query is rewritten into a set of sub-queries that can ideally navigate different sections of the XML tree. The number of sub-queries matches the number of participating processors. In many cases, each sub-query is an invocation of the original query using different parameters. Each processor executes its assigned sub-query on the entire XML data tree. The final result of the query can be then computed using either the union or merge of the per-processor resulting node sets. Unlike the data partitioning approach, this approach achieves parallelism via exploiting potentially non-overlapping navigational patterns of the sub-queries.

The data and query partitioning approaches can be used together to form a hybrid partitioning approach. In this approach, the input XPath query is first partitioned into different sub-queries for a set of virtual processors (query partitioning). Each virtual processor is a set of physical processors and it executes its assigned sub-query using the data partitioning approach.
2.4 Overview of Parallel Computation Complexity

This overview is based on [48, 43].

\( P \), \( LOGSPACE \), and \( NLOGSPACE \), denote the well-known complexity classes of problems solvable on Turing machines in deterministic polynomial time, deterministic logarithmic space, and nondeterministic logarithmic space, respectively.

A Parallel Random Access Machine consists of an unbounded collection of numbered RAM processors \( P_0, P_1, P_2, \ldots \) that run in parallel. They communicate via an unbounded collection of shared memory cells \( C_0, C_1, C_2, \ldots \). Each processor \( P_i \) has its own local memory and knows its own index \( i \). It also has instructions for read/write access from/to the shared memory.

The class \( P \) is the set of all languages \( L \) that are decidable in sequential time \( n^{O(1)} \), where \( n \) is the size of the input word.

The class \( NC \) is the set of all languages \( L \) that are decidable in parallel time \( (\log n)^{O(1)} \) and processors \( n^{O(1)} \), while \( n \) is the size of the input word in \( L \).

It is a widely-held conjecture that problems complete for complexity class \( P \) are inherently sequential and cannot profit from parallel computation. Instead, a problem is called highly parallelizable if it belongs to the complexity class \( NC \).

A simple model of parallel computation is that of Boolean circuits, which is an acyclic directed graph whose vertices compute elementary Boolean functions.
and its edges carry unidirectional logical signals. For the full definition see [48].

By a **monotone circuit**, I denote a circuit in which only the input gates may possibly be negated. All other gates are either \(\land\)-gates or \(\lor\)-gates (but not \(\neg\)-gates).

A **Boolean circuit family** \(a_n\) is a collection of \(n\) circuits, computing the same function \(f\) where circuit \(a_i\) computes \(f\) for \(i\) inputs.

Such a family is called **LOGSPACE-uniform** if there exists a LOGSPACE-bounded deterministic Turing machine which, on the input of \(n\) 1 bits (the string \(1^n\)), outputs the circuit \(a_n\).

For each \(i\), the class \(NC^i\) is the set of all languages \(L\), such that \(L\) is recognized by a LOGSPACE-uniform Boolean circuit family \(a_n\) with \(size(a_n) = n^{O(1)}\) and \(depth(a_n) = O((log n)^i)\).

**LOGCFL** is defined as the complexity class consisting of all problems which are LOGSPACE-reducible to a context-free language.

I have \(NC^1 \subseteq LOGSPACE \subseteq NLOGSPACE \subseteq LOGCFL \subseteq NC^2 \subseteq NC \subseteq P\). All inclusions \(\subseteq\) are suspected to be strict. \(P\), LOGCFL, and NLOGSPACE are closed under LOGSPACE-reductions.

### 2.5 Complexity of Processing XML Documents

In this section, I shortly overview the parallel complexity of the XPath query language, based on [43].
A brief introduction to XPath

XPath 1.0 is a language with a large number of features and therefore somewhat awkward for theoretical treatment. In this overview, only some of these features are introduced. For a detailed definition of the full XPath language, see [31], and for a concise yet complete formal definition of the XPath semantics, see [41].

Two basic fragments of XPath were defined. Core XPath, first defined in [41], supports the most commonly used features of XPath, path navigation and conditions with logical connectives, but excludes arithmetics, string manipulations, and some of the more esoteric aspects of the language. The second fragment, which was first discussed in [101], contains XPath’s logical and arithmetic features, but excludes string manipulations. It is referred to as the Wadler Fragment, WF.

Core XPath is discussed first. The syntax of Core XPath is defined by the grammar in Figure 2.3. "locpath" is the start production, "axis" denotes axis relations, and "ntst" denotes tags labelling document nodes or the star "*" that matches all tags ("node tests"). The main syntactical feature of Core XPath are location paths. Expressions enclosed in square brackets are called conditions or predicates. The main application of XPath is navigation in XML document trees. This is done using the axis relations. For natural binary relations such as "child" and "descendant" between nodes, their intuitive meanings are conveyed by their names (see [31, 41] for more details).
Positive Core XPath is defined be removing negation ("not") from the Core XPath fragment.

Positive (or parallel) XPath (pXPath) is obtained by imposing the following restrictions on XPath:

1. Expressions of the form $\mathcal{X} :: a[e_1]...[e_k]$ with $k \geq 2$ are not allowed, where $\mathcal{X}$ denotes an axis, $a$ is a node test and the $e_i$’s are XPath expressions.

2. The following functions may not be used: not, count, sum, string, and number as well as the string functions local-name, namespace-uri, name, string-length, and normalize-space.

3. Constructs of the form $e_1 \text{ RelOp } e_2$ where at least one of the expressions $e_i$ is of type Boolean, are forbidden.

4. The depth of nesting of arithmetic operators and of the concat-function is bounded by some given constant $L$. Moreover, without loss of generality, the arity of the concat-function is bounded by some constant $K$.

The syntax of the "Wadler Fragment" (WF) is defined by the Core XPath grammar with some extensions. The changed/new rules are presented in Figure 2.4. "expr" (rather than "locpath") is now the start production and "number" denotes constant real-valued numbers. XPath is mainly understood as a language for selecting a subset of the nodes of an XML document.
tree. Hence, XPath expressions are referred to as XPath queries. Note, however, that XPath query results can also be of different types, namely for the $WF$-numbers and Booleans (as well as character strings for full XPath). XPath expressions are evaluated relative to a context, which by definition is a triple of a context-node and two integers, the so-called context position and the context-size. For more details, see [31, 41].

**Definition:** Let $e$ be a XPath expression. The depth of negation of $e$ is bounded by 0 if and only if the $not$-function does not occur at all in $e$. For any $k > 0$, the depth of negation of $e$ is bounded by $k$ iff for any subexpression $not(e')$ occurring in $e$, the depth of negation of $e'$ is bounded by some $l$ with $0 \leq l < k$.

$pWF$ (positive $WF$) is obtained by restricting $WF$ in the following way:

1. Expressions of the form $\mathcal{X} :: a[e_1]...[e_k]$ with $k \geq 2$ are not allowed, where $\mathcal{X}$ denotes an axis, $a$ is a node test and the $e_i$'s are XPath expressions.

2. Negation may not be used.

3. The nesting depth of arithmetic operators is bounded by some constant $K$.

The first two restrictions above mean that the grammar defined above for Core XPath and for $WF$ has to be modified as described in Figure 2.5. Consider the example query $child :: a[position()+1 = last()]$. Relative to a context-triple $(v, i, j)$, $i$ and $j$ are ignored when the location step $child :: a$
selects those children of \( v \) that are labelled "a". Let \( w_1, \ldots, w_m \) be this set of
nodes, where the indices correspond to the relative order of the nodes in the
document. The application of an axis causes a change of context to which
the condition \([\text{position}(\cdot) + 1 = \text{last}(\cdot)]\) is applied. The condition is tried on
each of the triples \((w_1, 1, m), \ldots, (w_m, m, m)\). It will select all those nodes \( w_k \)
for which \( k + 1 = m \), that is, it will select the singleton \( w_{m-1} \).

The complexity results are presented for the following decision problem.
Given a XPath query \( Q \), a XML data tree \( t \), a context \( D \), and a result
\( r \), one has to decide whether evaluating the query \( Q \) over the data tree \( t \) for
the context \( D \) yields the result \( r \). If the result of a XPath query is inde-
pendent of a context (e.g., absolute location paths), the short-hand notation
\( Q(t) \) is used to denote the result of evaluating \( Q \) over the data tree \( t \).

Let \( C \) denote some complexity class and let \( X \) be some fragment of XPath.
If the above decision problem of evaluating XPath queries from the fragment
\( X \) is in the complexity class \( C \) (respectively, \( C \)-hard or \( C \)-complete), then ,
as a short-hand, it is said: "the fragment \( X \) is in \( C \)" (respectively, "\( C \)-hard"
or "\( C \)-complete").

**Complexity classes of different XPath fragments**

The rest of the section contains the complexity characterization of different
XPath fragments. All proofs can be found in [43].

*Combined complexity* refers to the complexity in terms of both the size of
the data and of the size the query.

- **Proposition:** The XPath query evaluation problem is in \( P \) with re-
spect to combined complexity ([41]).

- **Proposition:** Core XPath queries can be evaluated in time $O(|Q|^{|t|})$, where $|Q|$ denotes the size of the query and $|t|$ denotes the size of the XML data tree.

- **Theorem:** Core XPath is $P$-complete with respect to combined complexity.

- **Corollary:** Core XPath remains $P$-hard even if:
  
  1. the document tree is limited to depth three.
  2. only the axes child, parent, and descendant-or-self are allowed.

- **Theorem:** The combined complexity of positive Core XPath is in $LOGCFL$ (i.e., it is highly parallelizable).

- **Theorem:** The combined complexity of positive Core XPath queries augmented by negation with bounded depth is in $LOGCFL$.

- **Theorem:** Positive Core XPath is $LOGCFL$-hard (combined complexity).

- Let $PF$ be the fragment of Core XPath containing only the location paths, without conditions (i.e., no expressions enclosed in brackets are permitted).

  **Theorem:** $PF$ is $NLOGSPACE$-complete under $LOGSPACE$-reductions with respect to combined complexity.
• Let \( PF \) be denoted with the permitted axes restricted to the downward axes "child" and "descendant".

**Theorem:** \( PF \) is in \( LOGSPACE \) with respect to combined complexity.

• Let Core XPath\(^1\) be the restriction of Core XPath that supports only the axes self, child, and parent. Core XPath\(^1\) allows only step-expressions which can go to a node at distance at most one from the current node.

**Theorem:** Core XPath\(^1\) is in \( LOGSPACE \) with respect to combined complexity. This problem remains in \( LOGSPACE \) even if Core XPath\(^1\) is augmented by the "axes" previous-sibling and next-sibling.

• **Theorem:** \( pWF \) (see its definition above) is in \( LOGCFL \) with respect to combined complexity.

• *Iterated predicates*, are location steps of the form \( X :: a[e_1]...[e_k] \) with \( k \geq 2 \).

**Theorem:** The combined complexity of \( pWF \) queries extended by iterated predicates is \( P \)-complete.

• **Corollary:** The combined complexity of \( pWF \) extended by iterated predicates of the form \( X :: a[e_1][e_2] \) is \( P \)-complete.

• **Corollary:** The combined complexity of \( pWF \) queries augmented by negation with bounded depth is in \( LOGCFL \).
Theorem: $pXPath$ (see its definition above) is in $LOGCFL$ with respect to combined complexity.

Theorem: The combined complexity of $pXPath$ augmented by negation with bounded depth is in $LOGCFL$.

Figure 2.6 graphically presents the relationship between complexity class of different XPath fragments.

The problem I refer to is the Boolean version of Twig pattern matching (i.e. is there a match between some twig pattern $q$ and some data graph $G$). In fact, the definition of the twig pattern matching problem is identical to the definition of the FP fragment. An FP fragment problem can be solved in NLOGSPACE. Problems in NLOGSPACE are considered to be highly parallelizable. So, as Twig pattern matching is actually FP fragment, it is also highly parallelizable. Highly parallelizable means that the problem belongs to the complexity class NC. I.e., an efficient parallel algorithm can be found to solve this employing a polynomial number of processors and operating in poly-logarithmic time, both in the size of the input. Practically, it means that the problem may significantly benefit from a large number of cooperating processors beyond simply dividing the problem into sub-problems. Twig pattern queries can be effectively parallelized as exhibited experimentally in my research (see Sections 4.4 and 5.5), although there is no claim that my methods would have achieved poly-logarithmic time given sufficiently many processors. My algorithms were designed to be practical in light of hardware
system constraints.

2.6 PathStack and TwigStack

Let $q$ (with or without subscripts) denote a node in a twig pattern. Associated with each node $q$ there is an L-Stream $T_q$.

In PathStack and TwigStack, a stack $S_q$ is also associated with each query node $q$. Strictly speaking, with the label of node $q$. Each data node in the stack consists of a pair: (positional representation of a node from L-Stream $T_q$, pointer to a node in $S_{\text{parent}(q)}$). The set of stacks enables a compact encoding of partial and total answers to the query twig pattern, which can represent, in linear space, a potentially exponential (in the number of query nodes) number of answers to the twig pattern query. For example, Figure 2.7 illustrates the stack encoding of answers to a path query for a sample database. According to the algorithm, at each moment the algorithm handles one path in the data tree and one path in the twig pattern tree. In the example, the part of the data tree and the part of the twig pattern that are currently being processed are shown in bold.

2.6.1 PathStack

The key idea of Algorithm PathStack is to repeatedly construct (compact) stack encodings of partial and total answers to the query path pattern, by iterating through the L-Stream nodes in sorted order of their LeftPos values.
So, the query path pattern nodes will be matched from the query root down to the query leaf.

Figure 2.8 presents a sketch of the PathStack algorithm. The input to the algorithm are a streams database and a path pattern whose root is $q$. The output of the algorithm is the answers to the path query, which are presented as data paths from root to leaf according to $q$’s twig pattern. $eof$ in line 1 denotes End Of File. For a particular L-Stream, define the currently processed node as the first node in L-Stream order such that all nodes preceding it on the L-Stream have been processed already and it and none of the nodes following it have been processed. The function $getMin(q)$ in line 2 identifies the next query node to be processed. $getMin(q)$ looks in all the streams that correspond to the query nodes of the twig pattern rooted at $q$, and returns the query node associated with the L-Stream whose currently processed node, say $x$, has the smallest LeftPos value. In lines 3 and 4 all the stacks are cleaned, meaning that all the nodes that do not belong to the current path (from the root of the data tree to node $x$) are popped up from the stacks. In line 5 the currently processed node (i.e., node $x$ mentioned above) is copied from the L-Stream to the stack. While copying $x$ to the stack, a pointer is added from the copy of $x$ (on the stack) to the node that is currently on the top of the stack with the same labeled as the parent node of $qN$ (defined in line 2). This pointer helps to encode the answers of the query within the stacks. In lines 6-8 the currently found results are expanded from the stacks. For more details see [26].
Example: I use the source document tree that is presented in Figure 2.2, and the path query "Movie/First/Rupert".

The call starts with $q$ being the root of the query path, namely "Movie". According to line 2, the first query node that is chosen is "Movie" (namely $\text{getMin}(\text{Movie}) = "\text{Movie}"$), as node "Movie" has a data node with the smallest LeftPos value among all the unprocessed nodes in all the streams that belong to the query nodes in the query tree (the indices of the currently processed node in the "Movie" L-Stream are "1:26"). There is nothing to do in lines 3,4, as the stacks are empty. According to line 5 node "1:26" is copied to the stack with the name "Movie". Since, "Movie" has no parent, there is no need to install a pointer from this copy of "1:26". "Movie" is not a leaf, so (according to line 6) the algorithm loops to line 2.

The next query node that is chosen in line 2 is "First". It is chosen because the currently processed node in its L-Stream has indices "7:10", where 7 is the smallest LeftPos among all nodes of the streams corresponding to the query nodes. There is nothing to do in lines 3,4 as there is only one node in the stacks, and the currently processed node (node "First" with indices "7:10") is in the continuation of the data path from the root to node "1:26". According to line 5 node "7:10" is copied to the stack with the name "First", and set a pointer between this node and the node at the top of the "Movie" stack (the node with indices "1:26").

The next query node that is chosen in line 2 is "First". It is chosen because the currently processed node in its L-Stream has indices "17:20", where 17 is
the smallest LeftPos among all the nodes of the streams corresponding to the query nodes. According to lines 3, 4, the node with indices "7:10" is deleted from the "First" stack, as it does not belong to the path connecting the root of the data tree and the currently processed node which has indices "17:20". It is determined by the RightPos value of the currently processed node (20) that is larger than 10, the RightPos value of node "7:10". According to line 5 node "17:20" is copied to the stack with the name "First", and set a pointer between this node and the node at the top of the "Movie" stack (the node with indices "1:26").

The next query node that is chosen in line 2 is "Rupert". It is chosen because the currently processed node in its L-Stream has indices "18:19", where 18 is the smallest LeftPos among all the nodes of the streams corresponding to the query nodes. There is nothing to do in lines 3, 4, as the currently processed node (node "Rupert" with indices "18:19") is in the continuation of the path from the root to node "17:20" whose copy is currently on the stack. According to line 5 node "18:19" is copied to the stack with the name "Rupert", and set a pointer between this node and the node at the top of the "First" stack (the node with indices "17:20"). According to lines 6-8 the result that is currently encoded in the stack is expanded, as "Rupert" is a leaf node in the query. The result is: "Movie" with "1:26", "First" with "17:20", "Rupert" with "18:19". At this point all the nodes in the relevant streams were processed, so the algorithm terminates.
2.6.2 TwigStack

A straightforward way of computing answers to a twig pattern query is to decompose the twig into multiple root-to-leaf path patterns, use PathStack to identify solutions to each individual path, and then merge-join these solutions to compute the answers to the query. The merge join operation is needed, as the algorithm finds paths portions of the answers and not the full answers. The paths must be merged and joined to form the full twig answers. This approach, however, faces a fundamental problem: many intermediate results may not be a part of any final answer and are needlessly generated. The reason is that when this method look for the paths independently of my knowledge of the full twig pattern, this method would find many paths that do not have a common root. Consider some query twig rooted at the "Author" node and having 2 paths under it: "First Name/Jane" and "Last Name/Kuper". There can be a database in which this method find the following paths: "Author/First Name/Jane" and "Author/Last Name/Kuper" while the "Author" node is different for these paths. For example, in the hypothetic database (from which the fragment that is presented in Figure 2.2 could be possibly taken), there is also Jane whose last name is Fishman, and there is Kuper whose first name is Maya. These paths also will be found together with the correct paths, those which do participate in a final answer.

The main idea of Algorithm TwigStack is to intelligently choose the paths of the data tree. That is, choose only paths that participate in at least one final answer.
Figure 2.9 presents a sketch of the TwigStack algorithm. TwigStack operates in two phases. In the first phase (lines 1-7), some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 8), these solutions are merge-joined to compute the answers to the query. The input to the algorithm are a streams database and a twig pattern whose root is \( q \). The output of the algorithm are the answers to the query, which are presented as groups of paths in the data tree from root to leaf according to the query twig pattern rooted at \( q \). Each group contains all the paths that participate in the data pattern matching the twig.

The function \( \text{getNext}(q) \) identifies the next query node to be processed. This function is not intuitive. \( \text{getNext}(q) \) is a recursive function that looks in all the streams that correspond to the query nodes of the twig pattern rooted at \( q \). \( \text{getNext}(q) \) returns the query node associated with the L-Stream whose currently processed node, say \( x \), has the smallest LeftPos value and that is \textbf{guaranteed to be part of a final solution} (meaning that at least one of the final answers will contain node \( x \) in one of its paths). \( \text{getNext}(q) \) guarantees that each solution to each individual query root-to-leaf path is merge-joinable to some of the other root-to-leaf paths, so that together they form a solution to the twig query. This ensures that no intermediate solution is larger than the final answer to the query, as there are no intermediate results that are not part of the final answer. All the intermediate results are exactly of the same size as the final results. For more details about the \( \text{getNext}(q) \) function, see [26].
In line 3 all the stacks are cleaned, i.e., all the (irrelevant) nodes that do not belong to the current path from the root of the data tree to node $x$ (that was mentioned above) are popped up from the stacks. In line 4 the currently processed node (i.e., node $x$) is copied from the L-Stream to the stack. While copying $x$ to the stack, a pointer is added from the copy of $x$ on the stack to the node that is currently at the top of the stack with the same label as of the parent node of $qN$ (defined in line 2). This pointer helps to encode the answers of the query within the stacks. In lines 6-8, the currently found path results are expanded from the stacks.

**Example:** I use the source document tree that is presented in Figure 2.2, and the query pattern that is rooted at an "Actor" node and has 2 sub-paths: "First/Rupert" and "Last/Grint".

The call starts with $q$ being the root of the query tree, namely "Actor". According to line 2, the first query node that is chosen is "Actor" with indices "16:25" (namely $\text{getNext}(\text{Actor}) = "16:25"$). This node is returned as it has a L-Stream with a data node that is a part of a solution and that has the smallest LeftPos value among all the unprocessed nodes in all the streams that belong to the query nodes (the indices of the currently processed node in the "Actor" L-Stream are "16:25"). The currently processed node is not the node with indices "6:15" even though it has a LeftPos index that is smaller than that of the currently processed node (with indices "16:25"). The reason is that node "6:15" is not a part of any answer, so it is ignored. Query nodes "First" and "Last" were not chosen even though their streams have
unprocessed nodes with smaller LeftPos value than of the currently processed node in "Actor" L-Stream (nodes with indices "7:10" and "11:14"). The reason is that these nodes are not part of any answer. There is nothing to do in line 3, as the stacks are empty. According to line 4 the currently processed node ("16:25") in L-Stream "Actor" is copied to the stack with the name "Actor". Since "Actor" has no parent, there is no need to install a pointer from the copy of "16:25". "Actor" is not a leaf, so (according to line 5) I loop to line 2.

The next query node that is chosen in line 2 is "First". It is chosen because the currently processed node in its L-Stream that is part of some answer has indices "17:20", where 17 is the smallest LeftPos among all the nodes of the streams corresponding to the query nodes that are part of some answer. There is nothing to do in line 3 as there is only one node in the stacks, and the currently processed node (node "First" with indices "17:20") is in the continuation of the path from the data tree root to node "16:25". According to line 4 node "17:20" is copied to the stack with the name "First", and set a pointer between this node and the node at the top of the "Actor" stack (the node with indices "16:25").

The next query node that is chosen in line 2 is "Rupert". It is chosen because the currently processed node in its L-Stream that is part of some answer has indices "18:19", where 18 is the smallest LeftPos among all the nodes of the streams corresponding to the query nodes that are part of some answer. There is nothing to do in line 3 as the currently processed node (node "Ru-
“Rupert” with indices “18:19”) is in the continuation of the path from the root to node “17:20” whose copy is currently in the stack. According to line 4 node “18:19” is copied to the stack with the name “Rupert”, and set a pointer between this node and the node at the top of the “First” stack (the node with indices “17:20”). According to lines 5-7 the path that is currently encoded in the stack is expanded, as “Rupert” is a leaf node in the query. The path is: “Actor” with “16:25”, “First” with “17:20”, “Rupert” with “18:19”.

The next query node that is chosen in line 2 is “Last”. It is chosen because the currently processed node in its L-Stream that is part of some answer has indices “21:24”, where 21 is the smallest LeftPos among all the nodes of the streams corresponding to the query nodes that are part of some answer. According to line 3, the node with indices “17:20” is deleted from the “First” stack and the node with indices “18:19” from “Rupert” stack, as they do not belong to the path connecting the root of the data tree and the currently processed node which has indices “21:24”. It is figured, by examining the RightPos value of the currently processed node (24) that is larger than 19 and 20, the RightPos values of nodes “17:20” and “18:19”. According to line 4 node “21:24” is copied to the stack with the name “Last”, and set a pointer between this node and the node at the top of the “Actor” stack (the node with indices “16:25”).

The next query node that is chosen according to line 2 is “Grint”. It is chosen because the currently processed node in its L-Stream that is part of some answer has indices “22:23”, where 22 is the smallest LeftPos among all
the nodes of the streams corresponding to the query nodes that are part of some answer. There is nothing to do in line 3 as the currently processed node (node "Grint" with indices “22:23”) is in the continuation of the path from the root to node "21:24" whose copy is currently on the stack. According to line 4 node "22:23" is copied to the stack with the name "Grint", and set a pointer between this node and the node at the top of the "Last" stack (the node with indices "21:24"). According to lines 5-7 I expand the path that is currently encoded in the stack, as "Grint" is a leaf node in the query. The path is: "Actor" with "16:25", "Last" with "21:24", "Grint" with "22:23".

At this point I have processed all the nodes in the relevant streams, so the algorithm exits the while loop of line 1.

