Proc. Sec. Int. Workshop on Accelerating Data Management Systems (ADMS 2011), Seattle, WA, Sept. 2011

Efficient XML Path Filtering Using GPUs

Roger Moussalli
UC Riverside
Riverside, CA, 92507
rmous@cs.ucr.edu

Robert Halstead
UC Riverside
Riverside, CA, 92507
rhalstea@cs.ucr.edu

Mariam Salloum
UC Riverside
Riverside, CA, 92507
msalloum@cs.ucr.edu

Walid Najjar
UC Riverside
Riverside, CA, 92507
najjar@cs.ucr.edu

Vassilis J. Tsotras
UC Riverside
Riverside, CA, 92507
tsotras@cs.ucr.edu

ABSTRACT

Publish-subscribe (pub-sub) systems present the state of the art in information dissemination to multiple users.Current XML-based pub-sub systems provide users with considerable flexibility allowing the formulation of complex queries on the content as well as the structure of the streaming messages. Messages that contain one or more matches for a given user profile (query) are forwarded to the user. Recently various approaches focused on accelerating XML path query filtering using dedicated hardware architectures, like FPGAs. Despite their very high throughput, FPGAs require extensive update time while their physical resource availability is also limited. In this paper we exploit the parallelism found in XPath filtering systems using GPUs instead, which are favorable platforms due to the massive parallelism found in their hardware architecture, alongside the flexibility and programmability of software. By utilizing properties of the GPU memory hierarchy we can match thousands of user profiles at high throughput, requiring minimal update time. Efficient common prefix optimizations are also applied to the query set. An extensive experimental evaluation shows an average speedup of 10x (up to 2.5 orders of magnitude) versus the state of the art software approaches.

1. INTRODUCTION

Increased demand for timely and accurate event-notification systems has led to the wide adoption of Publish/Subscribe Systems (or simply pub-sub). A pub-sub system is an asynchronous event-based dissemination system which consists of three components: *publishers*, who feed a stream of messages into the system, *subscribers*, who post their interests (also called *profiles*), [1] and an infrastructure for matching subscriber interests with published messages and delivering *matched messages* to the interested subscriber.

Pub-sub systems have enabled notification services for users interested in receiving news updates, stock prices, weather updates, etc; examples include *google.news.com*, *pipes.yahoo.com*, and *www.ticket-master.com*. Such systems have greatly evolved over time, adding further challenges and opportunities in their design and implementation. The wide adoption of the *eXtensible Markup Language* (XML) as the standard format for data exchange has led to the current generation of XML-enabled pub-sub systems. Here messages are encoded as XML documents and profiles are expressed using XML query languages, such as XPath[2] [28]. Profiles can now describe requests not only on the message values but also on the structure of the messages.

The core of the Pub-Sub system is the *filtering* algorithm, which supports the complex query matching of thousands of user profiles against a high volume of published messages. Essentially, for each message received in the Pub-Sub system, the filtering algorithm determines the set of user profiles that have one or more matches in the message. Many software approaches have been presented to solve the XML filtering problem [1, 7, 12, 15]. These memory-bound solutions, however, suffer from the Von Neumann bottleneck and are unable to handle large volumes of input streams.

Our previous work [18, 19, 20] has focused on the acceleration of the filtering process through dedicated, customized, and parallel hardware on Field Programmable Gates Arrays (FPGAs). By exploiting parallelism, FPGAs provided filtering at wire-speed, where thousands of hardware query matching engines process, in parallel, streams of XML documents as soon as received from publishers. Nonetheless, FPGAs suffer from extensive profile update time, where updating a single user query would require re-'compiling' the hardware logic of all the system, a process requiring up to several hours. Moreover, circuits on FPGAs are limited by the amount of available resources. For instance, using the current generation of FPGAs, we were able to fit up to 8 thousands queries on a single FPGA.

In this paper we show how the parallelism evident in XML path filtering can be exploited by a GPU-based XML filtering engine. GPUs are favorable platforms due to the massive parallelism found in their hardware architecture, alongside the flexibility and programmability of software. On one hand, updating queries is a fast process requiring few sec-

[1]Here the terms "profile" and "query" are used interchangeably.