According to line 7 a merge between all the path solutions that were found is performed. So, both of the paths that were found were merged to form one answer. For the example document, this is the only answer.

2.7 Graphics Processors (GPUs)

In this section, I briefly introduce GPUs and CUDA, the underlying programming framework upon which the GGQ algorithm is coded.

GPUs were originally designed for dealing with graphics rendering. In recent years GPUs are also used as multi-threaded multi-core co-processors for CPUs and are widely available as commodity components in modern machines [77, 15]. GPUs have a SIMT (Single Instruction, Multiple Thread) ar-
architecture. In the SIMT architecture, there are multiple processing elements that perform the same operation on multiple data, simultaneously. Any algorithm for GPUs has to fit the SIMT scheme; hence an original CPU (or multi-core) algorithm should not be run as is on a GPU. If run as is, it will likely be extremely inefficient. Programmers write their algorithms so that the part of the algorithm that does not have to be massively parallelized runs on the CPU and the other part, which can be massively parallelized, runs on the GPU. GPU programming languages include graphics APIs such as OpenGL [11] and DirectX [22], and GPGPU frameworks such as NVIDIA’s CUDA [10] which I use.

**GPU Hardware Model.** The GPU architecture is shown in Figure 2.10. This architecture is common to both NVIDIA [74] and AMD [54]. The NVIDIA GTX processor, the type I used (NVIDIA has many types of GPUs), is a collection of multiprocessors (in GTX480 there are 15 multiprocessors), each with a group of processors (32 in GTX480). Each multiprocessor has its own shared memory, called the local memory, which is common to all the processors within it. It also has a set of registers, texture, and constant memory caches. At any given cycle, each processor in the multiprocessor executes the same instruction on different data. A warp is a collection of threads that run simultaneously at a particular point in time on a multiprocessor. The warp size is fixed for a specific GPU hardware type. Communication between multiprocessors (i.e., processors from different multiprocessors) is through the device memory (also called global memory), which is available to all the
processors of the multiprocessors. The size of the global memory is limited. The GTX480, for example, has 1.5GB of global memory. The global memory has both a high bandwidth and high access latency. GPU threads have both low context-switch and low creation time as compared to CPU threads. The global memory is available to all the threads, so any thread can concurrently access any memory location. The GPU has a hardware feature that is called coalesced memory access. Coalesced memory access means that when many threads in a warp access consecutive global memory addresses, these memory accesses are grouped into one access. By that, the number of (slow) global memory accesses is significantly reduced, and memory bandwidth utilization is improved.

**CUDA Programming Model.** Programmers use two types of code, the **kernel code** and the **host code**. The kernel code is executed on the GPU. The host code runs on the CPU. The host part is in charge of transferring data between the GPU and CPU main memory, and starting kernel-code instances (kernels) on the GPU. A computation task on the GPU is divided into three separate steps. First, the host code allocates GPU memory for input and output data, and copies input data from the main memory to the GPU memory. Second, the host code starts threads each executing the kernel code, kernels, on the GPU. The kernels perform the required task on the GPU. Third, when the kernels finish their work, the host code copies results from the GPU global memory to the CPU main memory. For the programmer the CUDA model is a collection of threads running in parallel.
A collection of threads (called a block) runs on a multiprocessor at a given time. One can assign multiple blocks to a single multiprocessor and then the blocks execution on the multiprocessor is time-shared.

**Execution.** All threads of all blocks executing on a single multiprocessor share its resources. Each thread and block has a unique ID. In addition, each thread has a program counter, registers, per-thread private memory, and inputs that can be used by the thread during its execution. Each thread in a set of parallel threads executes an instance of the kernel code, in parallel. Blocks are further organized into grids of thread blocks by the programmer. Each grid is a 2 or 3-dimensional arrangement of blocks. When a block is executed, it is further divided by a scheduler into warps. Using the thread and block IDs each thread can perform the kernel code on different set of data. In some cases, during some operations, for example an if else statement, some of the threads in a multiprocessor are idle (during the if block or the else block), as according to the if else statement, they do not have to process the body of the if block or the else block of the statement.

In conclusion, using CUDA, the GPU can be viewed as a massive parallel SIMD processor.
Figure 2.2: L-Stream storage scheme example
locpath ::= ‘/’ locpath | locpath ‘/’ locpath | locpath ‘—’ locpath | locstep.
locstep ::= axis ‘::’ ntst ‘[‘ bexpr ‘]’... ‘]’ bexpr ‘]’.
bexpr ::= bexpr ‘and’ bexpr | bexpr ‘or’ bexpr
| ‘not(‘ bexpr ‘)’ | locpath.
axis ::= ‘self’ | ‘child’ | ‘parent’ | ‘descendant’
| ‘descendant-or-self’ | ‘ancestor’ | ‘ancestor-or-self’
| ‘following’ | ‘following-sibling’ | ‘preceding’
| ‘preceding-sibling’.

Figure 2.3: Core XPath Grammar

bexpr ::= bexpr ‘and’ bexpr | bexpr ‘or’ bexpr
| ‘not(‘ bexpr ‘)’ | locpath | nexpr relop nexpr.
expr ::= locpath | bexpr | nexpr.
nexpr ::= ‘position()’ | ‘last()’ | number
| nexpr arithop nexpr.
arithop ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘mod’.
relop ::= ‘=’ | ‘!’ = | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’.

Figure 2.4: Wadler Fragment, changed and new rules

locestp ::= axis ‘::’ ntst ‘[‘ bexpr ‘]’.
bexpr ::= bexpr ‘and’ bexpr | bexpr ‘or’ bexpr | locpath
| nexpr relop nexpr.

Figure 2.5: pWF changed and new rules
Figure 2.6: Combined complexity of XPath. An arrow from $L_1$ to $L_2$ means that language $L_1$ is a fragment of language $L_2$. Taken from [43]
**Input**: 1. Streams database. 2. Query path pattern with root $q$.

**Output**: The answers to the query.

**Method**:
1. **while** ($\neg$eof)
2. $qN = \text{getMin}(q)$
3. **for** $q_i$ in subtree nodes of $q$
4. clean stack of node $q_i$ by popping irrelevant nodes
5. push a copy $T_{qN}$’s current element to $S_{qN}$
6. **if** $qN$ is a leaf node
7. **expand** solutions
8. **pop** node from $S_{qN}$
9. **end while**
**Input:** 1. L-Stream database. 2. Query twig pattern with root q.

**Output:** The answers to the query.

**Method:**
1. while (leaf)
2. \( qN = getNext(q) \)
3. clean stacks
4. if \( T_{qN} \)’s first element is part of a solution
5. push it to \( S_{qN} \)
6. if \( qN \) is a leaf node
7. expand solutions
8. \text{mergeAllPathSolutions()} 
9. end while

Figure 2.9: The TwigStack Algorithm Sketch

Figure 2.10: GPU architecture model
Chapter 3

Partitioning L-Streams

According to a Given XML Tree Partition

I start with an overview of the system. The system operates in two phases: loading phase and querying phase. The loading phase is done just once. During the loading phase, the input XML document is parsed and stored according to the XML L-Stream storage scheme (see Chapter 2.2.1). Parsing is done using SAX [16], an event based sequential access parser API for XML documents that provides a mechanism for reading data from a XML document. During the querying phase, The system can run as many queries as needed over the stored XML database.
3.1 XML Tree Partition

The key to parallelizing the PathStack and TwigStack algorithms is in the ability to partition L-Streams according to a given XML document tree partition. This allows threads to work in parallel on different parts of the L-Streams. I define the FindAllSubStreams Algorithm. This algorithm allows to isolate the sub-L-Stream ss, in L-Stream s, whose nodes belong to a given subtree st (see the L-Stream and sub-L-Stream definitions in Section 2.2.1).

I define a partition of tree T as a sequence of disjoint subtrees of T, each represented by its root node and containing all this root node’s descendant nodes, such that each leaf of T belongs to exactly one such subtree. This is not a partition in the set theoretic sense as there are data tree nodes (namely, nodes on the paths between the root of the entire data tree and the roots of the partition’s subtrees) that do not belong to any partition subtree (see Figure 3.1: for example, the nodes that are "above" rt1, rt2 and rt3 do not belong to any member, i.e., subtree, of the partition).

To parallelize query processing on some source document tree, I need to create a partition, named P, of the source document tree (referred to as T). Partition P is practically virtual as it is represented by a list that contains only pointers to the root nodes of the subtrees of P. Note that P is created once, during the loading phase. P serves to process all future queries against the document. The size of a tree is measured by the number of its nodes. There are two requirements that influence the way I build P. The first is that
each subtree in the partition must be large enough. Otherwise, the time that each thread spends on getting the task from the pool (collection of subtrees) is much larger than the time it spends on processing the subtree. For good load balancing, the second requirement is that the number of subtrees should be much larger than the number of threads, so as to prevent idle threads. So, I build P such that the maximal size of a subtree \( st \) in the partition is equal to \( S \), a tuning parameter. Clearly, \( S \) needs be determined for each system and working environment. Experimentally, I found that in my working environment, for \( S = 50,000 \) nodes, both of the above requirements hold for the documents that were checked.

Algorithm 1 describes the \( \text{addToPartition} \) function that builds P during the parsing process that is performed using the SAX parser API. The \( \text{addToPartition} \) function is called at the \( \text{endElement} \) function of SAX [16]. When the \( \text{endElement} \) function is called for an element \( n \), the size of the subtree of \( n \) is already known. Line 2 checks if \( n.\text{subtreeSize} \) is smaller than the \( \text{maxSubtreeSize} \) parameter. \( \text{maxSubtreeSize} \) is the size of the maximal possible subtree size in the partition. Before inserting \( n \) to the end of the partition \( P \) list (line 4) the \( \text{rmvChildNodes} \) function is called (line 3). \( \text{rmvChildNodes} \) scans the list of pointers to tree roots that make up \( P \) and deletes all nodes that are children of \( n \).

There can be documents for which partition \( P \) , as built above, does not provide sufficient performance, because one of the requirements described above is not holding (the subtrees are too big, so there are not enough sub-
Algorithm 1 The addToPartition function

1: function addToPartition(d, n, P)
   \(d\) is the source document tree, \(n\) is a node in \(d\), \(P\) is the partition, i.e., list of subtree roots.
   \(\triangleright\) Function Goal: add \(n\) to \(P\).
2: if \(n\.subtreeSize \leq maxSubtreeSize\) then
3:   rmvChildNodes\((P, n)\)
4:   Add \(n\) to \(P\)
5: end if
6: end function

trees for balanced parallelization). For such documents, in order to obtain performance results, which are similar to most of the documents, I create an additional partition \(P_1\) using a smaller value for the parameter \(S\). With this change PPS, using \(P_1\), performs similarly.

3.2 findAllSubStreams Algorithm

Ordinarily, a thread fetches a sequence (i.e., one forming a contiguous sub-list) of subtrees of partition \(P\). Then, the sequent subtrees are united into one tree, called \(ft\) (abbreviation for Final Tree).

Tree \(ft\) is defined as follows. Assume that I have a partition \(P\) of some document tree. Assume that \(g\) is a sub-sequence of sequent subtrees in \(P\). Tree \(ft\) is composed of the subtrees in \(g\) as well as nodes and edges on the paths from the (entire) document tree root to the roots of the subtrees in \(g\), for example, see Figure 3.1. On the left side of the figure we see 3 subtrees \((t1,t2,t3)\) that form \(g\). These subtrees have solid edges (colored red). On the right side we see the \(ft\) tree whose root is node \(ft\)-root and which has solid
In order to recognize the sub-L-Streams restricted to the $ft$ tree, I use function $\text{findAllSubStreams}$; see Algorithm 2. Assume that $qStreams$ is the set of L-Streams of the query nodes of query twig pattern $q$. The goal of this function is to find the start indices of the sub-L-Streams (for each L-Stream in the set $qStreams$) that belong to the tree $ft$ (that is built from $g$’s subtrees).

For each tree $ft$, the relevant nodes for query $q$ in $ft$ can be represented by a set of sub-streams $ssSet$. $ssSet$ consists of one sub-L-Stream $ss$ for each L-Stream $s$ in $qStreams$.

Function $\text{findAllSubStreams}$ finds all sub-L-Streams of $ssSet$ using the $\text{findSubStreamsIndexes}$ function which finds the start index of some sub-L-Stream $ss$ within L-Stream $s$. To find it, the function performs a special
binary search. It performs a search for a node \( n \) with the smallest index in \( s \) such that \( n \) belongs to \( ft \), and it has the smallest possible \( \text{LeftPos} \) value. More precisely, let \( rt_1 \) be the root node of the leftmost subtree in \( g \), let \( rt_x \) be the root node of the rightmost subtree in \( g \). I search for a node \( n \) in \( s \) with the smallest \( \text{LeftPos} \) such that:
\[
n.\text{RightPos} > rt_1.\text{LeftPos} \quad \text{and} \quad n.\text{LeftPos} < rt_x.\text{RightPos}.
\]
Lines 11-22 implement the binary search \textit{while} loop. Lines 12-13 define the current node of the current loop iteration. The \textit{if} in line 14 checks and updates the bottom border of the search. The \textit{if} in line 17 checks if the current node is a possible answer and in case of a positive answer, it updates the \textit{returnNI} variable. Line 20 updates the upper border of the search. Lines 23-26 check for the last unchecked node in the L-Stream. Note, that if \textit{returnNI} remains ‘-1’ at the end of the function run, it means that the current L-Stream does not
contain any nodes that belong to $ft$.

In fact, it is enough to look for the start index of some substream $ss$. I do not need to find it’s end index. This is because I have this information readily available. All nodes whose $LeftPos$ value is bigger (or equal) to the $LeftPos$ value of the root of the first subtree in $g$, and smaller than the $RightPos$ value of the root of the last subtree in $g$ belong to tree $ft$. For example, for the $ft$ that is presented in Figure 3.1 and for L-Stream $s$, the nodes that belong to $ss$ are all nodes whose $LeftPos$ value is bigger (or equal) than $n.LeftPos$ but smaller than $rt3.RightPos$. 

Algorithm 2 the findAllSubStreams algorithm

1: function findAllSubStreams(subLLIdx, subRLIdx, q)
   ▷ q is a path or twig pattern, subLLIdx is the LeftPos of rt, subRLIdx is the RightPos of rt.
   ▷ Function Goal: find the first node n in substream ss (defined by subLLIdx, subRLIdx) for all the streams of nodes appearing in the q tree.
2:     for i = 1 → q.numOfNodes() do
3:         nI ← q.getNode(i)
4:         nI.ssLimit ← findSubStreamsIndexes(subLLIdx, subRLIdx, nI.getStream())
5:     end for
6: end function

7: function findSubStreamsIndexes(subLLIdx, subRLIdx, s)
   ▷ s is a L-Stream, subLLIdx is the LeftPos of rt, subRLIdx is the RightPos of rt.
   ▷ Function Goal: find the first node n in substream ss (defined by subLLIdx, subRLIdx) of L-Stream s.
8:     firstNI ← 0 ▷ smallest node index in s
9:     lastNI ← s.size − 1 ▷ largest node index in s
10:    returnNI ← −1 ▷ return value
11:    while lastNI > firstNI do
12:        currI ← firstNI + (lastNI − firstNI)/2
13:        currN ← s[currI]
14:        if currN.RightPos < subLLIdx then
15:            firstNI ← currI + 1
16:        else
17:            if currN.LeftPos ≤ subRLIdx then
18:                returnNI ← currI
19:            end if
20:        end if
21:    end while
22:    cN ← s[firstNI]
23:    if cN.RightPos > subLLIdx ∧ cN.LeftPos < subRLIdx then
24:        returnNI ← firstNI
25:    end if
26:    return returnNI
27: end function
Chapter 4

The Parallel PathStack and TwigStack Algorithms

This part of the research was published as a TKDE journal article in 2014 [89].

4.1 The Parallel PathStack (PPS) Algorithm

The main idea behind the Parallel Path Stack algorithm is to virtually partition the XML document tree to, more or less, equal subtrees, and then to run the original PathStack algorithm on each subtree with some additional operations. The previously described virtual partition (named P) is created once while parsing and loading the document. It is important to note that the data partitioning strategy is a general approach to parallelization, while PPS
is a specific algorithm that "implements" this philosophy over XML data. Note that PPS operates only on queries with the "//" axis. The algorithm can be easily extended to operate with the "/" axis as well (as explained in [26]).

In the next sections, I denote by $ft$ the tree constructed in the manner described in Chapter 3.

I start with a short overview of the algorithm. The subtrees of partition P act as a pool of tasks, in which each subtree constitutes one task. After activating the desired number of threads (according to the $numOfThreads$ parameter), the pool tasks are processed by all the active threads. Each available thread takes $numOfTasks$ tasks from the pool ($numOfTasks$ is a parameter), until all the tasks in the pool are processed. Note that each group of taken subtrees (tasks) are sequenced according to their order in the tree (to allow the usage of the findAllSubStreams algorithm, see Chapter 3). After picking $numOfTasks$ subtrees from the pool, for each L-Stream $s$ in $qStream$ (see Chapter 3) the thread (by using the findAllSubStreams algorithm) finds the limits of sub-L-Stream $ss$ in $s$; the found sub-streams correspond to the $ft$ tree that is composed of the picked subtrees. After finding these limits, the thread processes the query twig pattern, corresponding to its $ft$ tree, using these sub-streams.

The main difference between the original and the Parallel PathStack algorithms is in the way streams are treated. In the original PathStack, all the streams are read from beginning to end. In the parallel version, each thread...
reads from the streams only nodes that belong to its built \( ft \) tree. For this, I have re-implemented all the relevant functions in the PathStack algorithm \((\text{eof}, \text{getMinSource}, \text{etc} ; \text{See} \ [26] \text{for more details about these functions}).

**Algorithm 3** The \textit{Parallel Engine} function

1: \textbf{function} \textit{parallelEngine}(d, q, P, numOfThreads, numOfTasks, qProcessFunction)  
\quad \triangleright d \text{ is the source document tree, } q \text{ is the twig or the path pattern, } P \text{ is the document partition, } numOfThreads \text{ is the number of threads to be run in parallel, } numOfTasks \text{ is the number of tasks taken each time from the pool by a thread, } qProcessFunction \text{ is the processing algorithm that is used by the threads.}
\quad \triangleright \textbf{Function Output}: answers to \( q \) in \( d \).
2: \quad \text{pool} \leftarrow \text{makePoolOfTasks}(P)  
3: \quad \text{createThreads}(\text{numOfThreads})  
4: \quad \textbf{while} \text{ an unprocessed task } t \text{ exists in pool} \textbf{ do}  
5: \quad \quad \text{thread} \leftarrow \text{findAvailableThread}()  
6: \quad \quad \text{tasks} \leftarrow \text{pickTasks(thread, pool, numOfTasks)}  
7: \quad \quad \text{thread.runThread(queryProcessFunction, tasks, q)}  
8: \quad \textbf{end while}  
9: \textbf{end function}

10: \textbf{function} \textit{runThread}(queryProcessFunction, tasks, q)  
\quad \triangleright \text{queryProcessFunction} \text{ is the function that executes the query, } tasks \text{ is group of tasks, } q \text{ is the twig or the path pattern.}
\quad \triangleright \textbf{Function Output}: The answer to \( q \) found in subtrees corresponding to \( tasks \)
11: \quad \text{subStreamsL} \leftarrow \text{findSubStreamsIndexes(tasks)}  
12: \quad \textbf{return} \ qProcessFunction(q, tasks, subStreamsL)  
13: \textbf{end function}

Algorithm 3 presents the \textit{parallelEngine} function for running different base algorithms, in parallel, over XML documents. The input to the function are the source document tree, the twig pattern referred to as \( q \), the \text{numOfThreads} which denotes the number of threads to be run in parallel,
the *numOfTasks* which denotes the number of subtrees to be picked up each time by a thread, and the *qProcessFunction* which is the function that processes *q* over some part of the source document tree. All the subtrees of partition P form a pool of tasks. A pool of tasks is created in line 2. All active threads are initiated according to the *numOfThreads* parameter in line 3. An available thread takes *numOfTasks* tasks from the pool (line 6), and processes them (line 7). This process continues until all the tasks in the pool are processed (lines 4-7). The limits of sub-streams of query nodes of *q* that belong to the relevant *ft* tree for this thread are found in line 11 and assigned to a structure called *subStreamLimits*. The *ft* tree is constructed only conceptually (to make it easier to understand the rationale of the performed operations) according to the whole sequence of tasks that were assigned to the current thread. Now, in line 12, after finding the sub-L-Stream limits for a thread, I can execute the *queryProcessFunction*. This function is executed on the *ft* tree that was found in line 11 (and *not* separately on each subtree in the fetched group).

Algorithm 4 presents the full Parallel PathStack algorithm. The parameters are as indicated, with the same meaning as before. The *runPathStack* function executes the query over the sub-streams that belong to the current thread. The *runPathStack* function is passed to the *parallelEngine* function (it is the fifth parameter of the *parallelEngine* function) and activated in the *parallelEngine* function (see Figure 3, line 7). The *P* parameter that is sent to the *parallelEngine* function is the partition that was prepared in advance.
Algorithm 4 The Parallel Path Stack Algorithm

1: function \textit{ParallelPathStack}(d, q, numOfThreads, numOfTasks)
\hspace*{1em}\triangleright d is the source document tree, q is the path pattern and 
\hspace*{1em} numOfThreads is the number of threads that has to be run in parallel
\hspace*{1em}\triangleright \textbf{Function Output}: answers to q in d
2: \hspace*{1em} return parallelEngine( d, q, P, numOfThreads, numOfTasks, 
\hspace*{1em} runPathStack)
3: end function

4: function runPathStack(q, tasks, subStreamL)
\hspace*{1em}\triangleright q is the path pattern, tasks is group of sequent subtrees, subStreamL contains limits in streams of current subtrees.
\hspace*{1em}\triangleright \textbf{Function Output}: The answer to q found in tasks
5: \hspace*{1em} results $\leftarrow \{\}$
6: \hspace*{1em} while $\neg$eof(tasks, subStreamL) do
7: \hspace*{1em} \hspace*{1em} qN $\leftarrow$ getMin(tasks, subStreamsL, q)
8: \hspace*{1em} \hspace*{1em} cleanStacks()
9: \hspace*{1em} \hspace*{1em} $S_{qN}.push(T_{qN}.firstElement)$
10: \hspace*{1em} \hspace*{1em} if qN.isLeafNode() then
11: \hspace*{1em} \hspace*{1em} \hspace*{1em} expandSolutions(results)
12: \hspace*{1em} \hspace*{1em} end if
13: \hspace*{1em} end while
14: \hspace*{1em} return results
15: end function
while parsing the document. The limits of sub-streams of query nodes of \( q \) that belong to the \( ft \) tree are found in the \textit{runThread} function inside the \textit{parallelEngine} function and assigned to a structure called \textit{subStreamLimits} which is then passed to the \textit{runPathStack} function. In line 5 \textit{results} are initialized, which is the set of found answers for \( q \). Function \textit{eof}, in line 6, checks if all the sub-streams that belong to \( ft \) are completely used. Function \textit{getMin} in line 7 finds the node with the smallest \textit{LeftPos} value in all the sub-streams that belong to \( ft \). In function \textit{expandSolutions} (line 11), the found answers are added to \textit{results} set. Lines 6-13 are essentially the original Path-Stack algorithm, for more details see \cite{26}. Practically, I managed to handle the problem of non-uniform distributions (or size-unbalanced subtrees). As explained above, I create a pool of subtrees. These subtrees are usually size-unbalanced and/or have a non-uniform distribution of labels. However, the method of working with a pool of tasks exhibits good load balancing behavior. As I have numerous subtrees (much much more than processing cores), during the time that a large (heavy) task is processed by some thread, other cores process many small subtrees, thereby providing good load balancing.

\textbf{Example:} Assume that the data source tree is the one that is presented in Figure 3.2, the path pattern is "/Movie//last//Radcliffe", and the \textit{numOfThreads} is 2. Partition P, created while parsing the document, has 3 subtrees. The partition subtrees’ roots are nodes: (2;5,2), (6;15,2) and (16;39,2). Assume that \textit{numOfTasks} equals 1. The first thread processes the subtree rooted at
The second thread processes the subtree that is rooted at (6:15,2).

In line 11 of Algorithm 3 I run the `findAllSubStreams` function, and find the limits of all sub-streams of `q` that belong to the relevant `ft` tree. This `ft` tree is composed of the path from the entire document tree root (`movie`) to node (6:15,2) (the root of task 2), and from subtree 2. In lines 5-14 (of Algorithm 4), using the `subStreamsL` structure, I discover one full path answer (1:40,1), (11:14,3), (12:13,4) and send it to the output. As the first thread has less work (its subtree does not yield any answer), I can assume that it finishes processing its task first. So, it checks the pool and finds another unprocessed task, and processes it.

**PPS complexity:** Please see the complexity analysis of the PTS algorithm (Section ). The complexity analysis of the PPS algorithm is a special case of the PTS algorithm. Thus, the PTS complexity analysis, with minor adjustments (not taking into account the merge component of the TwigStack algorithm), also applies to the PPS algorithm.
4.2 The Parallel Twig Stack (PTS) Algorithm

For document partitioning purposes, in most cases PTS uses the same partition P as PPS. PTS operates only on queries with the "//" axis. Again, the algorithm can be easily extended to work with "/" (as explained in [26]). The main difference between the PPS and PTS algorithms is that the PTS algorithm has to deal with overlapping paths within one query. Thus, partition P can possibly cause a problem while processing a twig pattern, as it may happen that in P part of some query answer is within one subtree while the other parts are placed within other subtrees. This means that P potentially breaks some answer apart, and that special operations must be performed in order to assemble this answer.

To solve this problem, one solution is as follows. I prepare in advance an additional partition, called twigP. Partition twigP is prepared during document parsing. The main parameters of any partition are the number of subtrees in the partition and the maximal size of subtrees in the partition. Partition twigP is chosen in a way that its maximal subtree size is much larger. I.e., the roots of twigP subtrees are much closer to the document root. While preparing partition twigP, I attempt to achieve two objectives. First, that there would be at least 10,000 subtrees. This system constant is chosen in light of the number of threads, conceivably this constant would be different when more threads are available. Second, I would like the partition roots to be as close as possible to the root of the document tree, so that the resulting
subtrees would have significant sizes, otherwise, there will not be enough actual work for threads to perform. The main idea of this additional partition is to avoid a situation in which a partition can potentially break some answer apart. Thus, before using the basic partition $P$, I examine it and check if it possibly can break some query answer apart. In case of a positive answer, PTS uses partition twig$P$ instead of partition $P$.

Rarely, even twig$P$ potentially breaks some answers apart. In such cases, I break twig pattern $q$ into a collection of twig patterns as will be shortly described.

To check if a partition can potentially break some query answer apart, I perform the following: (1) Find the upper junction of the twig pattern. I.e., the node that has a minimal level and that has more than one child. I will refer to this node as the junction. (2) For each subtree in the partition, check for each node $n$ on the path between the document root node and the parent node of the subtree root, whether the label of $n$ equals to the label of the (twig pattern) junction. A positive answer means that this partition can potentially break an answer apart.

**Discussion:** If the mapping of the junction node to matching nodes in the document tree results in nodes that are close to the document tree root (for example, at level 3), then, it seems possible that partitions have few large subtrees. In fact, this is usually not the case. For starters, parallelization is beneficial for large documents. Also, ordinarily XML documents are flat, i.e., the maximal depth is small. These facts imply that somewhere close to the
root, or even at the root (1'st or 2'nd level), there are nodes with a very large number of children. So, even if one is forced to choose partition twigP whose subtrees’ roots belong to low levels, this does not limit the number of possible useful threads. Because, in most cases, there are many more subtrees than possible threads.

Still, there can be a problem if the junction node is the root of the document tree. First, usually queries do not have a junction at the root, as queries about the root of the document are rare. For example, if I look at the DBLP database, using the root of the DBLP database within the pattern of the twig is fairly meaningless. Nevertheless, there is a simple solution. I can separate the twig pattern at its root into sub-patterns, run PTS on each such sub-pattern and then join the answers. This is correct as the roots of all these sub-patterns correspond to the same root in the source document tree.

Algorithm 5 presents the full Parallel TwigStack algorithm. The main differences in comparison to the Parallel PathStack algorithm are as follows. \( q \) denotes a twig pattern (and not a path pattern). The function \( \text{ChoosePartition} \) at line 2, checks if \( P \) possibly brakes some answer apart for query \( q \), and in case of positive answer, return \( \text{twigP} \) instead of the regular \( P \) partition. The \( \text{runTwigStack} \) function executes the twig query over the sub-streams that belong to the current thread. In line 5 \( \text{results} \) is initialized, which is the set of found paths answers for \( q \). Function \( \text{eof} \), in line 7, checks if all the sub-streams that belong to \( ft \) have been fully used. Function \( \text{getNext} \) in line
Algorithm 5 The Parallel Twig Stack Algorithm

1: function ParallelTwigStack(d, q, numOfThreads, numOfTasks)
   ▷ d is the source document tree, q is the twig pattern, numOfThreads is the number of threads to run in parallel, numOfTasks is the number of tasks taken each time from the pool by a thread
   ▷ Function Output: answers to q in d
2:     prt ← ChoosePartition(d, q)
3:     return parallelEngine(d, q, prt, numOfThreads, numOfTasks, runTwigStack)
4: end function

5: function runTwigStack(q, tasks, subStreamL)
   ▷ q is the twig pattern, tasks is group of sequent subtrees, subStreamL limits in streams of current subtrees.
   ▷ Function Output: The answer to q found in tasks
6:     results ← {}
7:     while ¬eof(tasks, subStreamL) do
8:         qN ← getNext(tasks, subStreamL, q)
9:         cleanStacks()
10:        if isPartOfSolution(T_qN.firstElement) then
11:            S_qN.push(T_qN.firstElement)
12:        end if
13:        if qN.isLeafNode() then
14:            expandSolutions(results)
15:        end if
16:     end while
17:     return mergeAllPathSolutions(results)
18: end function
8 identifies the next node to be processed (it looks in the sub-streams parts that belong to the relevant ft tree and that correspond to the twig pattern rooted at q, and chooses the node with the smallest LeftPos value, that is guaranteed to be part of a solution). In function expandSolutions (line 14) the found path answers are added to results set. In line 17, all the found path results are merged in order to combine the twig answers. Lines 5-17 are essentially the original TwigStack algorithm [26].

In some cases, in which there are few query answers, or in which answers are distributed non-uniformly, the single thread parallel algorithm is more efficient than the corresponding sequential algorithm (that is run separately on each sub-tree in the parallel version). This has to do with skipping substantial portions of streams. I.e., when operating on a subtree, the parallel algorithm will "skip" scanning all sub-streams (of this subtree) in case at least one of the sub-streams of this subtree is empty.

### 4.3 PTS Complexity

Note that this analysis also applies to the PPS algorithm (which is a special case of the PTS algorithm), while excluding the merge component of the TwigStack algorithm.

An input/output-operation (I/O) is reading (or writing) a block from (or to) disk. I/O complexity measures the number of I/Os used to solve a problem as a function of the input. Time complexity measures the amount of CPU
time units required to solve the problem.

**Claim 1:** The number of I/Os and runtime of all the threads run by the PTS algorithm have the same complexity as those of the original TwigStack algorithm. I.e, PTS’s worst-case I/O and total time complexities are linear in the sum of the sizes of the \( r \) input lists (streams) and the output list.

**Proof:** In [26] it was proved that, given a query twig pattern \( q \) with \( r \) nodes, and only ancestor-descendant edges, and an XML database \( D \), the TwigStack algorithm has worst-case I/O and runtime complexities linear in the sum of the sizes of the \( r \) input lists and the output list. Further, the worst-case space complexity of TwigStack is the minimum of (i) the sum of the sizes of the \( r \) input lists, and (ii) \( r \) times the maximum length of a root-to-leaf path in \( D \). The complexity of the I/O and runtime of each thread is the same as that of the TwigStack algorithm, had it processed the same subtree that is processed by the thread. While running the PTS algorithm, I only need to choose one of the partitions. The process of choosing a partition takes constant time. After choosing a partition, each thread executes (almost) the original TwigStack algorithm. The first difference is in the process of finding sub-streams limits. The complexity of this search for a L-Stream is \( \log(n) \) where \( n \) is the size, i.e. the number of nodes, of the L-Stream in which I search the sub-L-Stream limit. This search is done \( k \times p \) times, where \( p \) is the number of streams that participate in query \( q \) and \( k \) is the number of times that a thread approaches the pool of tasks (subtrees) to pick up a group of tasks. As \( k \times p \times \log(n) \ll n \), the complexity is unaffected. Additional
differences exist in functions \textit{getNext} and \textit{eof}. These functions check the boundaries of the current subtree during their run. This checking is done in constant time, so the complexity of the \textit{getNext} and the \textit{eof} functions remains the same. There are no other differences between PTS thread and the original TwigStack algorithm.

The conclusion from the above arguments is that the total I/O and runtime taken over all the threads run by the PTS algorithm are of the same complexity as those of the original TwigStack algorithm executed over the same input.

I define the data nodes that are the images (in a match) of the rightmost leaf node of the twig pattern as \textit{answer nodes}. Denote the runtime of the TwigStack algorithm as $TS$. Let $N$ be the number of threads.

\textbf{Observation:} By Claim 1 and the fact that answer nodes are presented as output, the runtime over a tree (including I/O operations), is proportional to the number of query streams nodes in the tree.

\textbf{Claim 2:} For the case where only answer nodes are presented as output, the parallel runtime complexity of PTS is $O\left(\frac{TS}{N}\right)$.

\textbf{Proof:} According to the definition of P, each subtree in P has maximum size $ms$. I define the \textit{amount of work} in subtree $t$ as the number of nodes in $t$ that belong to some query L-Stream. $k$ is the total size (i.e., the number of nodes) of all the query streams. $nt$ is the number of subtrees (tasks) in the partition P. $TS$ is thus the time of processing $k$ nodes by the original TwigStack algorithm. A thread $tr$ is \textit{idle} (resp., \textit{busy}) during the portion
of runtime of the algorithm in which thread \( tr \) is not (resp., is) executing while the computation is not finished. The time\(_{Waste} \) of an execution, is defined as: \( \sum_{i=1}^{N} idle(tr_i) \), where \( idle(tr) \) is the amount of time that thread \( tr \) was idle during the execution of the algorithm. In PTS, when a thread finishes processing a task it takes an unprocessed task from the pool. Hence, \( idle(tr) > 0 \) can happen only when a thread \( tr \) approaches the pool and finds no available task. This can happen only once the last task from the pool was picked. Therefore, the time\(_{Waste} \) is at most as in a hypothetical case in which all the threads but one, say \( tr \), finished their tasks right after \( tr \) picked the last task \( t \), and furthermore \( t \) has both the maximal size (\( ms \)) and the maximal amount of work (i.e., each node in \( t \) belongs to the query streams). Hence, by the Observation, the maximum possible time\(_{Waste} \) (\( maxTime\_{Waste} \)) is bounded by: \( c_1 \times (N - 1) \frac{ms}{k} \times TS \), where \( \frac{ms}{k} \times TS \) is the processing time of the last subtree \( t \), during which \( N - 1 \) threads were idle in the hypothetical scenario outlined above, and where \( c_1 \) is a constant.

Based on claim 1, \( \sum_{i=1}^{N} (busy(thread \ i)) = c_2 \times TS \), where \( c_2 \) is a constant. The parallel total runtime of PTS is the runtime of the thread with the largest executing time. During this time, other threads are partly busy and partly idle. Hence the total runtime of PTS is: \( \frac{c_2 \times TS + time\_{Waste}}{N} \). Therefore, the total runtime of PTS is bounded by \( \frac{c_2 \times TS + maxTime\_{Waste}}{N} \), i.e., \( \frac{1}{N} (TS + (N - 1) \frac{ms}{k} \times c_2 \times TS) \). After some algebraic reorganization I obtain: \( = c_2 \times TS \times (\frac{1}{N}(1 - \frac{ms}{k}) + \frac{ms}{k}) \). According to the algorithm, \( nt \), the number of subtrees (tasks) in the partition \( P \), is very large (the algorithm chooses a partition with many
subtrees). Hence, the size of the maximal subtree in the pool is much smaller than the total size of the query streams, i.e., \( ms \ll k \). Therefore, the fraction \( \frac{ms}{k} \) is very small, i.e., the time complexity approaches \( O(\frac{TS}{N}) \).

It is important to note that in case the output is presented as (full) answers to the query (i.e. vectors \( d_1, d_2, ..., d_n \)), the above claim does not hold in all cases. In case that the answers are not distributed uniformly in the data tree, the worst case complexity of the parallel algorithm is \( TS \). This can happen if all the answers of the query are placed in one subtree.

**Conclusion 1:** According to [26], TwigStack is I/O and runtime are optimal among all sequential algorithms that read the entire input (denote them by \( \textit{REI-algorithms} \)). According to Claim 1, the total I/O and the total runtime complexity, runtimes summed over all the threads run by the PTS algorithm, are optimal. So, the total I/O and the runtime complexity summed over all the threads run by PTS are optimal compared to all REI-algorithms.

**Conclusion 2:** For all parallel algorithms \( A \) that execute threads so that the entire input is read, and so that each thread runs the TwigStack algorithm on a part of the input (call this class \( \textit{PREFI} \)), the runtime of \( A \) is at least \( O(\frac{TS}{N}) \). So, PTS has a parallel runtime complexity that is optimal among \( \textit{PREFI} \) algorithms.
4.4 PPS and PTS Experimental Evaluation

I compared PPS to the original PathStack algorithm and PTS to the original TwigStack algorithm. I implemented the PathStack and the TwigStack algorithms in C++ from scratch according to [26]. I generated XML documents with different sizes using the XMark tool. XMark [85] is a commonly used generator for large XML documents. I also experimented with the DBLP database and representative hierarchical documents from the Treebank project ([70]) that were converted to XML format. I used different path and twig patterns. All experiments were run on a Sun Fire T2000 Server with Chip Multithreading Technology (CMT [76]) and with the UltraSPARC T1 processor. This processor has 8 cores and can run up to 32 simultaneous execution threads (with 2GB of RAM). I used the speedup of parallel algorithm execution time versus sequential algorithm execution time as the main metric of performance. The disk space overhead for enabling query processing was not significant.

4.4.1 Experiments Description

4.4.1.1 Setting Up

An experiment run has two input files: an XML document, and a text file with query (twig) patterns to run against the given document. An experiment begins with loading the input document into the XML Streams storage by the parser. In the second stage, I build the data that is required by the query
processor (partitions P and twigP etc.).

I used different path and twig patterns. I could not use XMark benchmark queries directly as they do not fit the requirements of my algorithms. XMark benchmark queries have predicates functions and complex axes, whereas twig pattern can have only the child or descendant axes. So, I created path and twig patterns for the experiments by taking the original benchmark queries and transforming them to the form that is required by my algorithms. Some representative patterns are as follows. Path patterns 1-3: 1-“//item//parlist//text//bold”; 2-“//site//person//name//text”; 3-“//item//mail//keyword//emph”; Twig patterns 1-2: 1-“//item[mail//emph]//listitem // parlist//text // bold”; 2-“//annotation[parlist//text // keyword//bold]//listitem[bold]//emph;  

4.4.1.2 Experiment Description

Every experiment has two runs. The first run is **Sequential Run**. After loading the document, I process the queries in the queries text file. While processing the queries, I run the original PathStack/TwigStack algorithm. The second run is **Parallel Run** - This run is performed many times with a varying number of threads. At each time, after loading the document, I process the queries in the queries text file. While processing the queries, I run the new parallel algorithms (PPS/PTS) as described in this Chapter. I compare the performance of PPS and PTS to PathStack and TwigStack (respectively), by comparing the processing runtime of the above algorithms.
I do not count the time of parsing the document and creating the required partitions (P and twigP), because this is done only once while the document is inserted into the storage system. I.e., I start the time measurement at the moment the query execution begins, and stop it right after finishing the query processing. Time is measured in milliseconds. Each experiment is characterized by settings to parameters. The main parameters are the size, number of threads and the type of the pattern (path pattern or twig pattern). The sizes I have used are 10MB, 20MB, 30MB, 112MB, 240MB and 512MB. The 512MB maximum size is dictated by the main memory size. I ran up to 32 threads.

### 4.4.2 Experiments

In all the graphs, the point *number of threads = 1* displays the speedup of executing the parallel scheme with a *single* thread against the standard sequential algorithm [26].

As previously described, I vary the number of threads in the experiments. I experimented with up to 32 threads, but do not present the results with more than 12 threads. The server I use has 8 physical cores, and with more than 12 threads, the results improve insignificantly. In all the performed experiments, there was no degradation in performance for any tested number of threads. There is still an improvement with 12 threads in comparison to 8 threads, due to the CMP technology used by the server.

The first set of experiments tested PPS. I define the data nodes that are the
images of the leaf node of the Path pattern in successful matches as answer nodes. Here, the path between the document root and the result node is not presented as part of an answer. In this set of experiments, I used a scenario in which only the node that is the answer to the query is presented as output. In Figures 4.1 and 4.2 I show the timing results of running PPS on documents with various sizes.

Observe that when there are more than 8 threads, I achieve improvement of 5-8.3 times for small documents and 7-13 times for large documents. I observe that the improvement for small documents is smaller than for large documents. This is due to the fact that in small documents the subtrees (tasks) are small, thus the time of processing a subtree is very small, and a lot of time is spent by the threads on fetching a new task from the queue of tasks. I tried to increase the number of tasks fetched by a thread from the pool each time with no significant improvement, but this undermines load balancing. This phenomenon is characteristic of other small documents.

The second set of experiments tested PTS. In this set of experiments, the results were presented as (full) answers to the query, i.e. vectors \((d_1, d_2, \ldots, d_n)\). Figure 4.3 shows the timing results of running PTS on documents with various sizes. The point \(\text{number of threads} = 1\) represents execution of the parallel scheme with only one thread. Observe that when there are more than 8 threads, I achieve speedup of 6-7.2 for small documents and 7.5-10 for large documents. Usually, when moving from one thread to two the improvement is not more than two times, and when moving to three threads it should not
Figure 4.1: Results of PPS with one node solutions on small documents with sizes 10MB, 20MB, 30MB for path patterns 1 and 3.

Figure 4.2: Results of PPS with one node solutions on larger documents with sizes 112MB, 203MB, 524MB for path patterns 1 and 3.
Figure 4.3: Results of PTS with full path solutions on documents with sizes 10MB, 20MB, 30MB, 112MB, 204MB, 524MB for twig pattern 1

Figure 4.4: Results of PTS with one node solutions on documents with sizes 10MB, 20MB, 30MB for twig patterns 1 and 2
Figure 4.5: Results of PTS with one node solutions on documents with sizes 112MB, 204MB for twig patterns 1 and 2.

Figure 4.6: Results of PTS with one node solutions on 524MB document for twig pattern 2, while changing the `numOfTasks` parameter.
Figure 4.7: Experiments with the DBLP database (PTS)

Figure 4.8: Experiments with the DBLP database (PPS)
be better than three times, in comparison to that with one thread. Note that in the presented experiments two threads in some cases give speedup which is larger than two. This happens because the algorithm that is used in the parallel scheme is not simply the ordinary sequential algorithm but a version modified to run in the new settings, and in some cases one thread that runs the parallel scheme gives a better result than the base sequential algorithm.

The third set of experiments tested PTS. I define the data nodes that are the images of the rightmost leaf node of the twig pattern in matches as *answer nodes*. In this set of experiments I used a scenario in which only the node that is the answer to the query is presented as output. Figure 4.4 presents results for twig patterns 1 and 2 over XMark, for documents with various sizes (10MB, 20MB and 30MB). Observe that when there are more than 8 threads, for twig pattern 1, I achieve speedup of 5.3-7. While for twig pattern 2, I achieve speedup of 2.5-4.4. I observe that the improvement is relatively small. This is due to the fact that the documents are small. As before, I also tried to increase the number of tasks that a thread fetches each time from the pool, but with no significant improvement. Thus the phenomenon of minor improvements for small documents also occurs when using the PTS algorithm.

The improvement of the PTS algorithm for large documents is significant as can be seen in Figure 4.5. Figure 4.5 presents the performance of PTS on larger documents (112MB, 204MB) while running both twig queries (twig pattern 1 and 2). Observe that when there are more than 8 threads, I achieve
speedup of 15.3-22. The variability in the results depends on the number of answers for a specific query. As stated before, when there are few answers, the parallel algorithm with one thread runs faster than the original sequential algorithm. The performance of PTS with twig pattern 1 is better by about 30% than the performance with twig pattern 2 as the number of results for query 1 is smaller by about 30% than the number of results for query 2 (1850 as compared to 2740). Thus, in case of query 1 there are less nodes to process and report than in query 2.

Observe that as explained before (in this Chapter) in most cases the single thread algorithm outperforms the sequential algorithm in this experiment. True, the parallel algorithm performs one time preparation (building partitions P and twigP), but amortized over many runs, this preparatory work is negligible. My parallel algorithms are based on some sequential base algorithms (e.g., TwigStack or TSGeneric+). Any such base algorithm, that scans the entire input streams, is inferior to the parallel single thread version that is based on it. This is due to the fact that in the worst case the sequential base algorithm (any one) and the parallel algorithm with 1 thread are doing exactly the same work, but in many other cases, the parallel single thread version can potentially skip large portions of the streams (this is due to its amortized preparatory work, not present in the base algorithms).

Figure 4.6 presents results of running PTS on a 524MB document for twig pattern 2. In this experiment I have changed the numOfTasks parameter. This parameter sets the number of tasks fetched by a thread each time from
the pool of tasks. The performance improves significantly when this parameter is increased. For example, while using $numOfTasks = 3$ the speedup is about 8.5, while using $numOfTasks = 20$ the speedup is about 13.3, and while using $numOfTasks = 60$ the speedup is about 14.7. The reason is that I used a partition with very small subtrees. So, by picking many tasks at once I do not undermine load balancing and I also decrease the time that the threads spend in the pool queue.

Figure 4.9: Experiments with TSGeneric+ as the base search algorithm

I conducted experiments over the DBLP database. I decreased the size of the document to 400MB. This size is dictated by the main memory size. I ran both PTS and PPS. Figure 4.7 presents the results of PTS in 2 different runs - when full answers are presented (marked as full solution) and when the path between the document root and the answer node is not presented (marked as only leaf). Figure 4.8 presents the results of PPS when the path between the document root and the answer node is not presented. Observe
that when there are more than 8 threads, I achieve a speedup of 30 for the path query and a speedup of 44 for the twig query! The reason is that there are very few results. As previously observed, when there are few answers, the parallel algorithm with a single thread runs faster than the original sequential algorithm. As observed in Figures 4.7 and 4.8, the parallel algorithm with 1 thread (point number of threads = 1) achieves speedup of about 3.3 as compared to the original sequential algorithm. So, the usage of multiple threads results in a speedup of 13.3 (44/3.3), while the fact that I use the parallel algorithm provides another speedup factor of about 3.3.

To check the PTS performance for highly recursive documents, I conducted experiments over two representative documents from the Treebank project [70] that are published in XML format by [13]. The document sizes are 82MB and 56MB, their maximal depth is 36, and their average depth is 7.8 (XMark maximal depth is 12 and average depth is 5). The twig patterns
used for the experiments are as follows.
(1) "//EMPTY//PP[NP//DT]/PP_/BACKQUOTES_";
(2) "//EMPTY//S[NP//JJS]/S//ADVP//X"; The results are presented in Figure 4.10. I obtain speedup of about 12 for twig pattern number 1 (it has about 1000 results per document) and 18 for twig pattern number 2 (it has very few results) in both documents. This improvement is larger than the improvement I obtain while using the XMark benchmark documents. So, in my experiments, the performance of PTS for highly recursive documents with similar depth parameters to the Treebank documents does not degrade as compared to flat documents such as XMark.

To check the claim presented in the introduction that my parallel methods work for any reasonable extension of TwigStack, I conducted extensive experiments with TSGeneric+ [55]. TSGeneric+ improves TwigStack’s running time by more than 2 times (by changing the getNext function). I first implemented the sequential version of TSGeneric+. I then created a parallel versions of the TSGeneric+ algorithm by using my methods of parallelization with the TSGeneric+ algorithm as the base algorithm (instead of TwigStack). I compared the performance of the parallel version of TSGeneric+ to the sequential version on the XMark benchmark. The results are presented in Figure 4.9. Observe a speedup of up to 20 in the runtime. This improvement is of the same scale as the improvement I obtain while using TwigStack as the base search algorithm.