[2]In the rest we use '/' and '//' as shorthands to denote the */child:: axis* and */descendant-or-self:: axis*, respectively.

onds; on the other hand, the number of matching engines is virtually unlimited, as a single kernel can be executed as many times as desired.

The contributions of this paper are:

- A generalized and modified version of the filtering algorithm, as applied to GPUs.
- A common prefix optimization approach applied to the list of user queries, with the goal of reducing the filtering time on large query sets.
- An extensive performance evaluation of XML filtering on GPUs versus the leading software implementations.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 provides an in depth description of the proposed GPU framework targeted for XML query filtering, while Section 4 details the common prefix optimization we apply and evaluates the related approaches and parameters. Section 5 presents an experimental evaluation of the GPU-based filtering as compared to the state of the art software counterparts. Finally conclusions appear in Section 6.

2. RELATED WORK

As traditional platforms are increasingly hitting limitations when processing high volumes of streaming data, researchers are investigating alternative platforms for database applications. Recent work has focused on the adoption of Field Programmable Gate Arrays (FPGAs) for data stream processing [21, 24, 22, 26]. The Glacier component library is presented in [21] which includes logic circuits of common operators such as selection, aggregation, and grouping for stream processing. [26] investigated the speedup of the frequent item problem using FPGAs, while in [24] they utilize FPGAs for complex event detection which uses regular expressions to represent events. Predicate-based filtering on FPGAs was investigated by [22] where user profiles are expressed as a conjunctive set of boolean filters. While FPGAs have shown to provide advantages over traditional architectures, they are limited by small resources and expensive programming time. These two limitations have led researches to examine GPUs, since they offer flexibility of reprogramming.

GPUs have evolved to the point where many real-world applications are easily implemented and run significantly faster than on multi-core systems, thus, a large number of recent work has investigated GPUs for the acceleration of database applications[11, 13, 10, 3, 14, 16]. [13] the authors utilized GPUs to accelerate relational joins, while in [10] GPUs are utilized for computing Fourier transforms. In [14] a CPU-GPU architecture is presented to accelerate tree-search, which was shown to have low latency and support online bulk updates to the tree. Recently, [3] proposed the utilization of GPUs to speed-up indexing by offloading list intersection and index compression operations to the GPU. [16] proposed a similarity join algorithm designed to exploit the parallelism and high data throughput on GPUs. For additional details on the architecture and programming model of GPUs, see Appendix A [3].

[3]We are making use of the NVIDIA CUDA model and terminology, as detailed in Appendix A.

Software Approaches for XML Filtering: Several software-based approaches have been proposed and can be broadly classified into three categories: (1) FSM-based, (2) Sequence-based, and (3) other. FSM-based approaches use a single or multiple machines to represent the user profiles. YFilter [7] built upon the work of XFilter[2], and proposed a Non-Deterministic Finite Automata (NFA) representation of user profiles (path expressions) which combines all profiles into a single machine, thus reducing the number of states needed to represent the set of user profiles. [12] proposed a lazy Deterministic Finite Automation (DFA) which has a constant throughput with respect to the size of the query workload, however, lazy-DFA may suffer from state explosion depending on the number of nodes and level of recursion in the XML document, and the maximum depth of the XPath expressions.

Sequence-based approaches as in [15, 23] transform the XML document and user profiles into sequences and employ subsequence matching to determine which profiles have a match in the XML sequence.

Several other approaches have been proposed [5, 9]. XTrie [5] uses a trie-based data structure to index common substrings of XPath profiles, but it only supports the /child:: axis. AFilter [4] exploits both prefix and suffix commonalities in the set of XPath profiles. More recently, [9] proposed two stack-based stream-querying (and filtering) algorithms, LQ and EQ, which are based on lazy strategy and eager strategy, respectively.

FPGA Approaches for XML Processing: Previous works [6, 8, 17] that have used FPGAs for processing XML documents have mainly dealt with the problem of parsing and validation of XML documents. An XML parsing method which achieves a processing rate of two bytes per clock cycle is presented in [8]. This approach is only able to handle a document with a depth of at most 5, and assumes the skeleton of the XML is reconfigured and stored in a content-addressable memory. These approaches, however, only deal with XML parsing and do not address XPath matching.

The work in [17] proposed the use of a mixed hardware/ software architecture to solve simple XPath queries having only parent-child axis.Our previous work [18] was the first to propose a pure-hardware solution to the XML filtering problem. Improvements of more than one order of magnitude were reported when compared to software. However, this method is unable to handle recursion in XML documents or wildcards '*' in XPath profiles.

In [19] we presented a new approach for matching complex path profiles that supports the /child:: axis and /descendant-or-self:: axis [4], wildcard ('*') node tests and accounts for recursive elements in the XML document. In [20] we consider complex twig matching on FPGAs (i.e. the query profiles can be modeled as complex trees). Both approaches, proved very efficient for matching path and twig profiles, however, they are static in the sense that query profiles must be compiled and offloaded to the FPGA beforehand. If queries are added or queries are modified, the query compilation and FPGA synthesis step must be performed. FPGA synthesis, which includes place and route is a time-consuming process and cannot be done in a online fashion. Furthermore, compared to CPUs and GPUs, FPGAs have a small memory

[4]In the rest of the paper we shall use '/' and '//' as shorthand to denote the /child:: axis and /descendant-or-self:: axis, respectively.

thus limiting the number of query profiles that can be processed at a given time. Thus, in this paper, we propose to use GPUs for the XML filtering problem, which offer the flexibility of fast recompilation while exploiting the parallelism found in the application.

3. PATH FILTERING FRAMEWORK

In this section, we go over the details of the path filtering algorithm as introduced in [19], however applied here to GPUs.

3.1 Algorithm Details

Path queries expressed in the XPath language consist of several nodes, with each pair sharing a parent/child or ancestor/descendant relation. A path of length J is said to have matched if a sequence of nodes in the XML document sharing the same relations and tags in the query has occurred; this is only true if the sub-path of length *J-1* has matched.

Due to the tree nature of XML documents, we make use of a stack-based approach for filtering, where each path is matched using one stack. A single pass is required over the XML document, such that the top of the stack is updated at every *open(tag)* (push event) and *close(tag)* (pop event) of the document. The stack width is determined by the path length, and every entry in the stack holds a *boolean* value, indicating, at a given node of the document tree, the match state of the sub-path of length the column number respective to that entry. A'1' in the last column of the stack indicates that the totality of the path has been detected in the document.

Figure 1(a) shows the tree representation of a sample XML document as it is processed. Figures (b) - (g) show several (two-node) queries and the top of the stack filtering mechanisms as it is being updated upon an *open 'b'* event. Using Figure 1, we visit some properties of path queries as applied to the stacks:

- **path root nodes:** as these do not depend on a previous node, a '1' is pushed onto their respective column of the stack (the first column) upon encountering a node with similar tag as part of the XML document (left column of Figure 1(b), vs. no effect in 1(d)).
- **parent-child relations:** these result in a *diagonally upwards* propagation of a '1'. In Figure 1(c), a '1' previously stored in the root column propagates upwards to the right column upon a push, due to an *open 'b'* event. Note that an open 'b' event does not suffice to store a '1' in a 'b' column (see of Figure 1(d)).
- **ancestor-descendant relations:** these result in a *vertically upwards* propagation of a '1' in columns to which query nodes followed by '//' are mapped. Once a '1' is pushed onto the column based on the two previously listed properties, and as long as it has not been popped, the '1' is always pushed onto higher following tops of stack as the document is processed (see Figure 1(f)).
- **path leaf nodes:** A '1' stored in the column to which a query leaf node is mapped, implies a successfully matched query. This occurs through a '1' propagating upwards from the root column, through all middle columns, to the leaf. Figures 1(c), (e) and (g) exhibit matched query states.
- **wildcard nodes:** these imply a level of freedom where any document node is valid for a propagating '1'. See Figure 1(e).

Paths of any depth can be mapped to stacks, where each two columns are related using the above properties.

More examples and algorithm details can be found in [19].