I compared the performance of PTS to that reported in other published
papers on parallelizing the TwigStack algorithm. In these comparisons I used the same twig patterns as those reported in these papers, and on documents that have approximately the same size as the documents in the relevant papers. I compared with papers [23] and [67]. I could not perform an exact comparison with [69]. They tested on documents with a minimal size of 1GB, while, due to my system limitations, the largest document size that I could test is 520MB. The systems used by [23] and [67] are more powerful than my system. They have used Intel Dual core processors which are much faster than the Sparc system that I used. In comparing with [67], I ran 3 different twig patterns. (1) The twig pattern "/proceedings[editor] [ee]/year" on a document with size 450MB using 12 threads. My best parallel running time is 1.01 seconds. Whereas, according to [67], their best parallel running time on a document of size 490MB is 27 seconds. (2) The twig pattern "/category[name]/description//text" on a document with size 524MB. My best parallel running time is 3.7 seconds, whereas [67] reports a running time of 15 seconds for their best parallel scheme on a document of size 450MB. (3) The last twig pattern "/open_auction//annotation [author][description]//happiness" on a document with size 524MB. My best parallel running time is 1.01 seconds, whereas [67] reports a running time of 10.7 seconds for their best parallel scheme on a document of size 450MB. I performed approximate comparisons with [69] for queries Q1,Q2 and Q3 (as they appear in [69]) on a document of size 0.5GB. PTS provided speedups of 4.8 for Q1, 4.6 for Q2 and 4.4 for Q3.
For a 1GB document, [69] reported speedup of 3.3 for Q1, 3.8 for Q2, and 3.1 for Q3. If I take into account that I experimented on an old Sparc System with just 8 cores, while [69] used a few four-core Opteron processors, which is a newer and more powerful platform, my results indicate a competitive potential. I compared PTS to the work in [23]. I ran the twig pattern "//site//closed_auctions//closed_auction//annotation//description//text//keyword" on a document of size 112MB. The PTS running time was 0.4 seconds, while [23] reported 1.65 seconds for a 116MB document.

Table 4.1 presents parsing time and consumption of main memory for the tested documents. For ease of experimentation, the parsed data was held in main memory. The in-memory size listed in the table includes the size of all the memory occupied by the algorithm during execution. Observe that the time for parsing the document depends linearly on the size of the document. Also, note that the parsed data of a document with size $M$ occupies about $3.5M$ in main memory. It is important to note that the parsing work is done only once while starting up the system. From the moment the parsing has finished, any number of queries can be performed, so the relatively long parsing time is still negligible as compared to the queries processing time.
Table 4.1: Parsing time and size in main memory of the tested documents

<table>
<thead>
<tr>
<th>Document Size (MB)</th>
<th>Parse Time (sec)</th>
<th>Size In Main Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>72</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>78</td>
<td>360</td>
</tr>
<tr>
<td>200</td>
<td>133</td>
<td>600</td>
</tr>
<tr>
<td>525</td>
<td>339</td>
<td>1690</td>
</tr>
</tbody>
</table>

Table 4.1: Parsing time and size in main memory of the tested documents
4.5 Using PTS for Non-Main-Memory Resident (NMMR) Documents

The scheme for processing PTS as presented till now requires that all the relevant streams are fully uploaded to main memory. This requirement cannot always be met for large documents, as the relevant streams size may be larger than the main memory size. In this section I outline a scheme that overcomes this limitation.

The NMMR scheme: In the NMMR scheme, after parsing the document, I create a binary file on disk for each L-Stream. This file holds a list of data units, one data unit for each node. A data unit holds the encoding of the relevant information for the corresponding node. For example, for some node \( n \) that resides in L-Stream \( s \), the data unit consists of 3 elements which are “3”, “78”, and “2”, where “3” is the LeftPos, “78” is the RightPos, and “2” is the LevelNum. So, when a thread needs some data from one of the streams, it has to perform a reading operation of the required node from the matching file.

Suppose I try to run PTS on large documents exactly in the way I run it for small documents. Of course, the main limitation of this approach would be the enormous disk access times. To solve this problem I designed and implemented a special parallel scheme for running PTS on NMMR documents.

As in the original scheme, the first step is to choose an appropriate partition (the ordinary partition P or partition twigP), and the second step is to pro-
cess the query using the chosen partition as a pool of tasks. In this scheme for twig pattern queries I always choose partition twigP, as in large documents the number of subtrees in the twigP partition is much larger than the number of threads. A `prefetchTaskArray` is an array of `prefetchTask` units. Each `prefetchTask` unit has the following data: (1) `isReady`, a boolean flag that indicates if the task is already ready to be processed. (2) `isProcessed`, a boolean flag that indicates if the task was already processed. (3) `taskData`, a memory area that holds all the data required to process the current task.

Basically, for processing a query, I run two groups of threads in parallel. The `prefetching group` is in charge of preparing tasks for processing, while the `processing group` is in charge of processing the prepared tasks.

Each `Prefetching group` thread `p` is assigned the responsibility for part of the `prefetchTaskArray`. Thread `p` looks for `prefetchTasks` that were already processed (i.e., the value of `isProcessed` is `true`) but still not prepared (i.e., the value of `isReady` is `false`). When `p` finds such a task, say `t`, it performs the following: (1) resets the `isProcessed` flag; (2) takes a new portion of subtrees, `stPortion`, from P according to the `numOfTasks` parameter (as defined in this Chapter); (3) finds the limits of the sub-streams that are matching to `stPortion` using the `findSubStreams(stPortion)` function (as described in this Chapter); (4) uploads the relevant sub-streams (according to the found limits) to the `taskData` memory area; (5) sets the `isReady` flag.

Thread `p` repeats the above, until all the subtrees in the `prtn` are processed. When all the subtrees in P are processed it assigns the value `true` to a special
Boolean flag called \textit{finishedQ}.

Each \textit{Processing group} thread \(p\) looks for \textit{prefetchTasks} in the \textit{prefetchTaskArray} that were already prepared (i.e., the value of \textit{isReady} is \textit{true}) but still not processed (i.e., the value of \textit{isProcessed} is \textit{false}). When \(p\) finds such a task, say \(t\), it performs the following: (1) resets the \textit{isReady} flag; (2) processes the \textit{runPathStack} function (if the query is a path) or the \textit{runTwigStack} function (if the query is a twig) on the data that is placed in the \textit{taskData} memory area; (3) sets the \textit{isProcessed} flag to \textit{true}. Thread \(p\) repeats the above, until it finds that the \textit{finishedQ} is \textit{true} and that there are no more ready unprocessed tasks in the \textit{prefetchTaskArray}.

4.5.1 Experimenting with Large NMMR Documents.

I performed preliminary experimentation of the new NMMR scheme with extremely large data sets of up to 30GB. As in other experiments, I compared the runtime of the sequential TwigStack algorithm to the runtime of my scheme. I ran these experiments on a 3 GHz ShadyCove 5520 workstation with 12 cores. I experimented with large documents that cannot be held fully in RAM. I could not experiment with real Terabyte sized documents as my hard disk size was just 1TB. During experimentation, I found the best proportion between the size of the \textit{prefetching group} and the size of the \textit{processing group} to be 1 : 1, i.e., for each processing thread I need one prefetching thread. Figures 4.11 and 4.12 present representative results of the above experiments. I compared the scheme to (1) a naive implementation
of TwigStack working directly on the disk, and (2) an optimized version of TwigStack in which large data chunks of about 10K nodes are read, buffered and processed \((\text{optimized}, \text{ at point } X = 1)\). I obtained an average speedup of about 30 compared to \textit{naive} and of about 2 compared to \textit{optimized}. The presented results are for Twig Patterns 1 (Figure 4.11) and Twig Pattern 2 (Figure 4.11).

![Graph showing speedup vs. number of threads]

Figure 4.11: NMMR scheme experiments with very large documents for PTS with Twig Pattern 1. X GB chart line name means that the experiment was performed for the X GB-sized document

The likely explanation is that disk access times are dominant and that in order to significantly benefit from parallelism for NMMR documents, disk data must be replicated, or distributed, over a number of disks that may be accessed in parallel.
Figure 4.12: NMMR scheme experiments with very large documents for PTS with Twig Pattern 2. X GB chart line name means that the experiment was performed for the X GB-sized document.
Chapter 5

The GPU-Twig Algorithm

This part of the research was published in a ADMS@VLDB workshop 2012 paper [88].

The GPU-Twig algorithm is a SIMD algorithm. The main advantage of the GPU-Twig algorithm is the ability to divide the work to hundreds or even thousands of threads that run in parallel. The GPU-Twig algorithm does not make use of some of the GPU features, such as shared memory or coalesced memory access (these features are described in Section 2.7). The main used GPU-specific features are as follows. The first is the massive parallelization. The second is utilizing the GPU’s high bandwidth in transferring data from the global memory to the local memories, which is more efficient than data transfer from main memory to the CPU. The limiting factor of the algorithm’s performance, i.e., the algorithm’s bottleneck, is mainly the global memory bus bandwidth, as the algorithm is not compute-intensive.
GPU-Twig processes only the document parts that are relevant to the input query, as it processes only streams whose labels appear in the input query. In the first phase of the algorithm, it iterates over all the relevant streams and derives additional structural information according to the query. In the second phase of the algorithm, this structural information is used to produce the answer set.

To gain intuition about the algorithm, I start with an example. Next, I present the formal definitions and the algorithm itself. Finally I present a correctness proof for the algorithm. Experiments are presented in Section 5.5.

5.1 An Intuitive Example

Consider the qTree and dTree presented in Figure 5.1. Assume I want to find all nodes c that are part of the match between the qTree query tree and the dTree data tree. As part of the first step of the algorithm, I derive additional structural information about all dTree nodes that are potentially part of such matchings, i.e., all nodes with labels a, b, c d and e (the labels that appear in qTree). The additional derived structural information addresses the following question: for a node n in dTree with one of the above labels, is it potentially a part of some matching? For that, I use the qArray structure of node n (the small bitmap that can be seen on the left side of each node). The indices of the qArray are the indices of qTree nodes. Intuitively, bit i
in the $qArray$ of node $n$ (in $dTree$) is set to true if (1) there exist a node $q$ (in $qTree$) with the same label as $n$ and (2) the following holds: $q$ has a $qChild$ $qC$ with index $i$ in $qTree$, and $n$ has a descendant $nD$ such that the subtrees rooted at $qC$ and $nD$ match.

To resolve the question whether some node $n$ is potentially a part of some match, I start to look first at data nodes that are labeled the same as leaf nodes in $qTree$. For example, consider node $nD=(9:10,5)$ in $dTree$. Clearly, there is a match between $nD$ and node 4 in $qTree$. This information is recorded in the $qArray$ structure of all ancestors of $nD$ that are labeled $b$, the same as the $qParent$ of node 4 in $qTree$ (node 3). Only node $n=(5:12,3)$ is an ancestor labeled $b$ of $nD$, thus, I set bit 4 in the $qArray$ of $n$ to true. In a similar manner, I set bit 5 in the $qArray$ of $n$ to true.
Next, I look at the node $nD=(5:12,3)$ labeled $b$, and check if it is part of a match between the subtree rooted at $nD$ and the subtree rooted at query node 3. There is such a match since $nD$ has a child labeled $e$ and a descendant labeled $d$ in the data tree. I know that this match exists since I see that bits 4 and 5 in the $qArray$ of $nD$ are true. The information regarding the match between the subtrees of $nD$ and of node 3 is recorded in the $qArray$ structure of all ancestor nodes of $nD$ that are labeled the same as the $qParent$ of node 3 (query node 1). The only relevant ancestor is node $m=(2:15,2)$, thus I set the value of bit 3 in the $qArray$ of node $m$ to true. In a similar manner, I compute the structural information in all relevant nodes in $dT ree$ (i.e., all nodes with labels $a$, $b$, $c$, $d$ and $e$).

For the second step of the algorithm I first define a twig pattern, $qPath$, formed by the path from the answer node $ansQ$ in $qTree$, labeled $c$, to the root of the query tree in $qTree$. A node-match between node $q$ in $qPath$ and node $n$ in $dT ree$ exists if the following holds: (1) $n$ and $q$ have the same label; (2) The $qArray$ of $n$ represents a match between subtrees rooted at $n$ in $dT ree$ and $q$ in $qTree$. A path-matching between path $qPath$ and node $n$ exists if: (1) there exist a node-match between $ansQ$ and $n$; (2) for the rest of the nodes in $qPath$, there exist a node-match between each node $q$ in $qPath$ and some node $pn$ in $dT ree$ such that the structural (parent-child and ancestor-descendant) relationships between $qPath$ nodes are satisfied by the corresponding $dT ree$ nodes.

In the second step of the algorithm, I check for each node $n$ labeled $c$ in
if there exist a path-matching between \( qPath \) and \( n \). This check is done using the relevant \( qArray \) structures of \( dTree \) nodes. Nodes labeled \( c \) that conform to the above path-matching are added to the output answer set. In the example, node \((3,4:3)\) labeled \( c \) satisfies the above path-matching: there is a node-match between node \((3,4:3)\) in \( dTree \) and node 6 in \( qTree \) and a node-match between node \((2,15:2)\) in \( dTree \) and node 1 in \( qTree \). There is a node-match between node \((2,15:2)\) and node 1 since I see that bits 2, 3 and 6 in the \( qArray \) of node \((2:15,2)\) have value \textit{true}. Therefore, node \((3:4,3)\) is added to the output set. In contrast, node \((14:15,3)\) is not added to the answer set. This is because there is no match between the node \((13:20,2)\) in \( dTree \) and node 1 in \( qTree \). This is because bit 3 in the \( qArray \) of node \((13:20,2)\) has the value \textit{false}, which means that node \((13:20,2)\) does not have descendant labeled \( b \) such that it has descendants labeled \( d \) and \( e \).

Note, that according to the defined query semantics it could be possible that \( c \) labeled node would be in the same subtree as \( b \) labeled node.

### 5.2 Definitions

I define a \( qChild \) as the child of node \( q \) in \( qTree \). Next, I define when a node \( n \) of data tree \( dTree \) is \textit{qualifying} w.r.t. a node \( q \) of twig pattern \( qTree \). The intuition behind this definition is that if \( n \) is \textit{qualifying} w.r.t. \( q \), then the subtree rooted at node \( n \) satisfies the query requirements of the subtree rooted at node \( q \). So, it means that \( n \) is part of at least one match to the
Definition 1: Let \( q \) be a node in \( qTree \) labeled \( qLabel \), let \( n \) be a node labeled \( qLabel \) in \( dTree \). I define, inductively on the height \( h \) of \( qTree \), when \( n \) is qualifying w.r.t. \( q \) as follows.

\( h = 0 \), \( q \) is a leaf node in \( qTree \): Node \( n \) is qualifying w.r.t. node \( q \) if they both have the same label, namely \( qLabel \).

\( h > 0 \), \( q \) is a non-leaf node in \( qTree \): Node \( n \) is qualifying w.r.t. node \( q \) if:

1. \( n \) and \( q \) have the same label, namely \( qLabel \).
2. For each child node \( qChild \) of \( q \) in \( qTree \) there exist a descendant node \( nD \) of \( n \) such that \( nD \) is qualifying w.r.t \( qChild \). The order between the \( qChild \) nodes of \( q \) does not have to be preserved by their bijection images. ■

Example 1. For the twig pattern \( qTree \) and the data tree \( dTree \) in Figure 5.1, nodes (5:12,3) and (16:19,3) in \( dTree \) are qualifying w.r.t. the node with index 2 in \( qTree \). Node (5:12,3)) is qualifying w.r.t. the node labeled \( b \) with index 3 in \( qTree \). This is because it has a descendant labeled \( d \) (9:10,5) that is qualifying w.r.t. the node labeled \( d \) with index 4, and a descendant labeled \( e \) (6:7,4) that is qualifying w.r.t. the node labeled \( e \) with index 5.

Node (2:15,2) is qualifying w.r.t. the node with index 1 in \( qTree \). This is because it has child labeled \( c \) (3:4,3) that is qualifying w.r.t. the node labeled \( c \) with index 6, a descendant labeled \( b \) (node (5:12,3)) that is qualifying w.r.t. the node labeled \( b \) with index 3, and which is also qualifying w.r.t. node \( b \) with index 2. □

Each node in a L-Stream has a field named \( qArray \) which is a Boolean array.
The qArray is used to encode the additional structural information that is derived in the first phase of the algorithm. In the following definition I define when bit $i$ in the qArray of node $n$ is set correctly w.r.t. some node $q$ in $qTree$, where $0 \leq i <$ the number of nodes in $qTree$. Intuitively, if bit $i$ in the qArray of node $n$ labeled $qLabel$ is set correctly w.r.t. node $q$ labeled $qLabel$ and its value is true, then the following holds: node $n$ has some descendant node $m$ whose subtree satisfies the query requirement w.r.t. the subtree rooted at a $qChild$ $qC$, indexed $i$, of node $q$. And conversely, if the bit is false then there is no such descendant satisfying the requirement.

**Definition 2:** Let $n$ be an arbitrary node in $dTree$ labeled $qLabel$. Let $q$ be a node labeled $qLabel$ in $qTree$. Let $qC$ be a $qChild$ of $q$ indexed $i$ and labeled $decLabel$. Bit $i$ in the qArray of node $n$ is set correctly w.r.t. node $q$ if the following holds. The value of bit $i$ is true iff: (1) node $n$ has a descendant node $m$ labeled $decLabel$, and (2) node $m$ is qualifying w.r.t. $qC$. ■

To indicate if the subtree rooted at node $n$ (in $dTree$) satisfies the query requirements w.r.t. the subtree rooted at node $q$ (in $qTree$) using the qArray, I define *subtree-correctness*.

**Definition 3:** The qArray structure of node $n$ labeled $qLabel$ is subtree-correct w.r.t. node $q$, labeled $qLabel$ in $qTree$, if for every $qChild$ of $q$, say with index $i$, bit $i$ of the qArray is set correctly, and its value is true. ■

The qArray structure stores information that is applicable to the, potentially many, nodes labeled $qLabel$ in the $qTree$ pattern.
Example 2. For the twig pattern \( qTree \) and for the \( dTree \) of Figure 5.1, node \((2:15,2)\) is subtree-correct w.r.t. the query node with index 1. This is because, for every \( qChild \) of the node with index 1 in \( qTree \), say with index \( i \) (\( qChild \) nodes indices are 2,3,6), bit \( i \) of the \( qArray \) of node \((2:15,2)\) is set correctly. Bit 2 is set correctly to \( \text{true} \) because node \((2:15,2)\) has a descendant that is qualifying w.r.t. the node with index 2 (node \((5:12,3)\)). Bit 3 is set correctly to \( \text{true} \) because node \((2:15,2)\) has a descendant that is qualifying w.r.t. the node with index 3 (node \((5:12,3)\)). Bit 6 is set correctly to \( \text{true} \) because it has a child that is qualifying w.r.t. the node with index 6 (node \((3:4,3)\)). □

5.3 The Algorithm

The algorithm has two phases. The first phase derives the additional structural information for all the nodes in the streams that correspond to the twig pattern nodes. The first phase of the algorithm operates in a bottom up manner over the \( qTree \) twig pattern. It chooses some node \( q \) such that all its \( qChild \) nodes are already processed. I.e., the first nodes to be processed are the leaves. I define \( qParent \) as parent of node \( q \) in \( qTree \). For every node \( q \) in \( qTree \), the algorithm processes all the nodes in the L-Stream that corresponds to \( q \), i.e., have the same label. Suppose that the label of \( q \) is \( qLabel \), the index of \( q \) in \( qTree \) is \( qIdx \), and the \( qParent \) of \( q \) is node \( pq \) labeled \( pqLabel \). For every node in the \( qLabel \) L-Stream, the first phase of the
algorithm updates all the relevant qArray structures of nodes in the pqLabel L-Stream, such that after processing all the nodes in the paLabel L-Stream, for each node \( n \) in the pqLabel L-Stream, bit \( qIdx \) in the qArray of node \( n \) is set correctly w.r.t. \( pq \); in particular, if \( n \) has a descendant in the qLabel L-Stream that is qualifying w.r.t. \( q \), bit \( qIdx \) is set to \textit{true} (and if it does not have such a descendant it is set to \textit{false}).

Note that I can process in parallel qTree nodes if all their qChild nodes were already processed. As the processing of node \( q \) relies only on the fact that all its children are already processed.

In terms of GPU implementation, before starting the first phase of the algorithm, I do the following. First, all the relevant information for the query is copied using \textit{cudaMemcpy} calls. This information includes the streams corresponding to query nodes and the query nodes data itself. Second, all the qArray fields in all the streams are initialized to zero.

Figure 5.2 presents the first phase of the GPU-Twig algorithm. The input to the first phase are the data tree \( dTree \) and the twig pattern \( qTree \). Line 4 contains the invocation of the CUDA kernel function \textit{gpuTwigFirstPhase} which is processed on the GPU. I.e., the derivation of the additional structural information of \( dTree \) is executed on the GPU, while the CPU task is to run the outer loop over \( qTree \) nodes, and to prepare the information for the GPU.

The \textit{gpuTwigFirstPhase} kernel call \textit{sets correctly} bit \( qIdx \) in the qArray of all nodes \( pm \) in the pqLabel L-Stream w.r.t. node \( pq \), for each node \( pq \) in
The code of *gpuTwigFirstPhase* is run for every node in *qStream*, which is the "Multiple Data", while the "Single Instruction" is the code of the *gpuTwigFirstPhase* function itself. This provides the potential for an enormous number of parallel threads, as the number of nodes in *qStream* can be in the tens and even hundreds of thousands. According to the GPU GTX 480 architecture, the maximum number of resident threads per MP (multiprocessor) is 1536 (i.e., 1536*15 for all the MPs), while the number of threads that are executing at each moment of time in the MP is 32 (other threads can be waiting for data from the global memory, or just waiting for their turn to be run). So, the maximum number of threads that can actually run in parallel at any point of time is 480 (32 on each of the 15 MPs). As the number of nodes in a L-Stream is usually much larger than the number of compute units in the GPU, the utilization of the GPU is very high, i.e., the throughput of processing the work is high in comparison to multi-threaded CPU systems. The number of nodes in different streams can vary. This fact does not present a major performance problem, as usually the number of nodes in a L-Stream of a large document is much larger than the number of GPU processors.

Note that, for speeding up job processing, the algorithm uses mainly one feature of the GPU, namely that of high parallelism. It does not use the shared memory (of the processor), as each piece of data has to be read only once. Line 1 assigns a task to the current thread according to CUDA’s semantics. Line 2 checks if the *idx* of the current thread refers to existing
node. A case in which the number of kernel threads is smaller than the number of nodes in the relevant L-Stream is impossible. This is due to the way I calculate the parameters of the kernel that one of them is the required number of system threads. Line 4 checks if node \( n \) fulfills the conditions needed to set bit \( qIdx \) in the \( qArray \) of node \( pn \) according to Definition 2. I.e., whether node \( n \) is qualifying w.r.t. \( q \) (checked by the \( subtreeCorrect \) function). In case it is, I run the \textbf{foreach} loop at lines 5-7. The loop in line 5 is run only for nodes labeled \( pqLabel \) that are ancestors of node \( n \). For each ancestor node \( pn \) of node \( n \) I set bit \( qIdx \) of its \( qArray \) to \texttt{true} using the \texttt{atomicAssign} operation (line 6). I use \texttt{atomicAssign} to prevent a possible race condition between two threads that try to concurrently change the \( qArray \) field of some node \( pn \). To find which nodes in the \( pqLabel \) L-Stream are ancestors of \( n \), I use the \texttt{ancStreamL} list of \( n \) (as explained in 2.2.3) and take from there the index \( aIdx \) in the \( pqLabel \) L-Stream of a node that belongs to the path between \( n \) and the root which is closest to the root. To find all other ancestor nodes of \( n \) in \( pqStream \) I iterate in \( pqStream \) starting from \( aIdx \). For each node \( na \) starting from index \( aIdx \), I check whether the \texttt{leftPos} of \( na \) is smaller than the \texttt{leftPos} of \( n \) and the \texttt{rightPos} of \( na \) is higher than the \texttt{rightPos} of \( n \). When I find the first node that does not satisfy the above requirements, it means I iterated over all ancestor nodes of \( n \) in the \( pqLabel \) L-Stream.

If the query pattern is allowed to contain the child axis, then line 5 of the algorithm is changed as follows. In case that the edge between node \( q \) and
its parent is parent-child (and not ancestor-descendant), then instead of the
**foreach** loop in line 5, I need to set bit \(qIdx\) of the relevant \(qArray\) only for
a single node, which is the parent node of \(n\) labeled \(pqLabel\) (if such a node
exists). Parent nodes can be identified easily, by checking that the level of a
potential parent node is one less than the level of node \(n\). I.e., the **foreach**
is not needed any more in that case.

The \(subtreeCorrect(q, n)\) function checks if node \(n\) is \(subtree-correct\) w.r.t.
\(q\) according to Definition 3. The invariant maintained by this function is
that the value of all the bits in the \(qArray\) of \(n\) that correspond to the
\(qChild\) nodes of \(q\) are already set correctly w.r.t. \(q\). In Section 5.4, I prove
that when the \(subtreeCorrect\) function is called for some \(q\) and \(n\), the above
invariant holds. Thus, to check if node \(n\) is \(subtree-correct\) w.r.t. \(q\), according
to Definition 3, it suffices to check if for every \(qChild\) of \(q\), say with index
\(i\), bit \(i\) of the \(qArray\) of \(n\) is true. This is exactly what is done in the
\(subtreeCorrect\) function in lines (2-6). When I find that some bit \(i\) in the
\(qArray\) of \(n\) that corresponds to some \(qChild\) of \(q\) is false (line 4), it means
that \(n\) is not \(subtree-correct\) w.r.t. \(q\).

**Example 3.** I continue examining the scenario of Example 2. Consider the
twig pattern \(qTree\) and the \(dTree\) presented in Figure 5.4. After processing
the first phase of the algorithm for all the leaves of \(qTree\), the \(qArray\) of nodes
\((2:15,2)\), \((5:12,3)\), \((13:20,2)\) and \((16:19,3)\) in \(dTree\) are changed as shown in
Figure 5.4. The reason is that each one of these nodes has some descendant
with one of the leaves labels. For example node \((2,15:2)\) has descendants with
labels \( b \) and \( c \) according to the requirements of the node labeled \( a \) with index 1 in \( qTree \), so bits 2 and 6 of the \( qArray \) of node \((2:15,2)\) are set to \( \text{true} \). During the first phase of the algorithm for the node with index 3 (labeled \( b \) and heading a 3 nodes sub-tree of \( qTree \)), bit 3 in \( qArray \) of node \((2:15,2)\) is set to \( \text{true} \), because the \( qArray \) of node \((5:12,3)\) is \( \text{subtree-correct} \) w.r.t. the node with index 3. By this, the first phase of the algorithm is finished. \( \square \)

The goal of the second phase of the algorithm is to find the answer nodes for \( qTree \) in \( dTree \) using the \( qArray \) fields of nodes in \( dTree \) that were derived in the first phase of the algorithm.

The second phase of the algorithm processes all nodes in the L-Stream \( s \) that corresponds to the label of the \( qAnsN \) target node whose \( \text{isAnswer} \) field (see Chapter 2.2.1) value is \( \text{true} \). This is because each node \( currSN \) in L-Stream \( s \) is potentially an answer. I define a sub-query \( qPath \) as the path between the twig pattern root node and the answer node \( qAnsN \). A node-match between node \( q \) in \( qPath \) and node \( n \) in \( dTree \) exists if the following holds: (1) \( n \) and \( q \) have the same label; (2) The \( qArray \) of \( n \) represents a match between subtrees rooted at \( n \) and \( q \).

In order for a node \( currSN \) to be an answer, I need to check that there is at least one path-matching between path \( qPath \) and node \( currSN \). The algorithm proceeds bottom up. It starts from node \( currSN \) and checks if \( currQN \) that is initialized to \( qAnsN \)’s \( qParent \) maps to some ancestor of node \( currSN \), called \( ancN \), such that the \( qArray \) of \( ancN \) is \( \text{subtree-correct} \) w.r.t. \( pCurrQN \). In case that such \( ancN \) is found, I update \( currSN \) to be
ancN and currQN to be pCurrQN, otherwise I stop. This is because not finding such an ancN means that the match I am looking for does not exist. I continue this process until I find the desired match, or until I fail to find it (as explained above).

Figure 5.3 presents the second phase of the GPU-Twig algorithm. The input to the second phase are the source document tree dTree, and the twig pattern qTree. The output of the second phase is a set ansSet with all the answer nodes for qTree in dTree. Line 2 contains the invocation of the CUDA kernel function called gpuTwigSecondPhase that is processed on the GPU. In line 3, I prepare the ansSet set, by scanning the ansLabel L-Stream for nodes whose isAnswer bit is set to true.

The gpuTwigSecondPhase kernel call sets the isAnswer bit of all the answer nodes for qTree in dTree (this bit is initialized to false). The code of gpuTwigSecondPhase is run for every node in qStream. As before, this provides a potential for an enormous number of concurrently executed threads, one per qStream node. Lines (6-20) check that there is at least one match of qPath (defined above) in dTree such that the qAnsN node maps to n (line 6) and the qArray of each node in the match is subtree-correct w.r.t. the matching node in qPath (line 15). To check that node currSN has no ancestor labeled pCurrQL (line 11), I use the ancStreamL list of currSN (as explained in 2.2.3) and check if the pCurrQL label is part of it. In case it is not, I know that node currSN has no ancestor labeled pCurrQL. Line 21 checks if a qPath match was found, and in case of a positive answer, line 22
sets the \textit{n.isAnswer} bit to \textit{true}. If the query pattern is allowed to contain the child axis, then if the edge between $pCurrQN$ and $currQN$ is parent-child (and not ancestor-descendant), then instead of lines 11-13, I need to check if $CurrSN$ has a parent node in $pCurrQN$. In case of a positive answer, I need to run lines 15-16 and to continue at line 18, otherwise, to go to line 18 (without running lines 14,15). I.e., the \texttt{foreach} loop is run just once for the found parent node, or zero times, if such parent node does not exist.

\textit{Example 4}. This example is a continuation of Example 3 and is based on Figure 5.1. The value of the \textit{isAnswer} field in the node labeled $c$ (with index 6) is \textit{true}. So, according to the definition of answer nodes, the potential answer nodes are contained in the $c$ L-Stream. The $c$ L-Stream contains two nodes: $(3:4,3)$ and $(14:15,3)$.

First I describe the run of the second phase for node $(3:4,3)$. The \textit{qParent} of the node labeled $c$ in $qTree$ is the node labeled $a$. The only labeled $a$ ancestor node of $(3:4,3)$ is $(2:15,2)$. Node $(2:15,2)$ has the highest \textit{LeftPos} value in $a$’s L-Stream such that it is an ancestor of node $(3:4,3)$ and is \textit{subtree-correct} w.r.t. node 1 labeled $a$. I know that node $(2:15,2)$ is \textit{subtree-correct} w.r.t. node 1 labeled $a$ because the \textit{qArray} structure of node $(2:15,2)$ is \textit{subtree-correct} w.r.t. the node labeled $a$ (in $qTree$). So, I update \textit{currSN} to be $(2:15,2)$. As the node labeled $a$ is the root of $qTree$, the while loop ends and since \textit{currSN} is the query node labeled $a$, I add node $(3:4,3)$ to \textit{ansSet} (set of all answer nodes).

Lastly, I describe the run of the second phase for node $(14:15,3)$. The \textit{qParent}
node of the node labeled $c$ in $qTree$ is the node labeled $a$. The only ancestor node of $(14:15,3)$ with label $a$ is $(13:20,2)$. Node $(13:20,2)$ has the highest $LeftPos$ value in $a$’s L-Stream but, it is not qualifying w.r.t. $currQ$, because the $qArray$ of node $(13:20,2)$ is not subtree-correct w.r.t. the node labeled $a$ (in $qTree$). This is because bit 3 in the $qArray$ of $(13:20,2)$, that corresponds to node 3 in $qTree$, is false. As the node with label $a$ is the root of $qTree$, the while loop ends. As I did not update $currSN$ to be $(13:20,2)$, I do not add node $(14:15,3)$ to $ansSet$. □

5.4 Correctness Proof

Correctness Proof for the First Phase of the Algorithm

Lemma 1: For node $q$ labeled $qLabel$ in $qTree$. If the $qArray$ structure of $dTree$ node $n$ labeled $qLabel$ is subtree-correct w.r.t. $q$ then $n$ is qualifying w.r.t. $q$, and $n$ is part of at least one match of the $qTree$ sub-query rooted at $q$ in $dTree$.

Proof: Let $q$ be a node in $qTree$ labeled $qLabel$, let $n$ be a node labeled $qLabel$ in $dTree$, and let $h$ be the height $h$ of $qTree$.

For $h = 0$ and $q$ a leaf node in $qTree$. By Definition 1, any $dTree$ leaf node $n$ labeled $qLabel$ (as $q$) is qualifying w.r.t. $q$. Node $q$ is a leaf node, thus it is qualifying w.r.t. $q$ no matter how $qArray$ is filled.

For $h > 0$ and $q$ a non-leaf node in $qTree$. If the $qArray$ structure of node $n$ labeled $qLabel$ is subtree-correct w.r.t. $q$ then according to the definition of
subtree-correct, for every qChild of q, say with index i, bit i of the qArray is set correctly and its value is true. According to the set correctly definition, the value of bit i in the qArray of node n is true iff node n has a descendant node m labeled with the same label as qChild qC with index i such that m is qualifying w.r.t. qC. Thus, for each qChild qC with index qIdx of node q, the bit qIdx in qArray of node n is true, which means that for each qChild qC of node q in qTree node n has a descendant node m in dTree such that node m is qualifying w.r.t. node qC. So, according to Definition 2, n is qualifying w.r.t. q, and n is part of at least one match of the qTree sub-query rooted at q in dTree. ■

The following definition captures the desired correctness of information conveyed by the qArray structure w.r.t. a particular query node q.

**Definition 4:** For twig pattern qTree and for node q labeled qLabel in qTree, dTree is tree-correct w.r.t. q, if for each node n, labeled qLabel, in dTree that is qualifying w.r.t. q, the qArray of node n is subtree-correct w.r.t q. And, for each node n, labeled qLabel, in dTree that is not qualifying w.r.t. q, for every qChild of q, say with index i, bit i of the qArray of node n is set correctly but n is not subtree-correct w.r.t. q (i.e., bit i is false). ■

I assume that before starting the algorithm the qArray structures of all nodes in dTree are initialized to zero (i.e., false).

**Lemma 2:** At the end of the run of the first phase of the algorithm, for all nodes q in qTree, dTree is tree-correct w.r.t. q.

**Proof:** By induction over the structure of qTree.

129
**Basis:** For any leaf node $lfN$ labeled $qLabel$, $dT ree$ is tree-correct w.r.t. $q$.

According to the *qualifying* definition, each node $lfN$ that is labeled $qLabel$ is always *qualifying* w.r.t. $q$ without additional requirements. According to the *subtree-correct* definition $lfn$ and $q$ share the same label, there are no additional requirements for the $qArray$ of $lfN$ (as it does not have $qChild$ nodes), so $lfn$ is also *subtree-correct* w.r.t $q$. So, each $lfN$ that is qualifying w.r.t. $q$ (meaning that they share the same label) is also *subtree-correct* w.r.t. $q$.

I now prove that if $dT ree$ is tree-correct w.r.t. each $qChild$ node $qC$ of node $q$ in $qTree$, then $dT ree$ is tree-correct w.r.t. $q$.

**Induction hypothesis:** $dT ree$ is tree-correct w.r.t. all $qChild$ nodes $qC$ of $q$.

**Inductive step:** I show that the algorithm marks $dT ree$ such that $dT ree$ is tree-correct w.r.t. $q$. This is established using the following two Claims.

Assume that $q$ is labeled $qLabel$. Assume that $pq$ labeled $pqLabel$ is $q$’s $qParent$ (the first phase of the algorithm is run for any node except the root node, that is why $q$ always has a parent). Note that before starting the algorithm, all $qArray$ structures, of all nodes, are initialized to $false$. Node $n$ in $dT ree$ is *part of a match* if $n$ is the image of some node in $qTree$ in some match of $qTree$ in $dT ree$.

**Claim 1:** Assume $qC$, indexed $qcIdx$, is a $qChild$ of $q$. Assume $n$ is a node labeled $qLabel$ in $dT ree$. I claim that bit $qcIdx$ in the $qArray$ of $n$ is set *correctly* during the processing of $qC$. 


**Proof:** While $qChild$ $qC$ (indexed $qcIdx$ and labeled $qcLabel$) of $q$ was considered, I processed each node $nD$ in the $qcLabel$ L-Stream. By the algorithm, I set bit $cqIdx$ to true in all nodes $n$ in the $qLabel$ L-Stream that are ancestors of $nD$ in case that the following condition holds: For each $qChild$ $qCqC$ of $qC$ indexed $i$, the value of bit $i$ in the $qArray$ structure of $nD$ is true. According to Definition 3, if for each $qChild$ of $qC$, say with index $i$, bit $i$ of the $qArray$ of $nD$ is set correctly w.r.t. $qC$ and its value is true, then the $qArray$ structure of node $nD$ is subtree-correct w.r.t. $qC$. By the assumption, $dTree$ is tree-correct w.r.t. $qC$. So, based on Definition 3 and the assumption, if the requirement that for each $qChild$ $qCqC$ of $qC$ indexed $i$, the value of bit $i$ in the $qArray$ structure of $nD$ is true holds, it means that $qArray$ of $nD$ is subtree-correct w.r.t. $qC$.

So, the fact that I set bit $qcIdx$ to true is correct according to Definition 2, as I found that $nD$ is subtree-correct w.r.t. $qC$, and according to Definition 2 in that case bit $qcIdx$ has to be true.

If after finishing processing all nodes in the $qcLabel$ L-Stream, bit $qcIdx$ in a $dTree$ node $n$ labeled $qLabel$ was not set to true and its value at the end of the process remained false, it means that no node was found such that it satisfies the above requirement. The fact that no such node was found, means that node $n$ labeled $qLabel$ does not have any descendant that is labeled $qcLabel$ such that it satisfies the above requirement. This is because, if such a node exists, it must be part of the $qcLabel$ L-Stream, but I checked all nodes in $qcLabel$ L-Stream, and did not find such a node. So, if bit $qcIdx$
in node \( n \) was not set to \( true \) during processing nodes in \( qcLabel \) L-Stream, it must be because node \( n \) labeled \( qLabel \) does not have any descendant that is labeled \( qcLabel \) that satisfies the requirement of Definition 2.

So, the above process sets correctly the \( qcIdx \) bit in \( qArray \) of all nodes \( n \) with label \( qLabel \). Note that when some bit in \( qArray \) of any node is set to \( true \), its value does not change during the continuation of the algorithm. □

**Claim 2:** Now I claim that before starting to process node \( q \) in \( qTree \), \( dTree \) is tree-correct w.r.t. \( q \).

**Proof:** By the algorithm, node \( q \) processing starts only after finishing to process all the \( qChild \) nodes of \( q \). Based on Claim 1, for each \( qChild \) node \( qC \) of \( q \), the bit that corresponds to \( qC \) in the \( qArray \) structure of nodes labeled \( qLabel \) in \( dTree \) is set correctly while processing node \( qC \). So, after finishing processing all \( qChild \) nodes of \( q \), all bits that correspond to \( q \)’s \( qChild \) nodes in the \( qArray \) of nodes labeled \( qLabel \) in \( dTree \) are set correctly. Thus, according to Definition 4, \( dTree \) is tree-correct w.r.t. \( q \). □

This concludes the inductive step.

According to Claim 2 and since the algorithm continues until it processes all nodes in \( qTree \), and because \( dTree \) is tree-correct w.r.t. \( q \) as proved in the inductive step, at the end of the algorithm’s execution, for all nodes \( q \) in \( qTree \), \( dTree \) is tree-correct w.r.t. \( q \). ■

**Correctness proof for the second phase of the algorithm**

**Lemma 3:** At the end of the run of the algorithm, all the answer nodes of twig pattern \( qTree \) in \( dTree \) are added to the output.
The Lemma is established using the following three Claims.

**Claim 1:** For the subtree rooted at node $n$ in $dTree$, for node $m$ in this subtree, and for the subtree rooted at $q$ in $qTree$, if the $qArray$ of node $n$ is subtree-correct w.r.t. $q$, and $m$ is part of a match $mtch$ of the twig pattern defined by the subtree rooted at $q$ in the subtree rooted at $n$, and there exists some ancestor $ancN$ of node $n$ whose $qArray$ is subtree-correct w.r.t. the $qParent$ node $pq$ labeled $pqLabel$ of $q$, then $mtch$ can be extended to a match $mtchExt$ of the query defined by the subtree rooted at $pq$ in the subtree rooted at $ancN$ such that $m$ is part of the match $mtchExt$ matching the same $qTree$ node as in $mtch$.

**Proof:** According to problem definition, the $qArray$ of $ancN$ is subtree-correct w.r.t. $pq$, so according to Lemma 2, $ancN$ is part of at least one match of the twig pattern defined by the subtree rooted at $pq$ in the subtree rooted at $ancN$. Assume that the index of node $q$ is $qIdx$. Bit $qIdx$ in the $qArray$ of node $ancN$ is true, as according to the problem definition the $qArray$ of node $ancN$ is subtree-correct w.r.t. $pq$. According to the claim definition, the $qArray$ of node $n$ is subtree-correct w.r.t. $q$, so according to Lemma 2, $n$ is part of at least one match of the twig pattern defined by the subtree rooted at $q$ in the subtree rooted at $n$. Based on the above, I can extend the match of $q$ in the subtree rooted at $n$ to the match of $pq$ in the subtree of $dTree$ rooted at $ancN$. The mapping of nodes of $qTree$ subtree rooted at $q$ to nodes of $dTree$ subtree rooted at $n$ remains the same, and I add to this mapping the pair $(pq,ancN)$, i.e., $pq$ maps to $ancN$. So, as I
found a new (extended) match $mtchExt$, $m$ is part of a match of the $qTree$ subtree rooted at $pq$ in the subtree of $dTree$ rooted at $ancN$. □

Claim 2: For a $dT ree$ node $n$ labeled $qLabel$ and a $qTree$ node $q$ labeled $qLabel$, if the $qArray$ of node $n$ is subtree-correct w.r.t. $q$, and $n$ does not have any ancestor whose $qArray$ is subtree-correct w.r.t. $pq$, where $pq$ is the $qParent$ of $q$, then $n$ does not participate in any match of sub-query $pq$ in $dTree$.

Proof: Suppose, for the sake of deriving a contradiction, that the $qArray$ of node $n$ is subtree-correct w.r.t. $q$, and $n$ does not have any ancestor whose $qArray$ is subtree-correct w.r.t. $pq$ where $pq$ is the $qParent$ of $q$, and that $n$ participates in match $mtch$ of query $pq$ in $dT ree$. Match $mtch$ looks as follows: $(..., (q : n), (pq : pn))$. According to the definition of a match, each $qTree$ node maps to a node in $dT ree$, and the structural relationship between each $qParent$ $qChild$ pair of nodes in $qTree$ is satisfied by the corresponding $dT ree$ nodes. In match $mtch$, $q$ maps to $n$ and $pq$ maps to $pn$. According to match definition, there has to be an ancestor descendant relationship between nodes $n$ and $pn$, as $q$ and $pq$ have a $qParent$ $qChild$ relationship in $qTree$. According to the assumption, $n$ does not have any ancestor whose $qArray$ is subtree-correct w.r.t. $pq$ which contradicts the fact that $pn$ is an ancestor of $n$. Therefore, $n$ does not participate in any match of sub-query $pq$ in $dT ree$. □

Claim 3: At the end of the algorithm’s run, all the answer nodes of $qTree$ in $dT ree$ are added to the output.
Proof: I consider every node that can potentially be an answer node as I consider all nodes that belong to the ansLabel L-Stream. For each such node, say leafN, I run a while loop. In each loop iteration, I take the current node (namely, currS) and look for an ancestor (namely, ancN) such that the qArray of node currS satisfies the following condition: for each qChild of pCurrQ (the qParent of node currQ) indexed i, the value of bit i in the qArray structure of currS is true. Based on the correctness proof of phase one of the algorithm and on Definition 3, it means that the qArray structure of currS is subtree-correct w.r.t. pCurrQ. Each time I look for such an ancestor, I choose the ancestor with the highest lIdx value, which means the closest one to currS. This ensures that I never miss any potential matching data tree path. So, if such an ancestor does not exist, then based on Claim 2, currS is not part of any match, which means that leafN is also not part of any match. If I find such an ancestor ancN, according to Claim 1, node leafN is part of at least one match to the target node of the twig pattern defined by the subtree rooted at pCurrQ in the subtree rooted at ancN. □

After finishing the while loop, I check that currQ == rq (qTree’s root is rq) and that currS ≠ NULL (in line 21 of second phase I check that ancN ≠ NULL, but this check means the same, as the value of ancN is assigned to currS). According to Claim 1 it means that leafN is part of at least one match of the twig pattern defined by the subtree rooted at rq in the subtree rooted at currS. I.e., leafN is an answer node in dTree. Otherwise, by Claim 2, leafN is not an answer node. ■
It is easy to extend the above proofs to work for the case where parent-child edges in queries are allowed.
**Input:** 1) Data tree \( dTree \). 2) Twig pattern \( qTree \).

**Goal:** Record the additional structural information for all nodes in the streams that correspond to \( qTree \) nodes.

**Method (runs on the CPU):**
1. **while** there are unprocessed nodes in \( qTree \):
   2. Choose node \( q \) from \( qTree \) such that all its \( qChild \) nodes were already processed
   3. set \( qStream \) to \( q \)’s label L-Stream, \( qIdx \) to \( q \)’s index, \( pqStream \) to \( q \)’s \( qParent \)’s label L-Stream
   4. Invoke the CUDA kernel call for function:
      \( gpuTwigFirstPhase(qStream, pqStream, qIdx) \)
   5. Mark \( q \) as processed
   6. **end while**

// **gpuTwigFirstPhase** kernel function (runs on the GPU):

**Input:**
1) \( qStream \): \( q \)’s label L-Stream.
2) \( pqStream \): \( q \)’s \( qParent \) label L-Stream. //\( qParent \) is \( q \)’s parent in \( qTree \)
3) \( qIdx \) the index of \( q \) in \( qTree \).

**Goal:** For each node \( n \) in the \( pqStream \) bit \( qIdx \) in the \( qArray \) of node \( n \) is set correctly w.r.t. \( q \)’s \( qParent \). In particular, if it has such a descendant which is qualifying w.r.t. \( q \), bit \( qIdx \) is set to \( true \).

**Method:**
1. Set \( idx \) to a system assigned index of the current thread
2. if \( idx \geq \) number of nodes in \( qStream \) then **return**
3. Set \( n \) to node at index \( idx \) of \( qStream \)
4. if \( subtreeCorrect(q, n) == true \) then
5.   **foreach** node \( pn \) in the \( pqLabel \) L-Stream that is an ancestor of \( n \)
6.     **atomicAssign** \( pn.qArray[qIdx] \), true
7. **end foreach**

// **subtreeCorrect** function:

**Input:** 1) \( q \): node in \( qTree \). 2) \( n \): node in \( dTree \).

**Output:** Boolean value.

**Method:**
1. \( res = true \)
2. **foreach** \( qChild qC \) of \( q \)
3.   set \( i \) to the index of \( qC \)
4.   if \( n.qArray[i] == false \) then
5.     \( res = false; \) **break**
6. **end foreach**
7. **return** \( res \)

Figure 5.2: The first phase of the GPU-Twig Algorithm
**Input**: 1) Data tree \( dTree \). 2) Twig pattern \( qTree \).