3.2 Levels of Parallelism

Since an XML-enabled pub-sub system involves multiple profiles processed over the same document data stream, parallel architectures are suitable for accelerating its filtering performance. Using our proposed stack-based approach, two levels of parallelism are extracted, namely:

- **Inter-query parallelism** - where all queries (stacks) can be processed in a parallel fashion.
- **Intra-query parallelism** - where updating the state of all nodes within a query (top of stack) can be achieved in parallel.

In the remainder of this section we focus on the details of mapping the previously described filtering algorithm onto GPUs, while utilizing the inherent levels of parallelism.

3.3 Parallel Path Filtering on GPUs

With the filtering algorithm applied to an FPGA-based accelerator [19], the XML documents are directly streamed to the accelerator, and no external memories are required. Stacks are implemented using custom circuitry, where several optimizations are applied to reduce the width of stacks. Such a setup and optimizations are not directly applicable to GPUs, where the XML document has to be stored in the GPU memory, as it is being processed by software implementations of the stack and its properties.

3.3.1 GPU Streaming Processor Kernel

Every Streaming Processor on the GPU is associated with the task of updating the top of a single stack column (at a time), to which a query node is mapped. For every XML document tag opened, the operations performed by every SP on the top of the stack at a given column are as detailed in Procedure 1 GPU Kernel (omitting boundary cases for simplicity purposes).

Where:

- A '1' propagates **diagonally upwards** if the previous column holds a '1' at $current\_level - 1$, and:
  - the column tag is a wildcard.
  - or the column tag is identical to the XML document tag opened.
- A '1' propagates **vertically upwards** if the column holds a '1' at $current\_level - 1$, and the tag mapped to the column is followed by '//' in the path query (is ancestor).
- The *Column ID* is computed at run time using CUDA constructs (as $block\_number \times block\_size \times thread\_ID$). When computation on the GPU is complete, the CPU side selectively extracts final query match states from the global $match\_state$ array.

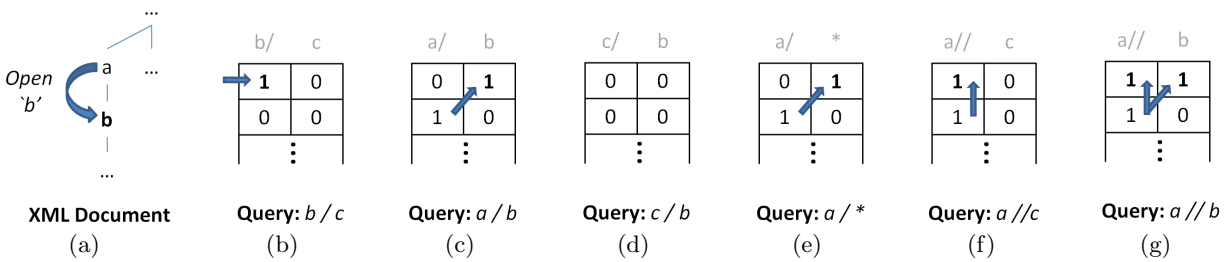

Figure 1: Sample XML document event (*Open 'b'*) shown in (a) alongside corresponding query stack updates at the respective top of the stacks (b) - (g). Query nodes and relations are portrayed in gray above the column they respectively map to.

Procedure 1 GPU Kernel

```
1  current_level ← 0
2  matched ← 0
3  for all XML document events do
4    if pop event then
5      current_level - -
6    else
7      current_level ++
8      if '1' propagates diagonally upwards OR vertically
       upwards then
9        stack[current_column][current_level] ← 1
10       matched = 1
11     end if
12   end if
13 end for
14 if current column is a leaf column then
15   match_state[column_ID] = matched
16 end if
17 return
```

3.3.2 Streaming Processor Personalities

As noted earlier, every instance of the GPU kernel requires knowledge of the query node mapped to the stack column it processes. The CPU parses all queries and sends to the GPU device memory a pre-processed form of all the queries as an array, where each entry represents a query node, and corresponds to a single kernel instance. Figure 2(b) represents a generic view of a query node representation, namely a Streaming Processor (SP) *personality*, on which the kernel instance functionality depends.