**Output**: \( ansSet \), the set of all answer nodes of \( qTree \) in \( dTree \).

**Method (runs on the CPU)**:

1. SET node \( qAnsN \) to be the node in \( qTree \) whose \( isAnswer \) field value is \( true \) (the target node)
2. Invoke CUDA kernel call for function:
   \( \text{gpuTwigSecondPhase}(qAnsN, qTree) \)
3. Insert all nodes from the \( qAnsN \) L-Stream whose \( isAnswer \) bit is \( true \) to \( ansSet \)
   // \( \text{gpuTwigSecondPhase} \) kernel function (runs on the GPU):

**Input**:
1) \( qAnsN \): node \( qAnsN \) from \( qTree \).

**Goal**: find the answer nodes in the L-Stream of \( qAnsN \)

**Method**:

1. set \( idx \) to a system assigned index of the current thread
2. set \( qStream \) to be the L-Stream of (the label of) \( qAnsN \)
3. if \( idx \geq \) number of nodes in \( qStream \) then return
4. set \( n \) to the node at index \( idx \) of \( qStream \)
5. set \( currQN \) to \( qAnsN \), and \( currSN \) to \( n \)
6. IF \( \text{subtreeCorrect}(currQN, currSN) == false \) then return
7. set \( rq \) to be the root node in \( qTree \)
8. while the index of \( currQN \) > the index of \( rq \)
9.     set \( pCurrQN \) to the \( qParent \) of \( currQN \)
10. set \( pCurrQL \) to the label of \( pCurrQN \)
11. if \( currSN \) has no ancestor labeled \( pCurrQL \) then break
12. set \( upperL \) (respectively, \( lowerL \)) to the node with the smallest (respectively, largest) \( LeftPos \) value which is an ancestor of \( currSN \) and is in the \( pCurrQL \) L-Stream
13. set \( ancN \) to \( NULL \)
14. foreach node \( nn \) between \( lowerL \) and \( upperL \) in the \( pCurrQL \) L-Stream //lower to upper is crucial
15.     if \( \text{subtreeCorrect}(currQN, currSN) == true \) then
16.         set \( ancN \) to \( nn \); break
17. end foreach
18. if \( ancN == NULL \) then break
19. set \( currSN \) to \( ancN \) and \( currQN \) to \( pCurrQN \)
20. end while
21. if \( currQN == rq \) and \( ancN \) is not \( NULL \) then
22.    set \( n.isAnswer \) to \( true \)

Figure 5.3: The second phase of the GPU-Twig Algorithm
Figure 5.4: The state of the \textit{dT}ree after processing the first phase of the algorithm for all the leaves of \textit{qTree}. For example, bit 6 in node (13:20,2) is set to 1, i.e., node (13:20,2) is \textit{subtree-correct} w.r.t. the query node with index 6.
5.5 GPUTwig Experimental Evaluation

I compared GPU-Twig to the TwigStack algorithm [26] and to the parallel PTS algorithm [89, 87]. I essentially use the same L-Stream storage scheme in both algorithms (PTS and GPU-Twig); however, with a slightly different information in each node (for example, the scheme used for PTS does not have the qArray field). I implemented the GPU-Twig algorithm from scratch on CUDA [10]. I experimented with XMark [85], a commonly used benchmark tool for generating large XML documents. I also experimented with the DBLP [3] database. I checked different twig patterns (some representative patterns are given below). All experiments were run on an 3 GHz Intel S5520SC ShadyCove 5520 12DDR3 6SATA/R 2LAN1000 EATX workstation with an NVIDIA GTX 480 GPU, with two Intel Xeon 6C X5650 processors, and with 24GB of RAM. Each such processor has 6 cores so altogether the workstation has 12 cores. Xeon processors use Hyper-Threading (HT) technologies [5], which allows to run 2 threads at the same time in one core. Usage of HT technologies provides time improvement of up to 30%. I used the actual runtime as the main metric of performance. I measured the time with, and without, the transferring data time to and from the GPU.
5.5.1 Experiments Description

Setting Up. An experiment run has two input files: an XML document, and a text file with query (twig) patterns to run against the given document. An experiment begins with loading the input document into the (extended) XML L-Stream storage by the parser. Then, I parse the queries, and process them against the input document.

I used different twig patterns. I could not use the XMark benchmark queries directly as they rely on a richer language. The XMark benchmark queries have predicates, functions and complex axes (like functions etc.), whereas twig patterns can have only the child or descendant axes. So, I created twig patterns by taking the original benchmark queries and transforming them to twig patterns. Some representative patterns are as follows.

Patterns for the XMark database: (1) "//open_auctions//annotation
  //text//keyword] //listitem[/bold]///emph"; (2) "//item[/mail//emph]  //listitem //parlist//text//bold"; (3) "//item[/mail//emph] //listitem //parlist//text//bold"]; (4) "//annotation[/parlist//text
  //keyword]///bold]// listitem[/bold ]//emph"; (5) "//regions//item[/mail //emph]//parlist//text";


Experiment Description. The document is first loaded to the storage system (the time of loading is not measured, as it is a one time procedure). Every
experiment has the following runs:

1. *Sequential Run* - I process the queries in the queries text file using the sequential TwigStack algorithm [26]. Information regarding start and end times of processing the queries is collected in the result log file.

2. *Parallel CPU Run* - This run is performed with 12 threads. I process the queries in the queries text file. While processing the queries, I run the parallel PTS algorithm [89, 87]. Information regarding start and end times of processing the queries is collected in the result log file.

3. *Parallel GPU Run* - This run is performed using the GPU. I process the queries in the queries text file. While processing the queries, I run the GPU-Twig algorithm as described in Chapter 5. Information regarding start and end times of processing the queries is collected in the result log file.

According to all performed experiments, our Extended Xml L-Stream Storage scheme occupies 4.2 times the space of the original scheme. This is a feasible enlargement in real-life applications, given the performance gains.

I compare the performance of GPU-Twig to TwigStack and to PTS by comparing the runtime of these algorithms in three different ways. In the first way I start the time measurement for the GPU-Twig algorithm before copying the data from the CPU to the global memory of the GPU, and stop after copying the result data from the GPU to the CPU. In the second way I start the time measurement for the GPU-Twig algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement right after finishing the
query processing, but before copying the results data from the global memory of the GPU to the CPU. In the third way I start the time measurement for the GPU-Twig algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement after copying the result data from the GPU to the CPU. The first way of measuring indicates the potential time improvement of the GPU for large documents that cannot fully reside in the global memory of the GPU. The second way of measuring indicates the potential time improvement for documents that can fully reside in the global memory of the GPU. This is an important measurement as in a case that the document can fully reside in the GPU, I have to copy it to the GPU only once and then I can run many queries over this document in a row, by this eliminating the need for copying the document to the GPU per each query. The third way of measuring indicates an improvement that can be gained in the future when the global memory of the GPU will be ”united” with the RAM (CPU memory). Time is measured in milliseconds. Each experiment is characterized by the size of the input XML document. I experimented with documents sized as following: 256MB, 524MB, 750MB, 1GB, 2GB. 2GB maximum size is dictated by the GPU global memory size (whose size is 1.5GB); note that only the relevant streams have to be copied to the global GPU memory, so ordinarily the amount of memory that is copied to the global GPU memory is much smaller than the document size.

In all the graphs: $GPU_{ALL}$ displays the duration for executing the GPU-Twig
algorithm including the time of transferring the data from the CPU to the
GPU and transferring the results from the GPU to the CPU; $GPU_{WITHRES}$
displays the duration for executing the GPU-Twig algorithm with the time
of transferring the results from the GPU to the CPU, but without count-
ing the time of transferring the input streams from the CPU to the GPU;
$GPU_{ALGONLY}$ displays the duration for executing the GPU-Twig algorithm
without the time of transferring the data to and from the GPU; $CPU$ with 12
threads displays the duration for executing the PTS algorithm on the CPU
while running 12 threads in parallel. I do not include the sequential results
in the figures, because of the huge scale differences with the parallel runs
results.

5.5.2 Experiments

Figure 5.5: Results of GPU-Twig on documents with different sizes, for twig
pattern 1

Figure 5.5: Results of GPU-Twig on documents with different sizes, for twig
pattern 1
Figures 5.5, 5.6 and 5.7 show the results of GPU-Twig on XMark documents with different sizes for twig patterns 1, 3 and 5, respectively. I observe that in all the graphs the results are similar. The GPU-Twig run with full memory transferring time (both directions) has speedup with respect to PTS of about 7.6 and 9 for twig patterns 1, 3, and 5, respectively. The GPU-Twig run with result transferring time (from the global memory to the CPU) has speedup with respect to PTS of about 17, 10 and 25 for twig patterns 1, 3, and 5, respectively. The pure GPU-Twig runtime without transferring times has speedup with respect to PTS of about 26, 12 and 43 for twig patterns 1, 3 and 5, respectively. Figure 5.5 presents the results of the sequential run. I can see that the PTS runtime in this case is about 2.5 times better than the sequential runtime. Investigating this phenomenon, I ran experiments on two multi-core architectures. I found that on the ”old”
Figure 5.7: Results of GPU-Twig on documents with different sizes, for twig pattern 5

Figure 5.8: Results of GPU-Twig on document with size 524MB, for different twig patterns
Figure 5.9: Results of GPU-Twig on document with size 1GB, for different twig patterns

Figure 5.10: Results of GPU-Twig on document with size 2GB, for different twig patterns
Figure 5.11: Results of GPU-Twig on the DBLP database document with size 670MB, for different twig patterns

Figure 5.12: Summary of GPU-Twig speedup in comparison to PTS with 12 threads, for different twig patterns
multi-core architecture (on which I experimented with the PTS algorithm), the runtime of PTS with 10 threads is 9.5 times better than the runtime of the sequential algorithm. On the current multi-core architecture, in which I also tested the GPU-Twig algorithm, the runtime of PTS is only 2.5 times better. A large effort was invested to understand why on a more powerful and modern parallel hardware, the results are worse. After trying to use some known parallelization schemes like Intel’s TBB (Threading Building Blocks) [32], and debugging with different profilers, I found that in the new platform, the threads spend most of their time waiting for a new task, whereas the time of processing the actual task is negligible. Unfortunately, I did not manage to find large enough input files for which task processing time would not be negligible, to check my hypothesis. This was mainly due to hardware limitations of the current multi-core architecture (such as RAM size - larger RAM is needed for parsing process of larger XML files). Thus, I did not resolve this problem.

Figures 5.8, 5.9 and 5.10 show the results of GPU-Twig with different twig patterns on documents with sizes 524MB, 1GB and 2GB, respectively. I see that in all the graphs the results are similar. The GPU-Twig run with full memory transferring time (both directions) for twig patterns 1-5 has speedup with respect to PTS of about 8, 6 and 6 on average for document sizes 524MB, 1GB and 2GB, respectively. The GPU-Twig run with result transferring time (from the global memory to the CPU) for twig patterns 1-5 has speedup with respect to PTS of about 23, 13 and 18 on average for
document sizes 524MB, 1GB and 2GB, respectively. The pure GPU-Twig runtime without transferring times for twig patterns 1-5 has speedup with respect to PTS of about 34, 25 and 28, for document sizes 524MB, 1GB and 2GB, respectively. The improvement of the GPU-Twig algorithm for large files is of the same order of magnitude as for smaller files. I see that the speedup in the results for 524MB sized document is similar to the speedup that I obtained for large documents (1GB and 2GB).

**Note:** An overlap exists in the information displayed in Figures 5.5, 5.6, 5.7 and Figures 5.8, 5.9, 5.10, but neither is contained in the other.

I also conducted experiments over the DBLP database. Figure 5.11 shows the results of GPU-Twig with different queries on a DBLP document (670MB). The improvement of GPU-Twig is of the same magnitude as on the XMark documents. To be precise, the speedup of GPU-Twig over the DBLP document in comparison to the sequential TwigStack algorithm is 42 on average (according to the presented results), which is similar to the obtained improvement for some of the queries with GPU-Twig over XMark documents in comparison to the sequential TwigStack algorithm.

DBLP is an example of a document that can fully reside in the global memory of the GPU. Thus the most important result for this document is the GPU-Twig run with result transferring time (from the global memory to the CPU), which has speedup of about 2.4 over PTS for twig patterns 1-3. The pure GPU-Twig runtime without transferring times for twig patterns 1-3 has speedup of about 2.6 over PTS. The reason that the speedup here is smaller
than for XMark documents is as follows. The DBLP database has a very simple structure, and it is difficult to find complex queries for such a simple structure. For documents with a very simple structure and queries, and with very small number of answers, the PTS algorithm has an optimization feature that leads to the unusual speedup of more than 3 times in comparison to its "usual" performance (see end of Section 4.2, and explanation regarding Figures 4.5 and 4.8, in Section 4.4). GPU-Twig does not have such a feature, as it has a vastly different design, so its speedup over PTS is smaller in this unusual case.

Figure 5.12 graphically shows the speedup with full memory transferring times (both directions), with result transferring time, and without transferring times, of the tested queries for XMark documents. The mean speedup across all queries was 26 without memory transferring times (according to the presented results), which was reduced to 16.5 when only results transfer time was included, and reduced to 6.6 when full memory transferring times were included. Thus, on average, running the queries for data that is bigger than the global memory was on average 6.6 times faster than executing the query on a multi-core system using the PTS algorithm, and running them on the data that is already loaded on to the GPU, and transferring the result set back, was on average 16.5 faster.

I conducted experiments for documents with different structures and for different queries. The XMark benchmark, is an example of a document whose nodes are not uniformly spread over the document. This can potentially
cause (parallel) work imbalance. Still, the experimental results do not show an imbalance.
Chapter 6

The GGQ (GPU Graph database Query) Algorithm

This part of the research was published in a GraphQ@EDBT workshop 2015 paper [90].

The GGQ algorithm is a SIMD algorithm. The main advantage of the GGQ algorithm is the ability to divide the matching work to hundreds or even thousands of threads that run in parallel, and that the work of each thread is exactly of the same length. The idea of the basic version of the algorithm is to use the ID of a thread to determine the portion of the data to which a pattern matching attempt will be executed by the particular thread. Then, as the number of bits in a thread ID is bounded, I designed an extension that allows the algorithm to be efficient also in cases when the query tree or the data graph are more complex.
GGQ processes mainly the document parts that are relevant to the input query by processing only edge streams that are relevant for the input query. GGQ does not make use of some of the GPU features (such as shared memory). The main used GPU-specific features are as follows. The first is the potential for massive parallelization. The second is utilizing the GPU’s high bandwidth in transferring data from the global memory to the local memories, which is more efficient than data transfer from main memory to the CPU. The third is usage of the coalesced memory access feature (as explained in Section 2.7). The limiting factor of the algorithm’s performance, i.e., the algorithm’s bottleneck, is mainly the global memory bus bandwidth (i.e., data transfer from the GPU global memory to the MP local memory), as the algorithm is not compute-intensive. The inputs of the algorithm are a labeled graph $G = (V, E)$, a TPQ $Q$, and a set of nodes $V_q$, subset of $V$, containing all data graph nodes which are part of legal possible matches for the root node of $Q$. The algorithm finds all possible matches between $Q$ and $G$ subject to the $V_q$ constraint.

Next, I explain the main idea of the algorithm. For ease of explanation, assume that set $V_q$ has just one node, $v_1$. Each GPU thread has a unique $ID$. For example, the $ID$ of thread $th$ is $thNum$, and in binary $thNum = < b_m, b_{m-1},...,b_0 >$. For any label $lbl$, let $outgNum(lbl)$ be the maximum number of edges labeled $lbl$ connected to a node in the database. So, for all nodes $v$, $v$ has at most $outgNum(lbl)$ outgoing edges with label $lbl$. Thus, I need $\lceil \log_2(outgNum(lbl)) \rceil$ bits to represent $outgNum(lbl)$. For
ease of exposition, I assume that $\text{outgNum}(\text{lbl})$ for any label is a power of 2. The bits of $thNumb$ define which edge has to be chosen at each step of checking for a match against the data graph.

For example, assume that I have just two types of labels, $\text{lblA}$ and $\text{lblB}$, in the graph. $\text{outgNum}(\text{lblA}) = 4$, $\text{outgNum}(\text{lblB}) = 16$. Assume that I have a query pattern $Q$ with 3 edges, the first and third edges are with label $\text{lblA}$, and the second edge is with label $\text{lblB}$. Assume that the maximal thread ID is 255, thus I have 8 bits $< b_7, b_6, ..., b_0 >$ to represent any possible thread ID. Bits $b_0$ and $b_1$ represent the index of all possible data edges with label $\text{lblA}$, for the first edge in the query pattern. Bits $b_2$, $b_3$, $b_4$, and $b_5$ represent the index of all possible data edges with label $\text{lblB}$, for the second edge in the query pattern. And finally, bits $b_6$ and $b_7$ represent the index of all possible data edges with label $\text{lblA}$, for the third edge in the query pattern.

I have also tried to reverse the ordering of enumerating the thread ID bits (i.e., to extract the bits from left to right - from MSB to LSB), so that the MSB bit will correspond to the top of the tree. However, the effectiveness of this will fully depend on the nature of the data tree. For the data I used which induces a flat structure on the data tree, this scheme turned out to have inferior performance.

To gain intuition about the algorithm, I start off with an example.
6.1 An Intuitive Example

Consider query $qTree$ and $dGraph = graph(V, E)$ presented in Figure 6.1.

Based on $dGraph$, I see that each node $v_i \in V$ has no more than $outgNum(created) = 2$ outgoing edges labelled $created$ and no more than $outgNum(knows) = 3$ outgoing edges labelled $knows$, thus I need 1 bit to represent $outgNum(created)$ and 2 bits to represent $outgNum(knows)$, here bit=0 means edge number 1 and bit=1 means edge number 2. Observe that for a node $v$, there are 3 possibilities for having an edge with label $created$: 0, 1, or 2 edges. Yet there is only one bit to represent possible $created$ edge. The ???trick??? is that zero edges are detected by the fact that a member with label $created$ is missing in the edges data list of the relevant node.

In total, I need 4 bits to represent all the possible potential matchings with graph $gData$ (given in Figure 6.1). During the run, I have 16 different...
running threads. For \( ID \) with bits \\
< \( b_3, b_2, b_1, b_0 \) >, bits \( b_0 \) and \( b_1 \) apply to the first edge (between query nodes 0 and 1), bit \( b_2 \) applies to the second edge (between query nodes 1 and 2) and bit \( b_3 \) applies to the last edge (between query nodes 1 and 3). \( V_q \) contains the data node with \( ID = 0 \). Next, I go over all the threads and explain how each operates at runtime.

The thread with \( ID \) 0000, finds that the data node with \( ID \) 0 has 3 outgoing edges labelled \( \text{knows} \), according to the first 2 bits "00" of the thread’s \( ID \), I choose the first edge which leads us to the data node with \( ID \) 1. While checking the data of this node, I find that it does not have label "Lila". So, this partial matching is not part of an answer, and the thread terminates with no match. The same scenario happens for the threads with ID: 0100, 1000, and 1100.

The thread with \( ID \) 0010, chooses the 3’rd outgoing edge (corresponding to the "10" bits) of the data node with \( ID \) 0. Thus, it matches the query node with \( ID \) 1 to the data node with \( ID \) 2. While checking the data of this node, I find that it does not have label "Lila". Thus, the thread 0010 terminates with no match. The same happens for the threads with ID: 0110, 1010, and 1110.

The thread with \( ID \) 0011, chooses the 4’th outgoing edge (corresponding to the "11" bits) of the data node with \( ID \) 0, but such an edge does not exist. Thus, the thread terminates with no match. The same happens for the threads with ID: 0111, 1011, and 1111.
The thread with ID 0001, chooses the 2’nd outgoing edge of the data node with ID 0. Thus, it matches the query node with ID 1 to the data node with ID 3. The data node with ID 3 has label ”Lila”?. Next, the thread chooses the 1’st outgoing edge, with label ”created”, of the data node with ID 3. Thus, it matches the query node with ID 2 with the data node with ID 4. The data node with ID 4 has no label ”Jonathan”?. Thus the thread terminates with no match. The same behavior happens for thread with ID 1001.

The thread with ID 0101, chooses the 2’nd outgoing edge of the data node with ID 0. Thus, it matches the query node with ID 1 to the data node with ID 3. The data node with ID 3 has label ”Lila”?. So, the thread now chooses the 2’nd outgoing edge of the data node with ID 3. Thus, it matches the query node with ID 2 with the data node with ID 2. The data node with ID 2 has label ”Jonathan”?. Now, the thread chooses the 1-st outgoing edge of the data node with ID 3 (corresponding to the leftmost ”bit” with value 0). Thus, it matches the query node with ID 4 with the data node with ID 4. The data node with ID 4 has label ”Nerya”?. At this point the thread has finished to match all the nodes and edges. Thus the thread reports that the currently identified assignment of query nodes to data nodes is an answer to qTree in dGraph, and terminates with a match.

The last thread does not find a match.
6.2 Base Algorithm

I specify how the algorithm operates for query Q, graph G, set V_q, and a thread with ID thNum. For ease of explanation, assume that set V_q has just one node, v_1. In case that V_q has more than one node, I run the same algorithm for each node in V_q separately.

Figure 6.2 presents the base version of the GGQ algorithm. The input to the algorithm are the data graph G, the TPQ Q and the set V_q that contains the matching data node of the TPQ query root node. Line 2 contains the invocation of the CUDA kernel function \textit{gpuGraphQuery} which is processed on the GPU. I.e., the query processing algorithm itself is executed on the GPU.

The \textit{gpuGraphQuery} kernel call finds all the matchings between the TPQ query Q and the data graph G. The code of \textit{gpuGraphQuery} is run in all the threads of the GPU. Conceptually, the actual hardware does not necessarily do that and certain scheduling is usually performed. They execute exactly the same code (i.e., the code of the \textit{gpuGraphQuery} function itself) at the same time ("Single Instruction") over different data in G by different threads ("Multiple Threads"). The more threads available for execution, the better the utilization of the GPU. As the possible number of pattern matchings between Q and G is very large, there is a potential for an enormous number of parallel threads, which promises high utilization of the GPU. According to the GPU GTX 480 architecture, the maximum number of resident threads
Input: 1) Data graph $G$. 2) TPQ query $Q$. 3) $V_q$ set.
Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query $Q$ in data graph $G$.

Method (runs on the CPU):
1. $ansSet = \{\}$
2. Invoke CUDA kernel call for function: $GpuGraphQuery(G, Q, V_q, ansSet)$
3. return $ansSet$

// $GpuGraphQuery$ kernel function (runs on the GPU):
Input: 1) Data graph $G$. 2) TPQ query $Q$.
3) $V_q$ set of nodes in $G$ that match to the root of $Q$.
4) $ansSet$ set of all answers (thread IDs).
5) $idConst$. Has default value of 0. Used for algorithm extensions

Goal: In case that current thread’s ID encodes an answer to query $Q$ in data graph $G$, add it into $ansSet$.

Method:
1. set $thID$ to a system assigned index of the current thread.
2. set $maxID$ to a system value of the maximal thread index.
3. $thNum = (idConst \cdot (maxID + 1) + thID)$
4. express $thNum$ in binary notation as $< b_0, b_1, ..., b_{k-1} >$.
5. set $bitIdx$ to 0. /* represents the currently processed bit in the binary notation of $thNum$ */
6. create $dataNodeArray$ of size $|Q|$ and initialize all its entries to nil. /* the element with index $a_i$ will be data graph node $d_n$, that corresponds to query node with index $a_i$ */
7. $dataNodeArray[1] = v_r$ /* w.l.o.g. the root index of $Q$ is 1 and $v_r$ is some matching data node (according to $V_q$) */
8. foreach edge $e_{cqIdx} = (q_a, q_b)$ in $Q$’s edges
9. $lbl = e_{cqIdx}.getLabel()$ /* According to the graph definition, there are no more than $2^k$ outgoing edges labeled $lbl$ from any node in $G$ */
10. $k = Q.getNumBits(lbl)$ /* These bits corresponds to edge number $cqIdx$ in $Q$ */
11. $currV = dataNodeArray[q_a.idx]$. /* the data graph node to which $q_a$ is mapped */
12. $currE = currV.getEdge(lbl, num)$. /* gets edge number $num$ out of outgoing edges labeled $lbl$ of node $currV$ */
13. if ($currE == nil$) then return
14. $currV = currE.getTargetNode()$. /* find $q_b$ */
15. if (not isMatching($currV, q_b$)) then return
16. $dataNodeArray[q_b.idx] = currV$. /* update the mapping array */
17. $bitIdx = bitIdx + k$. /* prepare $bitIdx$ to read next edge data */
18. end foreach
19. $ansSet.add(thNum)$. /* current thread encodes an answer */

Figure 6.2: The base GGQ algorithm

per MP (multiprocessor) is 1536 (i.e., 1536*15 for all the MPs), while the number of threads that are processed at any moment of time in the MP is
32 (other threads may be waiting for data from the global memory, or just
waiting for their turn to be run). The threads in the GPU are arranged
in blocks where each block can have a maximum of 1536 threads. If the
requested (by the algorithm) number of threads exceeds 1536*15, then the
GPU first handles the 15 first blocks, and then continues to process the next
15 blocks, and so on until all the blocks are processed. Note that maximum
number of threads that can actually run in parallel at any point of time is
480 (32 on each of the 15 MPs). The potential number of pattern matchings
between $Q$ and $G$ is very large, and is usually much larger than the number
of processors in the GPU. Thus the utilization of the GPU is usually very
high (as there are no idle nodes), i.e., the throughput of processing the work
is high in comparison to multi-threaded CPU systems (as much more results
are computed at parallel).

Next, I explain the $gpuGraphQuery$ kernel function (see Figure 6.2). Line 1
computes the thread’s ID, namely $thNum$, according to CUDA’s semantics.
In line 3 I compute the index for which the current thread is responsible.
In the base algorithm, $idConst$ is always 0. Thus, $thNum = thID$. $cqIdx$
that is defined in line 4, indicates the index of the currently processed edge.
Line 6 defines the $bitIdx$ variable. $bitIdx$ points to the bit that is currently
processed in the binary presentation of $thNum$. The $dataNodeArray$ array
that is defined in line 7 holds the data nodes that are matched against query $Q$
by the current thread. I.e., the element with index $a_i$ of the $dataNodeArray$
is the data graph node $d_n$ that is matched to the query node with index $a_i$. 
The size of \textit{dataNodeArray} is the number of nodes in $Q$, i.e., $|Q|$. In line 8, \textit{dataNodeArray}[1] is initialized. This is the data node that is matched to the root node of $Q$. This data node named $v_r$ is taken from set $V_q$ which is one of the parameters of the \textit{gpuGraphQuery} function.