A personality consists of a single bit to indicate whether the node is a query leaf, one bit to indicate the relationship (parent, ancestor) between the node and the following one in the query, and a 7-bit representation of the tag ID. The previous column index entry refers to the index of the previous stack column in the block; 5-9 bits are required since GPU blocks can hold 32 to 512 kernel instances. For the moment, it is assumed that query nodes are mapped to contiguous stack columns. However, this assumption will no longer hold starting Section 4.

3.3.3 Efficient Use of the GPU Memory Hierarchy

As described earlier, the filtering algorithm only depends on the events and tags of XML documents, rather than content. In order to minimize transfer from CPU to GPU and utilized memory space, the CPU would compress the XML document events into an optimized representation that is

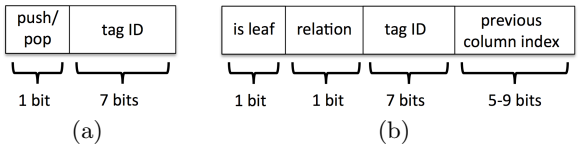

Figure 2: GPU Input format: (a) shows the storage format of every XML document event as streamed to the GPU, whereas (b) depicts the format of SP personalities, representing a query path node and its stack mapping.

then transferred to the GPU. Each compressed entry is 8-bit wide as shown in Figure 2(a), with one reserved bit indicating the event type (push/pop), and the remaining seven bits representing the corresponding tag ID. Every XML document node is translated into such an entry.

As the pre-processed XML is read by all SPs, it is then transferred to the global device memory of the GPU. Since the XML document is read-only, small documents could fit into the cached constant memory. However, our experiments show that minimal speedup was achieved with such small documents mapped to the constant cache. We thus only utilize the global memory to map XML streams of all sizes.

SP personalities are also transferred to the global memory of the GPU. However, since every SP will read its personality once, this is done initially, and the personality is stored in the SP registers.

Every group of SPs forms a Streaming Multiprocessor (SM) on the GPUs. SPs within an SM communicate through a small low-latency shared memory, exclusive to every SM. Stacks are stored in this fast shared memory of every SM because columns of a single stack are continuously updated, processed each by a single SP, and interdependent. The maximum stack depth is set at compile time to allocate the required amount of shared memory.

With the completion of processing on the GPU, the match state of every query is written once to the global memory and streamed back to the CPU.

4. COMMON PREFIX OPTIMIZATION

In this section we go over an evaluation of common prefix optimizations applied to the query set, with the goal of reducing the GPU processing time. With every SP processing one column at a time, reducing the number of columns across all queries will linearly affect the execution time on large query sets.

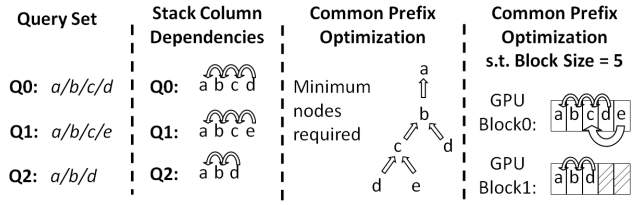

Figure 3: Sample query set shown initially, followed by the stack column dependencies for each query stack independently; these dependencies determine the propagating path of a '1' . A tree representation of all the queries as grouped by common prefixes is then portrayed. Finally, the query nodes are mapped to GPU blocks with a restriction on the block size.

4.1 Motivation

Figure 3 depicts a sample query set. These are queries that will be filtered on the GPU. The stack column dependencies are shown for each query stack separately; these dependencies determine the propagating path (through columns) of a '1' within each stack. One observation is that Q0 and Q1 share the prefix *a/b/c/*, and thus the first three columns of their respective stacks behave identically. On the other hand, Q2 shares the prefix *a/b/* with both Q0 and Q1. We can now build a tree representing all stacks merged at the common prefix nodes, such that tree nodes represent stack columns and query nodes (see Figure 3). This tree will consist of the minimum number of stack columns required to match all queries. In Figure 3 we see that queries Q0, Q1and Q2 require at least 6 stack columns for filtering, in contrast to the initial 11 when not merged.

Note that using our proposed approach, suffixes cannot be shared by queries since a '1' propagating into a column reflects the match state specific to the sub-path formed by the initial remaining query nodes.

A query node followed by '/' is different from a node followed by '//'; similarly wildcards are treated as separate nodes.

4.2 Implementations

When executing on the GPU, kernels are grouped into *blocks* that run on a single SM, even if the the number of threads within a block exceeds the number of Streaming Processors. Shared memories can only be accessed by threads within a single block. Using our approach, stack columns are stored in the shared memory of SMs. A CUDA limitation imposes that blocks hold 32 to 512 threads, a parameter passed at run-time; moreover all blocks are of the same size. With this limitation in mind, the tree, representing the minimum number of nodes (i.e. stack columns required), will be split if the number of tree nodes is larger than the block size, hence replicating common tree nodes across blocks, as described below.

For the sake of this example, we show in Figure 3 the mapping of Q0, Q1 and Q2 into blocks of size 5. Block 0 is able to hold both Q0 and Q1, whereas Q2 is solely mapped to block 1. As all blocks are of fixed size, the remaining two slots in block 1 are empty and will not compute any useful data. Q0, Q1 and Q2 require 10 nodes (2 blocks) instead of the minimal 6.

The fixed-sized block limitation implies that some of the minimal tree nodes will be mapped to more than a single block when the block is smaller than the tree; for instance, *a/b/* was computed in both blocks.

Moreover, the mapping of queries into fixed-size blocks is not unique. For instance, block 0 holding Q0 and Q2, and block 1 holding Q1, would result in one empty node (in block 1).

We look into the problem of mapping query nodes into blocks, while minimizing the number of required blocks. We take the minimal tree as a reference, even though this minimum cannot be achieved, as the assumed query set size is much larger than the maximum block size (tens of thousands of queries vs. block size at most 512).

The first approach we employ executes the following steps:

1. Sort all queries (by tag ID)
2. Instantiate a new GPU block
   (a) Pop the query at the top of the sorted list
   (b) Find the block offset of the last common tag with the previously inserted query (if any)
   (c) Append the remaining query nodes to the block, linking the first suffix node to the node at the index as found in (b)
   (d) Repeat (a)-(c) until the popped query doesn't fit in the block
3. Repeat step 2 on the remaining queries

Nodes are linked using the 'previous column index' portion of the representation described in Figure 2(b). Stack columns of a single query are no longer contiguous in the shared memory, as columns are now shared across tags.

The sorting step will insure that queries popped linearly will share prefix nodes.

We also looked into a variant of that approach, where, instead of linearly popping queries from the sorted list, we greedily pick the query sharing the longest of common prefix with its neighbor, place it in the block, and fill the block with its neighboring queries. This approach runs much slower than the initial one (few seconds vs. tens of seconds), while providing minimal improvement. Therefore, for the remainder of this discussion, we focus on the initial placement technique.

One artifact of placing queries into blocks is fragmentation. Here, blocks exhibit empty slots where queries wouldn't fit. Using the above approaches however, for the test queries in Section 4.3, the average wasted nodes per block was a little over a single node. Therefore, we can deduce that the effect of fragmentation is minimal.

4.3 Evaluation

In this section, we evaluate the common prefix optimization on distinct queries generated using the YFilter query generator [7], based on the following dtd's:

- DBLP, representing bibliographic information and tends to have more bushy trees [25].
- Treebank, representing tagged English sentences and tends to have deep recursive subtrees [25].

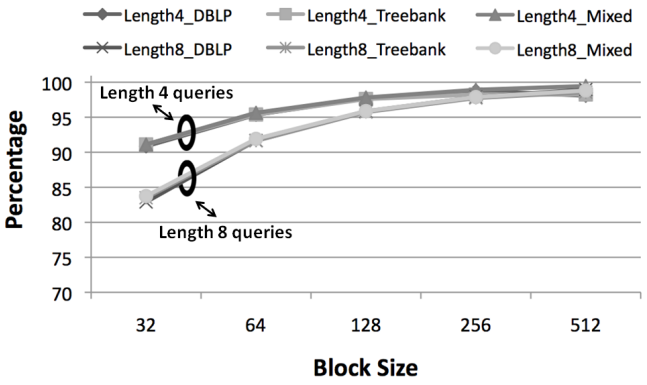

Figure 4: Percentage of reduced nodes vs. minimal tree, resulting from the common prefix optimization, with varying block sizes, on 32,000 queries. The mixed queries are generated from a set of four dtd's.