Line 9 the algorithm starts a \textbf{foreach}, that tries to perform a matching between the pattern that is encoded by \textit{thNum} (the index of the current thread) and $G$ according to TPQ $Q$. Note that before starting the algorithm, the edges of $Q$ are sorted in a way that if edge $e_x$ is on the path from $Q$’s root to the source vertex of edge $e_y$, then $e_x$ precedes $e_y$ in the order. And this is the order in which they are processed during the \textbf{foreach}. In lines 10-12, the algorithm finds the edge number \textit{num} that has to be chosen out of the outgoing edges labeled \textit{lbl} of node $currV$ (a value is assigned to $currV$ in line 13). $currV$ is the node that is matched to the $qa$ node, which is the source node of the $e_{cqIdx}$ edge. $currV$ is taken out of the \textit{dataNodeArray} according to the index of the $qa$ node. The ordering of $Q$ edges (described above) guarantees that $currV$ exists. To find \textit{num}, the algorithm first extracts the bits of the binary representation of \textit{thNum} that correspond to the $e_{cqIdx}$ edge. The decimal value that is encoded by these bits is inserted to \textit{num}. In line 14 the algorithm gets the outgoing \textit{lbl} labeled edge number \textit{num} of node $currV$ and assigns it to edge $currE$. If the value of $currE$ is \textit{nil}, it means that such an edge does not exist, thus according to line 15 the algorithm terminates the run, as this thread does not encode a matching pattern in graph $G$. In line 16-17, using edge $currE$, the algorithm finds the data node
that matches to the query node $q_b$ (the target node of edge $e_{cqIdx}$) and inserts it to $currV$, then it checks the matching between the data of the new $currV$ and the data of $q_b$. In case it finds that there is no matching between the data of $currV$ and $q_b$, it terminates the run, as this thread encodes a pattern that does not exist in graph $G$. In lines 18-19 the data of $dataNodeArray$ and $bitIdx$ is updated, as preparation to the next iteration of the `foreach`. If the algorithm finishes successfully the `foreach` loop for all the edges, without returning in lines 15 or 17, it means that the current thread encodes a pattern that exists in the graph and that fully matches $Q$. That is why in line 21, the algorithm inserts $th.Num$ to $ansSet$.

Note that the main feature that is used by the algorithm for speeding up query processing is that of high parallelism. The second feature that is used is the coalesced memory access (as explained in Section ??). By running a profiling tool on GGQ, I found that the execution uses the coalesced memory access feature of the GPU $^1$. This is due to an apparent matching between the structure of the storage and the way the algorithm traverses the data.

There are possible optimizations of the basic scheme. As pointed out by a reader, one can base thread addressing on a simple algorithm that takes into account the maximum number of edges with a particular label emanating from a node and, based on the query and the thread ID, deduce the thread’s search pattern. This will often result in fewer threads.

$^1$This means that when many threads in a warp access consecutive global memory addresses, these memory accesses are grouped into one access.
One can also wonder why not use a scheme in which a few threads are used for each processed node. Then, each thread examines a different edge, and when some edge is found to match, it is marked, and in the next step I launch a new kernel that will check the next node that is the target node of the matching (marked) edge, in the same fashion. However, such a scheme is not utilizing the GPU’s powerful massive thread parallelism. This scheme is appropriate for a CPU, or a multi-core, environment as it uses just a few threads at each kernel run and does not utilize the GPU’s capabilities.

6.3 First Algorithm Extension (Brute Force Looping)

| Input: 1) Data graph $G$. 2) TPQ query $Q$. 3) $V_q$ set. |
| Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query $Q$ in data graph $G$. |

**Method (runs on the CPU):**

1. $ansSet = \{\}$
2. $maxIDbitNum = getBinBitsNum(getMaxID())$
   
   /*getMaxIDbits is a system function*/
3. $maxQBinNum = getBinBitsNum(Q)$
4. for $(i = 0; i < 2^{(maxQBinNum−maxIDbitNum)}; i++)$
5.     Invoke CUDA kernel call for function:
6.     $GpuGraphQuery(G, Q, V_q, ansSet, i)$
7. end for
8. return $ansSet$

Figure 6.3: The first extension of the GGQ algorithm

There can be situations in which the maximal number of bits that may
be required to represent query patterns is larger than the number of bits of maximal thread ID. Possible reason is that the query pattern has many edges. Thus, I extend the algorithm as presented in Figure 6.3. Assume that the maximal thread ID is \( \text{maxID} \) and that I need \( \text{maxID\text{bits}} \) to represent it, that \( \text{maxBin} \) bits are required to represent the query pattern, and that \( \text{maxBin} > \text{maxID\text{bits}} \). In line 4 I start a for loop. The number of iterations is: \( 2^{(\text{maxQBin\text{Num}} - \text{maxID\text{bit\text{Num}}})} \). At each loop iteration (line 5), I run the base algorithm, where each thread in the current iteration will take care of the pattern represented by the following number: \( (i \times (\text{maxID} + 1) + \text{threadID}) \), where \( \text{maxID} \) is the ID of the maximal thread ID, and \( \text{threadID} \) is the system ID of the current thread. This computation can be seen in line 3 of the base algorithm (Figure 6.2). Note that this way, conceptually, I extend the thread’s ID bit representation to the left by placing there the bits corresponding to \( i \) in the current loop iteration.

For example, assume that the maxID of a thread is 1023, i.e. represented by 10 bits, thus \( \text{maxID\text{bits}} = 10 \). The number of edges in the query is 4, and \( \text{outg\text{Num}} \) of each label type is 8, i.e., representing each edge requires 3 bits. In that case \( \text{maxBin} \) is 12. I.e., \( \text{maxBin} = 12 > \text{maxID\text{bits}} = 10 \). The maximal number that can be represented by 12 bits is 4095. Thus, the number of iterations in the for loop is \( 4 \) (\( 2^{(12-10)} = 2^2 \)). When \( i = 0 \), I handle patterns with ID value from 0 to 1023. When \( i = 1 \), I handle patterns with ID value from 1024 to 2047. When \( i = 2 \), I handle patterns with ID value from 2048 to 3071. When \( i = 3 \), I handle patterns with ID value from 3072
This extension is used when there is a very large amount of possible patterns. In this case there is a need to run the kernel multiple times. Thus, the running time of the algorithm can increase drastically.

Often, when a query is posed, the desired answer is whether there exist any matching between the query tree and the data graph. In such cases, it is sufficient to find one matching in order to provide a positive answer. In a slightly modified version of the algorithm, the run is stopped the moment a first match is found. This feature decreases the running time of the algorithm in such cases (more details in Section 6.6).

Sometimes, the desired answer to a query corresponds to only one specific query node and not to all nodes corresponding to the whole set of query nodes. This does not affect the GGQ algorithm as an answer provided by GGQ to a query is an ID of some thread. This ID encodes the whole correspondence and the entire node matching can be extracted from this ID. So, in the case in which just one node is desired, the data node can easily be obtained from the ID. Note that compared to other algorithms, GGQ does not have to store much data regarding the answers as each answer is encoded within an ID.
6.4 Second Algorithm Extension (Multi Phase)

A substantial possible improvement, in case that the number of possible patterns is larger than the maximal thread ID, is a two phase exploration (and, in general, a multi phase exploration using the same principle). Here, I first limit the pattern by removing subtrees (actually edges leading to the roots of subtrees) so as to be left with the original rooted pattern with portions removed so that the remaining new pattern $Q'$ is a "prefix" of the original pattern $Q$. The idea is that I have sufficiently many bits in $\text{maxIDbits}$ to explore the smaller $Q'$ (with no need to use the first extension). A $Q'$ node is called a contact node if it is a node in $Q$ from which an edge leading to a subtree was removed along with the whole subtree. When evaluating $Q'$ I record for each solution the images in the data graph of the contact nodes of $Q'$ which I call a recorded solution vector. Then, I run the second phase in which, for each recorded solution vector, I explore the rest of $Q$ using all the threads I can utilize. In case two phases are not sufficient, I grow $Q'$ to $Q$ in more than 2 phases. Each such phase will produce a collection of recorded solution vectors in which additional $Q$ nodes are assigned values.

The advantage of this two phase (and in general multi phase) scheme is that (a) I employ many threads in the first phase working on a smaller query derived from the original query and obtain all the relevant prefixes, encoded in recorded solution vectors, out of the data graph. (b) In the second phase, for each recorded solution vector, I employ all threads on a relevant portion
of the data graph that can potentially lead to a solution to \( Q \).

For example, consider the query \( Q \) as presented in the left side of Figure 6.4. Suppose that an edge representation requires 4 bits for any label, namely the whole pattern requires 32 bits. Suppose that \( \text{maxID} \) requires 16 bits. So, I am "missing"? 16 bits. I can transform \( Q \) to the Prefix query \( Q' \) with 4 less edges, as presented in the right side of Figure 6.4. This way I can handle \( Q' \) with all threads. Once I evaluate \( Q' \) I obtain recorded solution vectors. Each such vector encodes a partial matching of the full matching, and determines the data contact nodes \( v_a \) and \( v_b \) that are matching to the query contact nodes \( v_2 \) and \( v_7 \). When phase 2 is carried out for each recorded solution vector, each thread will operate on the subtrees rooted at \( v_2 \) and \( v_7 \) where the \( \text{dataNodeArray} \) will be initialized with \( v_a \) in the location corresponding to \( v_2 \) and \( v_b \) in the location corresponding to \( v_7 \). As the subtrees rooted at
the contact nodes have a total of 4 edges, 16 bits will suffice to represent all possible matchings. This means that in phase 2, when considering a particular recorded solution vector, all threads will be employed in checking possible continuations for this recorded solution vector. Thus, the computing power is fully utilized in (the short) phase 1 and later on throughout phase 2.

Note that there is an advantage here over the loop scheme in that for a loop index that corresponds to a non-prefix of the data graph, all GPU threads are activated in vain. Here, the first phase guarantees that the sequence of GPU activations is done for recorded solution vectors that correspond to potentially extendable navigations. The disadvantage is that the recorded solution vectors need be stored so that they are available for the second phase.

Figure 6.5 presents the second extension to the algorithm. The function getPrefixQ (line 2), decides which part of $Q$ is going to be the "prefix" query. It makes the decision based on the number of bits $bLimit$ required to present the maximal thread $ID$, and on the structure of $Q$. Basically, it chooses the "upper" part of the tree (the part with the smallest depth), up to the limit of $bLimit$. I.e., it sums the number of bits that are required to present all the edges of the chosen part, and enlarges the chosen part up to the limit of $bLimit$. Lines 3,4 runs the base algorithm on $prefixQ$, and inserts the answer into prelimAnsSet. Line 5 initialize ansSet, the set of final answers. remainQ that is computed in line 6, is the remainder part of $Q$ after remove $prefixQ$ out of it. Line 7 starts a foreach that
computes the final answers for $Q$ based on the preliminary answers from $\textit{prelimAnsSet}$. The set of answers of the current iteration, $\textit{carrAnsSet}$ is defined in line 8. Lines 9-10, contain the invocation of the CUDA kernel function $\textit{gpuGraphQueryExt}$ which is processed on the GPU, and is slightly different from $\textit{gpuGraphQuery}$ (as defined in Figure 6.2). In line 11 I add the answers that were found in the current iteration to the final answers set (namely $\textit{ansSet}$).

Next I describe $\textit{gpuGraphQueryExt}$. This function has slight differences from the base algorithm GPU function, $\textit{gpuGraphQuery}$. Thus, I describe just these differences. The first difference is in line 8, in which the $\textit{thNum}$ is defined. $\textit{thNum}$ is equal to the system value of the $ID$ of the current thread. The second difference is in line 8, in which the $\textit{initNodesArray}$ function initializes $\textit{dataNodeArray}$. The function extracts from $\textit{baseAns}$ the matchings between nodes in $Q$ and nodes in $G$ that were found during the first phase, and assigns the found data nodes into the appropriate places in $\textit{dataNodeArray}$. Except for the described two changes, the function operates exactly as the base $\textit{gpuGraphQuery}$ function.

The number of edges that can be represented by one phase is $n$ such that

$$\sum_{edge=1}^{n} \textit{outgNum}(\textit{lbl}(e)) \leq 2^{\textit{bitsNum}(\textit{maxThreadID})}$$

where $\textit{bitsNum}(\textit{maxThreadID})$ is the number of bits that are used by the GPU to represent the maximal thread ID. For example, assume that a GPU thread ID is represented with 32 bits. Assume that for each edge $e$, on average, there are 16 potential $\textit{outgNum}(\textit{lbl}(e))$ from each node. Thus, on average, I need 4 bits to repre-
sent each edge of the query. Based on the above, each phase allows us to represent $32/4 = 8$ edges on average. Having 2 phases in the multi-phase extension described above allows us to represent fairly large TPQs with about 16 edges. The multi-phase extension can be easily extended to more than 2 phases. Based on this analysis, if I extend the multi-phase extension to 3 phases, I can represent a TPQ with 24 edges, which is a very large query. It is important to note that without the multi phase extension, experiments involving very large queries give very poor results that are worse than Grmolin’s performance on these queries.

Figure 6.6 presents the algorithm for braking the query into phases. Variable $currPhase$ (line 1), holds the current phase number. Variable $currBitsNum$ (line 2), holds the number of bits that are already occupied by edges that belong to $currPhase$. Lines 3-5 defines a BFS order pass over all the edges (first edges in depth one, then edges in depth 2 etc.). If there are no more bits for a new edge in $currPhase$ (line 5), then I increment the current phase (line 7) and reset the $currBitsNum$ variable (line 8). Lastly, I assign the current phase to the currently treated edge (line 11), and update the number of occupied bits in this phase (line 10).

6.5 Some Ideas for Additional Extensions

The GGQ algorithm can be easily used as a base algorithm for graph structured queries. For that I need to change the storage scheme. I need to have
the information regarding the incoming edges of a node and not only the outgoing edges of a node. Assume that I changed the storage scheme as described above, then, the steps for answering a graph structured query \( GQ \) using the GGQ algorithm are as follows. First, ignoring edge directions, find a spanning tree \( TGQ \) of \( GQ \) (using any known spanning tree algorithms). Then, run the GGQ algorithm over \( TGQ \). This is possible as I have both incoming and outgoing edge information. As a last step, filter the answers found by GGQ, and delete all answers that do not conform with the \( GQ \) graph structure. In Chapter 6.6, I present experimental results for processing a graph structured query using the simplified scheme described above. In future research one can improve this scheme by choosing appropriate spanning trees, and by optimizing the filtering step.

TPQ queries with ancestor-descendant (AD) edges are not common over graph databases. Yet, the GGQ algorithm can be extended to handle TPQ queries with an AD edge. I explain it for a case of a single AD edge. The main idea is to brake the query into two parts. Suppose that in TPQ \( Q \) the AD edge is (c-d). Then, one query part is the part above the (c-d) edge, and the second one is the part below the (c-d) edge. I start with running GGQ for the first part of \( Q \), the one above edge (c-d) on each node in set \( V_q \). As a result, I obtain a set of answers and extract from them all the relevant 'c' corresponding nodes. Then, I run (on the CPU or on the GPU) any known algorithm to find all the 'd' nodes that are reachable from these extracted 'c' nodes. The last step is to run GGQ on each one of the found 'd' nodes.
for the second part of \(Q\), the one below edge (c-d). By this I complete the query processing and obtain the full set of answers to the query. I can expand this idea also for a case when there is more than one AD edge a query. The generalization of this idea is left to future research.

Another possible extension is to find a better edges encoding scheme. I can encode by using a more optimal scheme, by this reducing the number of bits that are required for the encoding. I will show it by example \(^2\). Assume that I have 4 edges: \(e_1\), \(e_2\), \(e_3\) and \(e_4\). Where \(e_1\) has 3, \(e_2\) has 2, \(e_3\) has 5 and \(e_4\) has 6 edge possibilities \(^3\). I parse the threads ID in the following way: 

\[
3 \times (2 \times (5 \times (6 \times \text{node} + e_4) + e_3) + e_2) + e_1.
\]

That is, to know which possibility of \(e_1\) to choose, I divide the number by 3, and the remainder of the division defines which possibility of \(e_1\) to choose. Next, I divide the quotient by 2, and the remainder of the division defines which possibility of \(e_2\) to choose. Next, I divide the new quotient by 5, and the remainder of the division defines which possibility of \(e_3\) to choose. Next, I divide the new quotient by 6, and the remainder of the division defines which possibility of \(e_4\) to choose. The last quotient, defines which node out of \(V_q\) set I need to choose (where \(V_q\) contains all data graph nodes which are part of legal possible matches for the root node of the query tree).

While GGQ seems brute-force, note that (a) its simplicity is very powerful in a SIMD setting as it induces a very simple, short and uniform computation

\(^2\)Credit for the idea is due to an unknown referee.

\(^3\)The possible number of \(e_1\) edges is 0,1,2 for a total of 3 possibilities.
on each thread, and (b) the algorithm can be adjusted to a particular data and query characteristic by applying the scheme of multi-phase extension not only when I “run out” of address space but also on smaller files, and (c) it appears that utilization is low as many of the threads will quickly become inactive, however - as each execution of a thread is extremely short, this does not affect efficiency as it is compensated by simplicity and low overhead.
Input: 1) Data graph $G$. 2) TPQ query $Q$. 3) $V_q$ set.
Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query $Q$ in data graph $G$.
Method (runs on the CPU):
1. $prelimAnsSet = \{\}$
2. $prefixQ = getPrefixQ(Q)$ /*$getPrefixQ$ returns the "prefix" of the query $Q$*/
3. Invoke CUDA kernel call for function:
4. $GpuGraphQuery(G,prefixQ,V_q,prelimAnsSet,0)$
5. $ansSet = \{\}$
6. $remainQ = getRemainQ(Q,prefixQ)$ /*$getRemainQ$ returns $Q \setminus prefixQ$*/
7. foreach $ans$ in $prelimAnsSet$
8. $currAnsSet = \{\}$
9. Invoke CUDA kernel call for function:
10. $GpuGraphQueryExt(G,remainQ,currAnsSet,ans)$
11. $ansSet = ansSet \cup currAnsSet$
12. end foreach
13. return $ansSet$

// $GpuGraphQueryExt$ kernel function (runs on the GPU, just the differences from $GpuGraphQuery$ presented):
Input: 1) Data graph $G$. 2) forest $remainQ$.
3) $ansSet$ set of all answers (thread IDs).
4) $baseAns$ is the ID that encodes the matching between $prefixQ$ and $G$
Goal: In case that current thread’s ID encodes an answer to query $remainQ$ based on the matching presented by $baseAns$ in data graph $G$, add it into $ansSet$.
Method:
... 
3. $thNum = thID$
...
8. $initNodesArray(dataNodeArray,baseAns)$ /* $initNodesArray$ extracts $baseAns$, and fill all the nodes that already matched in dataNodeArray by answering $prefixQ$ in the first phase */
...

Figure 6.5: The second extension of the GGQ algorithm
Input: 1) query edges ($qEdges$). 2) $maxQdepth$, the max depth of $Q$
3) $maxBitsNum$, the number of bits required to represent
maximal thread ID

Goal: to set the field $phaseNum$ of each query edge

Method (runs on the CPU):
1. $currPhase = 1$
2. $currBitsSum = 0$
3. for $depth$ from 1 to $maxQdepth$
4. foreach $edge$ in $qEdges$
5. if $edge.depth == depth$
6. if $currBitsSum + edge.bitsNum > maxBitsNum$
7. $currPhase ++$
8. $currBitsSum = 0$
9. end if
10. $currBitsSum += edge.bitsNum$
11. $edge.phaseNum = currPhase$
12. end if
13. end foreach
14. end for

Figure 6.6: Query phase ordering algorithm
6.6 GGQ Experimental Evaluation

I compared GGQ to Gremlin [84] in terms of runtime (to completion). Gremlin is the only query processor that I was able to find that uses the native graph approach and that supports XPath-style queries over graph documents. Using Gremlin’s query language, one can easily express TPQs. I was not aware of any parallel graph query processor to which I could compare our results. I used the GLS storage scheme to store the data (see Chapter 6.7). I implemented the GGQ algorithm from scratch on CUDA [10]. I experimented with GRR [21], a benchmark tool for randomly generating random RDF documents. I also experimented with the Geospecies data document [2], and a representative data document example of the Census database [12]. I checked different TPQ query patterns (some representative patterns are given below). All experiments were run on the same hardware as described in Section 5.5. I used the actual runtime as the main metric of performance. I measured the time with, and without, the data transferring time to and from the GPU.

Note that Gremlin’s output is an unsorted set of single nodes which are the answer nodes to the input query, whereas GGQ’s output is an unsorted set of full matchings between the input query pattern and the input data graph. GGQ return full matchings (and not just answer nodes), because by returning the ID of the threads, the GGQ algorithm returns the full matching. That is as the thread ID encodes some full matching between the input query pattern
and the data graph. Thus I do not have to make any additional work to get the full matchings, so I return the full matching "at the same price".

6.6.1 Experiments Description

Setting Up. An experiment run has two input files: an RDF document, and a text file with query (TPQ) patterns to run against the given document. An experiment begins with loading the input document into the GLS storage system. Then, I parse the queries, and process them against the input document.

I used different TPQ patterns. The patterns I used have different lengths and of different tree structures. Recall that the syntax of a query is as follows. I indicate an edge by using '{ }'. The name that is written inside '{ }' denotes an edge label. I indicate a branch by using '[]'. The expression written inside '[]' defines a whole branch (subtree). '*' means that any label can match. A name of a label is defined by a string that does not contain spaces, or by a string inside "". If we use "" to define label’s name, than space can be part of the label.

Some representative queries are presented below. For example, in query 1, 'file://C:/GRR/UniBenchmark/Data/#Lecturer/Lecturer13172' is the name of a label is it does not contain spaces. For example, in query 5, "River Bend" is the name of a label as it contains spaces.

Query 1: / file://C:/GRR/UniBenchmark/Data/#Lecturer/Lecturer13172 { ub:worksFor } / * { ub:subOrganization } / * { ub:name } / University91
Figure 6.7: Outline of Representative TPQ examples

The structure of Query 1 is presented in Figure 6.7, TPQ (1).

**Query 2:** file:///C:/GRR/UniBenchmark/Data/#Professor/Professor28942
\{ ub:worksFor \} / * \{ ub:subOrganization \} / * \{ ub:name \} / University65

The structure of Query 2 is presented in Figure 6.7, TPQ (1).

**Query 3:** file:///C:/GRR/Data/#Lecturer/Lecturer13172 \{ ub:worksFor \} / * \{ ub:subOrganization \} / * \{ ub:name \} / University138

The structure of Query 3 is presented in Figure 6.7, TPQ (1).

**Query 4:** http://www.rdfabout.com/geo/etowah_county/wills_valley \{ dcterms:isPartOf \} / * \{ dcterms:hasPart \} / * \{ geo:lat \} / 34.071247

The structure of Query 4 is presented in Figure 6.7, TPQ (1).

**Query 5:** http://www.rdfabout.com/geo/hale_county/prairie_eden_newbern
Query 5: The structure of Query 5 is presented in Figure 6.7, TPQ (3).

Query 6: The structure of Query 6 is presented in Figure 6.7, TPQ (2).

Query 7: The structure of Query 7 is presented in Figure 6.7, TPQ (1).

Query 8: The structure of Query 8 is presented in Figure 6.7, TPQ (3).

Query 9: The structure of Query 9 is presented in Figure 6.7, TPQ (4).

Query 10: The structure of Query 10 is presented in Figure 6.7, TPQ (4).
The structure of Query 10 is presented in Figure 6.7, TPQ (5).

**Experiment Description.** The document is first loaded to the GLS storage system (the time of loading is not measured, as it is a one time procedure).

Every experiment has the following runs:

1. **Gremlin Run** - I process the queries in the queries text file using Gremlin [84]. Information regarding the runtime of the query is collected in the result log file. To run a query on Gremlin, I use Gremlin through a bash (shell) dialog. After opening the dialog, as first step, I upload the document to Gremlin’s database. Next, I take a timestamp, then I run the query, and finally I take one more timestamp. Using the taken timestamps, I calculate the time of running the query on Gremlin. Note, that I translate my queries to the ”Pipes” query language (Gremlin’s engine) [84].

2. **GPU Run** - This run is performed using the GPU. I process the queries in the queries text file. The queries are processed by the GGQ algorithm as described in this Chapter. Information regarding start and end times of processing the queries is collected in the result log file.

I compare the performance of GGQ to Gremlin by comparing the runtime of these algorithms in three different ways. In the first way I start the time measurement for the GGQ algorithm before copying the data from the CPU.
to the global memory of the GPU, and stop after copying the result data from the GPU to the CPU (namely, \textit{GPU-full}). In the second way I start the time measurement for the GGQ algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement right after finishing the query processing, but before copying the results data from the global memory of the GPU to the CPU (namely, \textit{GPU-alg}). In the third way I start the time measurement for the GGQ algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement after copying the result data from the GPU to the CPU (namely, \textit{GPU-ans}). \textit{GPU-full} reflects the potential time improvement of the GPU for large documents that cannot fully reside in the global memory of the GPU. \textit{GPU-ans} reflects the potential time improvement for documents that can fully reside in the global memory of the GPU. This is an important measurement as in a case that the document can fully reside in the GPU, I have to copy it to the GPU only once and then I can run many queries over this document in a row, by this eliminating the need for copying the document to the GPU per each query. \textit{GPU-\text{alg}} is appropriate for GPUs in which the global memory and RAM are merged, i.e., in more recent processors such as NVIDIA’s PASCAL GPU family. Time is measured in milliseconds.

Note that I do not count the time of preparing and finding the root node(s) of the query (according to the algorithm definitions and requirements, the
root node of the query is an input to the algorithm)\(^4\). I find the index of the root node of the query, by simply performing linear pass over the array with all the nodes of the data Graph. Then I pass the found index of the root node to the GPU.

Each experiment is characterized by the size of the input RDF document. I experimented with document sizes as follows: 40MB, 125MB, 180MB, 600MB. I did not use larger files, as the GRR benchmark tool was unable to create larger files. Also, the dominant factor that influences the complexity of GGQ is the size of the query and not the size of the RDF database document. That is why I did not try to find larger RDF database documents. Note that only the relevant edge streams have to be copied to the global GPU memory, so ordinarily the amount of data that is copied to the global GPU memory is much smaller than the document size.

### 6.6.2 Experiments

The performance of GGQ in comparison to Gremlin is so much superior that it is meaningless to present the results graphically. This is why I present all the results in tables.

Figures 6.1 and 6.2 show the results of GGQ on GRR documents with sizes 125MB and 600MB respectively for different TPQ queries. The queries that were used for Figure 6.1 are Query 1 and Query 2. The queries that

\(^4\)This will necessitate an efficient data structure such as a search tree or a hash table, all well-known in the literature.
Table 6.1: Results of GGQ on a document with size 125MB (GRR dataset), for path queries with 4 nodes (query 1 and query 2). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Path Q1</th>
<th>Path Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>114</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>0.74</td>
<td>0.73</td>
<td>154</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>0.09</td>
<td>0.08</td>
<td>1267</td>
</tr>
<tr>
<td><strong>GPU – alg</strong></td>
<td>0.085</td>
<td>0.075</td>
<td>1425</td>
</tr>
</tbody>
</table>

Table 6.2: Results of GGQ on a document with size 600MB (GRR dataset), for a path query with 4 nodes (query 3) and a tree query with 5 nodes (query 6). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Path Q1</th>
<th>Tree Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>76</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>3.09</td>
<td>3.11</td>
<td>25</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>0.08</td>
<td>0.10</td>
<td>950</td>
</tr>
<tr>
<td><strong>GPU – alg</strong></td>
<td>0.075</td>
<td>0.078</td>
<td>1085</td>
</tr>
</tbody>
</table>

Table 6.3: Results of GGQ on a document with size 40MB (Census dataset), for a path query with 4 nodes (query 4) and a tree query with 5 nodes (query 5). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Path Q1</th>
<th>Tree Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>119</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>0.4</td>
<td>0.98</td>
<td>297</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>0.09</td>
<td>0.67</td>
<td>1322</td>
</tr>
<tr>
<td><strong>GPU – alg</strong></td>
<td>0.085</td>
<td>0.6</td>
<td>1400</td>
</tr>
</tbody>
</table>

184
Table 6.4: Results of GGQ on a document with size 180MB (Geospecies dataset), for a path query with 4 nodes (query 7) and a tree query with 6 nodes (query 8). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Path Q1</th>
<th>Tree Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>81</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>0.75</td>
<td>0.78</td>
<td>108 96</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>0.09</td>
<td>0.12</td>
<td>900 625</td>
</tr>
<tr>
<td><strong>GPU – alg</strong></td>
<td>0.08</td>
<td>0.11</td>
<td>1012 682</td>
</tr>
</tbody>
</table>

Table 6.5: Results of GGQ on a document with size 180MB (Geospecies dataset), for two different tree queries with 11 nodes (query 9 and query 10). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Tree Q1</th>
<th>Tree Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>187</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>48.66</td>
<td>3.26</td>
<td>3.84 50.6</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>48</td>
<td>2.6</td>
<td>3.9 63.5</td>
</tr>
</tbody>
</table>

Table 6.6: Results of GGQ on small documents (with size 4.5MB and 1.6MB from Geospecies dataset), for two different tree queries with 11 nodes (query 9 and query 10). The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

<table>
<thead>
<tr>
<th></th>
<th>Tree Q1</th>
<th>Tree Q2</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gremlin</strong></td>
<td>118</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td><strong>GPU – full</strong></td>
<td>7.45</td>
<td>1.12</td>
<td>16 110</td>
</tr>
<tr>
<td><strong>GPU – ans</strong></td>
<td>7.4</td>
<td>1.09</td>
<td>16 113</td>
</tr>
</tbody>
</table>
Table 6.7: Results of a *GPU – full* GGQ run on documents with different sizes (from Geospecies dataset), for two different tree queries with 11 nodes (query 9 and query 10), while stopping right after finding the first answer

<table>
<thead>
<tr>
<th>Size</th>
<th>Tree Q1</th>
<th>Tree Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>180 MB</td>
<td>30.8</td>
<td>2.81</td>
</tr>
<tr>
<td>&lt; 5 MB</td>
<td>3.95</td>
<td>0.98</td>
</tr>
</tbody>
</table>

were used for Figure 6.2 are Query 3 and Query 6. The GGQ run with full memory transferring time (both directions) has speedup of about 154 and 115 for a document with size 125MB and of 25 and 23 for a document with size 600MB, for Q1 and Q2 respectively with respect to Gremlin. The GGQ run with result transferring time (from the global memory to the CPU) has speedup of 1267 and 1050 for a document with size 125MB and of 950 and 720 for a document with size 600MB, for Q1 and Q2 respectively with respect to Gremlin. The GGQ pure run (*GPU – alg* without transferring times) has speedup of 1425 and 1200 for a document with size 125MB and of 1086 and 923 for a document with size 600MB, for Q1 and Q2 respectively, with respect to Gremlin.

Figure 6.3 shows the results of GGQ on a Census sample document with size 40MB for different TPQ queries. The queries that were used for Figure 6.3 are Query 4 and Query 5. The GGQ run with full memory transferring time (both directions) has speedup of 297 and of 88, for Q1 and Q2 respectively with respect to Gremlin. The GGQ run with result transferring time (from the global memory to the CPU) has speedup of 1322 for Q1 and of 131 for Q2 with respect to Gremlin. The GGQ pure run (without transferring times)
has speedup of 1400 for Q1 and of 147 for Q2 with respect to Gremlin. The speedup in case of $GPU - full$ is much larger than in previous cases. This is because the data document is very small, so it takes much less time to copy it. Observe that the speedup of Q2 on the 40MB document in all cases is "just" about 130, and not more than 500 as in previous cases. The reason is that due to the TPQ pattern structure the order of the traversal over the data document does not allow the memory coalesced access feature to take effect.

Figure 6.4 shows the results of GGQ on a Geospecies data document with size 180MB for different TPQ queries. The queries that were used for Figure 6.4 are Query 7 and Query 8. I observe that in all the queries the results are similar. The GGQ run with full memory transferring time (both directions) has speedup of 108 for Q1 and of 96 for Q2 with respect to Gremlin. The GGQ run with result transferring time (from the global memory to the CPU) has speedup of 900 for Q1 and of 625 for Q2 with respect to Gremlin. The GGQ pure run (without transferring times) has speedup of 1012 for Q1 and of 682 for Q2 with respect to Gremlin.

Figure 6.5 shows the results of GGQ on a Geospecies data document with size 180MB for two different long TPQ queries (having 11 nodes). These queries are Query 9 and Query 10. The GGQ run with full memory transferring time (both directions) has speedup of 3.8 for Q1 and of 50.6 for Q2 with respect to Gremlin. The GGQ run with result transferring time (from the global memory to the CPU) has speedup of 3.9 for Q1 and of 63.5 for Q2.
with respect to Gremlin. The GGQ pure run (without transferring times) has speedup of 3.9 for Q1 and of 64 for Q2 with respect to Gremlin.

Figure 6.6 shows the results of GGQ on Geospecies data documents with small size (4.5MB, 1.6MB) for the same two long TPQ queries (having 11 nodes). These queries are Query 9 and Query 10. The GGQ run with full memory transferring time (both directions) has speedup of 16 for Q1 and of 83 for Q2 with respect to Gremlin. The GGQ run with result transferring time (from the global memory to the CPU) has speedup of 16 for Q1 and of 113 for Q2 with respect to Gremlin.

I experimented with graph structured queries. I have transformed the two long queries into directed acyclic graph queries. In the first long query (query 9), I made both of the "relatedMatch" edges point to the same node. Whereas in the second long query (query 10) I made both of the ?primaryTopic? edges point to the same node. The running time of the new queries (including all three steps) is practically the same as when running the original queries. The reason for this is that the query graph, in terms of graph sizes, is very small. Thus, the time it takes to find a spanning tree over such a small graph is negligible. The same holds for the last phase. As there are just few answers, the time of filtering the relevant answers out of the whole set of answers is negligible as well. Thus, I can see that even with this unoptimized scheme, the durations of the first and last steps are negligible.

I created a slightly modified version of the algorithm in which the run is stopped the moment a first match is found. Figure 6.7 shows the results
of the modified GGQ on Geospecies data documents with size 180MB and with small sizes (4.5MB, 1.6MB) for the same two long TPQ queries (having 11 nodes), queries 9 and 10, that were used in Figure 6.5, while accounting for the full memory transferring time (both directions). The results indicate that this feature decreases the running time of the algorithm by 1.5-2 times. I can see that the results of almost all the experiments show improvement of more than 500 times. Further, the high performance behavior of the algorithm is consistent.
This section describes the new graph lists storage scheme that I use, GLS.
GLS is based on the XML L-Stream representation scheme [26, 88]. The motivation of using this storage scheme is that it allows a very compact and easy to use storage, while faithfully preserving the graph structure of the data. The main goal in designing this scheme is that the traversal over the data graph will be as fast as possible. That is, the time of navigating from some node to another node via some edge will be minimal and constant. In the design, I can detect each outgoing edge from any node in constant time, and also traverse to the source or target nodes of each edge in constant time. So, our design fulfills the above goal.

A graph lists storage database is a set of data graphs. A data graph is a directed graph \( G = (V, E) \), where (1) \( V \) is a finite set of labeled nodes; (2) \( E \subseteq V \times V \) is finite set of labeled directed edges in which each pair \((v, v')\) denotes an edge from \( v \) to \( v' \).

In the GLS scheme there are two lists that hold the nodes, \textit{NodesList} and \textit{LeafNodesList}. There is also a lists manager that holds all the edges in the graph, \textit{EdgeListsManager}.

\textit{EdgeListsManager} consists of lists of edges, one for each label. Let \( lbl \) denote an edge label. Associated with each distinct \( lbl \) there is a list \( Lst_{lbl} \). The list is an ordered sequence of all the database edges having the \( lbl \) label. The edges in a \( Lst_{lbl} \) list are arranged according to the order in which the edges with label \( lbl \) appear in the input RDF file. That is, the edge with label \( lbl \) that is parsed first will appear first in \( Lst_{lbl} \), the edge with label \( lbl \) that is parsed second will appear second in \( Lst_{lbl} \), etc. Each edge \( e \) in
some edge list $Lst_{lbl}$ holds the following fields: (1) Edge ID, (2) Edge name, (3) The index of the source node in $NodesList$, (4) Target node data: name of the matching nodes list ($LeafNodesList$ or $NodesList$) and the index of the target node in the matching node list.

$NodesList$ holds all the nodes in the graph which are not leaves. Each node $v$ in $NodesList$ holds: (1) Node ID; (2) Node name (the name is unique); (3) Edges data list. For each edge label of an outgoing edge of $v$ the edges data list holds: (i) Edge Name, (ii) Location of the first edge in the $Edge Name$ labeled edges list, Number of outgoing edges of $v$ labeled $lbl$ (all edges going out of a node with the same label appear consecutively in the edge lists).

For example, given the following information: node $v$ has 3 outgoing edges labeled $Name$, and the first $Name$ labeled edge of $v$ is edge number 5 in the $Lst_{Name}$ list, the edges data list of $v$ contains: Name, 5, 3. $LeafNodesList$ holds all the leaf nodes of the graph, where a leaf node is a node that has no outgoing edges. Each node $v$ in $LeafNodesList$ holds: (1) Node ID; (2) Node name.

An example GLS database is presented in Figure 6.8. At the top left corner we see the RDF document. At the right bottom corner we see the graphical representation of the given document. The three tables that are presented in the figure, are built according to the description that is given above.
Chapter 7

Conclusions and Future Work

I presented two new algorithms for operating in a multi-core system. The Parallel Path Stack algorithm (PPS) and the Parallel Twig Stack algorithm (PTS) are novel and efficient algorithms for matching XML query twig patterns in a parallel multi-threaded computing platform while using a L-Stream representation scheme. PPS and PTS employ a special search technique for limiting L-Stream processing to specific subtrees. For extremely large files the EXSS scheme, a novel storage model, is used.

I performed extensive experimentation with PPS and PTS. I compared PPS and PTS to the standard (sequential) PathStack and TwigStack algorithms in terms of runtime (to completion). I examined their performance for varying numbers of threads. Experimental results indicate that using PPS and PTS significantly reduces the running time of queries in comparison with the PathStack/TwigStack algorithm (up to 54 times faster for a DBLP based bench-
mark and 18 times faster for an XMARK based benchmark). There are many published variations and improvements on PathStack and TwigStack. However, as I demonstrate, any of these variations is "pluggable" to my scheme, taken more or less as a black box. So, if an improved version of TwigStack is used, the performance of my algorithm will also improve. Therefore, for testing, I used the original TwigStack (the most accessible version). For validation, I also implemented one of the known improvements over TwigStack, TSGeneric+, and ran tests using it. Any other variation would have done as well.

For PTS, for a finite number of threads, the total I/O and runtime complexity of all the threads run by PTS is optimal among all algorithms that read the entire input. Also, for all parallel algorithms that execute threads so that the entire input is read, PTS has an optimal parallel runtime complexity. It will be interesting to consider a dynamic partitioning scheme that takes into account the number of available cores, the size of the memory and that can also learn, from experience, the optimal settings to the system’s operational parameters.

The next part of my research is the work on the GPU-Twig algorithm. GPU-Twig is a novel efficient algorithm for matching XML twig patterns. I use an extension of the stream representation scheme [26], in a parallel multi-threaded computing platform, using a GPU as a CPU co-processor. GPU-Twig employs techniques that allow it to run hundreds of threads in parallel. I made extensive experimentation with GPU-Twig. I compared GPU-Twig
to the standard sequential TwigStack algorithm, and the parallel multi-core PTS algorithm in terms of runtime. I checked performance for varying document sizes and for different queries. Experimental results indicate that using GPU-Twig significantly reduces the runtime of queries compared to other algorithms. Depending on the scenario (memory resident, including only reporting, including both uploading and reporting), the improvement was times 6.6 - times 35.

A promising future direction is to extend the GPU-Twig algorithm to deal with larger, terabyte sized, files. It is also interesting to examine certain features and their associated trade-offs, and determine whether I can further optimize the algorithm. Another research direction is to design different extensions to the basic algorithm. An example of such an extension is a modification in the definition of the Twig Pattern Matching problem. According to this modification, a match of twig pattern $Q$ in database $D$ is a mapping from nodes in $Q$ to distinct nodes in $D$ such that satisfies the original match requirements (see Chapter 2.2). The requirement for distinct nodes necessitates changes to the algorithm. Other extensions are possible, e.g., limit the distinctness requirement to siblings, maintain query sibling order in results and more. Whereas some such extension may be implemented via filtering the results, algorithmic changes may be more efficient.

The last part of my research deals with the GGQ algorithm, a novel efficient algorithm for processing TPQ queries on graph documents. I use a new storage scheme, GLS, in a parallel multi-threaded computing platform, using
a GPU as a CPU co-processor. GGQ employs techniques that allow it to run hundreds of threads in parallel.

I conducted extensive experimentation with GGQ. I compared, in terms of runtime, GGQ to Gremlin [84], that supports XPath-style queries over graph documents. I checked performance for varying document sizes and for different queries. Experimental results indicate that using GGQ significantly reduces the runtime of queries in comparison to Gremlin. Depending on the scenario (memory resident, including only reporting, including both uploading and reporting), the improvement was times 4 to more than times 1000. As part of future research, it will be interesting to construct a sophisticated query generator and to run GGQ on a much larger set of queries.

Another future research direction is to extend GGQ to handle queries that are in the form of a directed graph. For that I need to change the storage scheme. I need to have the information regarding the incoming edges of a node and not only the outgoing edges of a node. Assume that I changed the storage scheme as described above, then, the steps for answering a graph structured query \( GQ \) using the GGQ algorithm are as follows. First, ignoring edge directions, find a spanning tree \( TGQ \) of \( GQ \) (using any known spanning tree algorithms). Then, run the GGQ algorithm over \( TGQ \). This is possible as I have both incoming and outgoing edge information. The basic idea for this extension is to first build a spanning tree out of the query graph. Then, to run the above algorithm on each tree in the forest. As a last step, filter the answers found by GGQ, and delete all answers that do not conform with
the $GQ$ graph structure (namely, a case when the same query node is mapped to different data graph nodes).

Another possible future direction is to examine certain features of GGQ and their associated trade-offs, and determine whether I can further optimize the algorithm. For example, to extend the GGQ algorithm to support descendant type edges (and not only child type edges). Finally, it will be interesting to compare GPU-Twig and GGQ on XML data.
Bibliography


2002.

The lorel query language for semistructured data. Int. J. on Digital

Query co-processing on commodity processors. In VLDB, 2006.


Parallelgdb: a parallel graph database based on cache specialization. 
In IDEAS, 2011.


[86] H. Schoning and J. Wasch. Tamino - An Internet Database System. In


The two algorithms use auxiliary tables that are generated during the data preparation phase and build auxiliary tables for each node to identify the answers to the data query.

The fourth algorithm is **GGQ** (GPU Graph database Query). Its goal is to answer queries that are in the form of a tree on a database and it does so through the use of graphics database (GPU) in CPU. It is here that I see the advantage of algorithms GGQ, GPU-Twig, among others, that use the GPU's high data and task processing power when performing heavy tasks. The CPU is used for resource management and input-output operations. The novelty of the GGQ algorithm is that it uses constant identifiers (IDs) to determine the mapping between the nodes and the query tree in order to build the database. Since the number of possibilities that can be represented by identifiers is limited, there are methods in the graph that allow us to expand these possibilities.

I performed extensive and deep benchmark tests using known and specially created benchmarks. I compared the **GGQ** and **PTS** algorithms with the standard ones **PathStack** and **TwigStack**. I ran experiments with **GGQ** on DBLP and XMark benchmarks. I performed experiments with **GGQ** on DBLP and XMark benchmarks. The results showed that under different conditions, all four algorithms reduce the execution time of the queries. The results showed that under different conditions, all four algorithms reduce the execution time of the queries. The results showed that under different conditions, all four algorithms reduce the execution time of the queries.
באופיין החשיבה והתכנות בשתי הארכיטקטורות השונותocht את הארכיטקטורות
אלאויה, אלאורימים המועדים אלאריקטוטורות

אלאה בודא קליה לא ייעל בשנייה.

now enough time to calculate and code in both architectures and as a result, an algorithm is usually inefficient in the other.

maintaining the graphs or databases is more efficient than maintaining the frequently asked questions. However, smaller databases, such as those of XML documents, are maintained in their "natural" form. When a database is large, (and this is different from table representation) data is stored in graphs for database. In such cases, the tree path traversing and the corresponding traversing the database can be parallelized using a GPU.

I developed and implemented 4 parallel algorithms. The first two algorithms are Parallel Path Stack (PPS) and Parallel Twig Stack (PTS). PPS- and PTS- are "branch template" algorithms for parallel traversing XML documents in parallel with multiple threads (threads).

The algorithms PPS- and PTS- are based on PathStack and TwigStack [26], where PPS- and PTS- are used in the PathStack and TwigStack respectively. The algorithms are tested on real-life databases, such as tree queries. The results show that using the improved algorithms significantly improves the performance of the basic algorithms.

The third algorithm is GPU_Twig. This algorithm is designed to match between queries of "branch template" and which is intended to query XML documents of large (and this is different from table representation) data is stored in graphs for database. In such cases, the tree path traversing and the corresponding traversing the database can be parallelized using a GPU.

The central idea of this algorithm is to copy parts of the relevant to the query and perform the query on the GPU memory and finally copy the results to the CPU.

The algorithm GPU_Twig uses the power of the GPU and CPU to manage resources, and I/O operations. The algorithm traverses the tree (for input query) and the relevant algorithm traverses all the data, during which there are two steps. In the first step, the algorithm traverses all the data, and in the second step, it traverses the relevant data.
XML

A summary of information presented in the document about XML. Naturally, the language can be used to represent a document using XML, that is, XML is a language that can be used to represent data models based on popular queries for XML documents, and it also allows for searching and selecting certain nodes during the search. Actually, a query tool uses patterns (tree patterns) for matching between a pattern tree, where their shape is "twig patterns" (a set of trees). These twig patterns can be seen as a database of all document instances containing all occurrences of "twig pattern" displayed in XML, which is used as a basic function to answer queries on XML documents.

Technion - Computer Science Department - Ph.D. Thesis PHD-2016-03 - 2016

A summary of information presented in the document about XML. Naturally, the language can be used to represent a document using XML, that is, XML is a language that can be used to represent data models based on popular queries for XML documents, and it also allows for searching and selecting certain nodes during the search. Actually, a query tool uses patterns (tree patterns) for matching between a pattern tree, where their shape is "twig patterns" (a set of trees). These twig patterns can be seen as a database of all document instances containing all occurrences of "twig pattern" displayed in XML, which is used as a basic function to answer queries on XML documents.

Technion - Computer Science Department - Ph.D. Thesis PHD-2016-03 - 2016
המחושב בכנותו. בבריגות הלREFIX את הערכות העמוקה למונחים של פרופ' עודד שמואלי יחסי עם המניךスタート. ערcuador שפוף לממדים שהנحياית המסהרה ארוורכ ישראל העמקה. היא ה ACA, על כל הדרך, הוא יאי לא אחר עם סבלנות אין קץ. במחקר, אני חייב לו הרבה יותר ממה שהוא יודע. נינוחת למדידה. אני חיהびי על ההרבה יזר מהו שואו וודע.
בבריגות נ Sakura لديון א获得感, לו מוקדש המחקר, על האהבה, התמיכה והבין בין לארוך כל הדרך. אראה כי לציידי בכלי שבל ברד, וوعد אתינו בריצוף במלא הפreements והצל.
כמו כן בריגות לחהות מ לגרי חומניי הסמוי ב כהחיית ציידה, הלפידים של שנות לי את המוטיבציה ללהבש בבעוד הקשה.
הערכות וחזרתי לשכנות על העורה הפיגננסית הנידנה שעינויג על במלכי butto.
מֵסְמֵכִי גָּדוֹלִים או מְסָדִי נוֹמִים מוֹנְסָטִיסְרוֹפִּים

**הייבר על מחקר**

לְשָׁמוֹנִי חַלוֹק של הָדְרִישָׁה לַקְבוֹלָה החוֹרָה דּוֹכְרָו

לְפֶלְּלוֹסְפִּיתָה

לִילָה שְּגִיָּדְרָמְן

הוֹנֶשׁ לְסַנְטָה הַטֶּכְנִי–מֶכְוָן טֶכְנוֹלוֹגִי לִילְשָׁרְאָל

 Ebay

2015
מ侃בול של שאילתות ואחריות במנועי חיפוש תחומיי
מסמךי XML ודולים ואימהי חיפושי מבוססי גרפים
גדולים

לילנה שיגידרמן