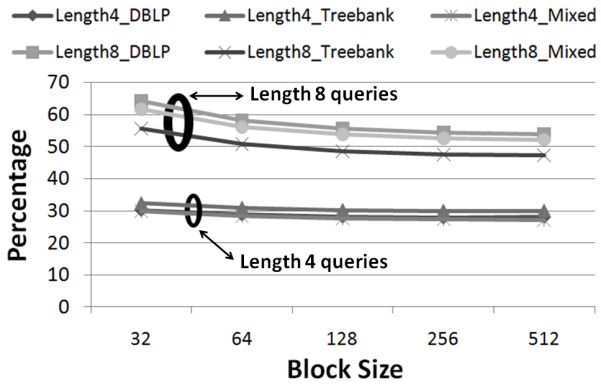

Figure 5: Percentage of resulting remaining nodes when applying the common prefix optimization, versus the original respective query sets (of size 32,000 queries each). The shown percentages includes empty GPU block nodes due to fragmentation.

- SwissProt, a curated protein sequence database providing a high level of annotations, a minimal level of redundancy and high level of integration with other databases [25].
- XMark, as part of the XML Benchmark Project [27].

We show in Figure 4 the percentage of reduced nodes compared to the minimal tree, resulting from the common prefix optimization, with varying block sizes. We perform our study on 32,000 queries of length 4 and length 8, while doubling the block size; queries are of class DBLP, Treebank and Mixed queries, the latter consisting of unrelated queries as generated by four dtd's (DBLP, Treebank, SwissProt, XMark). The purpose of the Mixed query set is to study the effect of the common prefix optimization on an un-biased set of queries exhibiting fewer overall commonalities.

As expected, larger block sizes provide more improvement, as there are fewer replicated nodes across blocks. Blocks of size 256 and 512 nodes result in an almost minimum number of nodes based on the minimum tree. On the other hand, length 4 and length 8 queries exhibit similar respective behavior, such that length 8 queries are always one step (block size) behind in terms of improvement; that is due to the fact that length 8 queries are likely to result in double the amount of query nodes.

Figure 5 shows the overall resulting remaining nodes from applying the common prefix optimization, as percentage of the original query set, with varying block sizes. The remaining nodes are nodes to be computed on the GPU, including the empty block nodes resulting from fragmentation. For all used query sets, length 4 queries result in the most reduction (mean of 71%). On the other hand, length 8 queries result in an average of 45% reduction, less than length 4 queries, due to the longer suffixes. This reduction will almost linearly affect the execution time, as discussed in the upcoming Section.

5. PERFORMANCE EVALUATION

For the remainder of this section, the performance of the GPU-based approaches is measured on an NVIDIA TESLA C1060 GPU (total of 30 SMs comprising of 8 SPs each). The CPU-based approaches were run on a quad core 2.33GHz Intel Xeon machine with 8GB of RAM running Linux Red Hat 2.6.

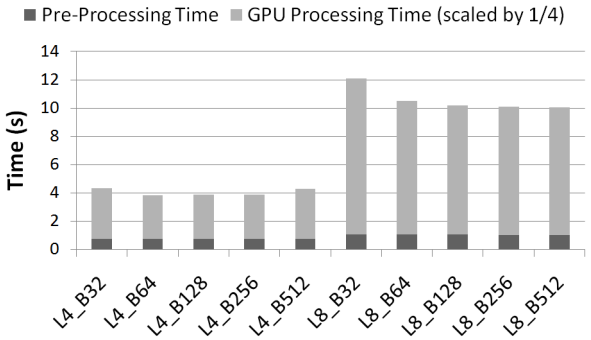

Figure 6: Total execution time filtering 32K queries over a 50MB DBLP XML document, highlighting pre-processing time on the CPU (with the common prefix optimization applied) , and GPU processing time (scaled by 1/4). The query set consists of length (L) 4 and 8 queries, while varying block sizes (B).

5.1 Effect of the GPU Block Size

When applying the common prefix optimization, increased block sizes result in a further reduced query set, as seen earlier. However, due to the contention on the available computing resources and the limited amount of available shared memory per SM, larger block sizes can negatively affect performance when executing on the GPU.

We show in Figure 6 the total execution time required to filter 32K queries over a 50MB DBLP XML document, while highlighting pre-processing time on the CPU (with the common prefix optimization applied) , and the corresponding GPU processing time (scaled by 1/4 for presentation purposes). The query set consists of length 4 (L4) and length 8 (L8) queries, while varying the GPU block size (B32 . . . B512).

The pre-processing time depends solely on the query set,

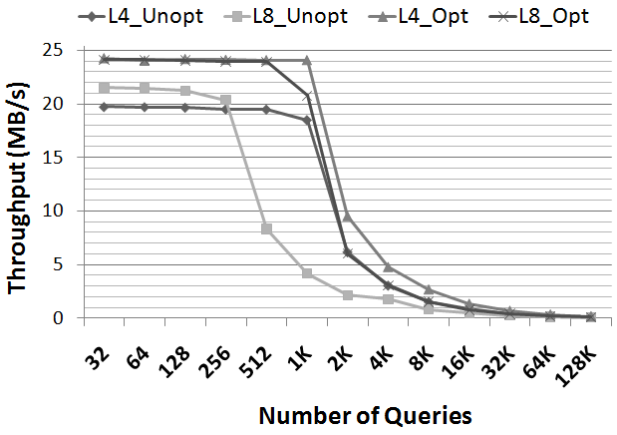

Figure 7: GPU throughput (MB/s) of filtering over a 50MB DBLP XML document, while doubling the number of queries. Data is shown for length 4 (L4) and length 8 (L8) queries, with the common prefix optimization ON (Opt) and OFF (Unopt).

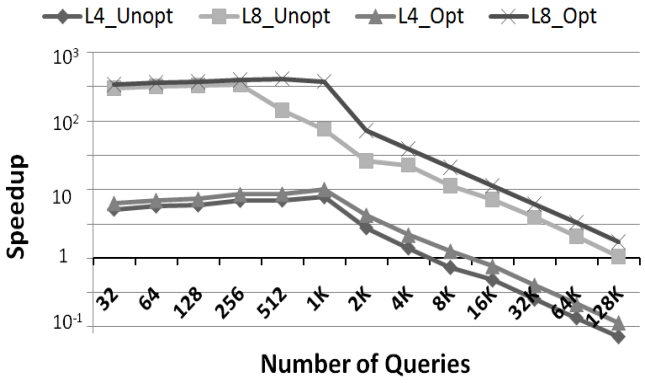

Figure 8: GPU speedup over YFilter when filtering over a 50MB DBLP XML document, as the number of queries is doubled. Data is shown for length 4 (L4) and length 8 (L8) queries, with the common prefix optimization ON (Opt) and OFF (Unopt). Slowdown is depicted for speedup values less than '1'.

and requires an average of 750ms for queries of length 4, versus 1,040ms for length 8 queries, less than 4% on average of the overall execution time.

On one hand, while resource contention is less considerable, a block size of 32 results in the most processing time due to the least optimized query set. However, utilizing larger block sizes can negatively affect GPU processing time, even though the resulting query set is smaller. Block sizes of 64 and 256 result in the least processing time for queries of length 4 and 8, respectively.

For the remainder of this study, we set the block size to 128, as being a middle point, providing effective common prefix optimization (Figures 3, 5), and less resource contention than in larger block sizes.

5.2 Throughput Characteristics

We show in Figure 7 the GPU throughput (MB/s) of filtering over a 50MB DBLP XML document, while doubling the number of queries. Data is shown for length 4 (L4) and length 8 (L8) queries, with the common prefix optimization ON (Opt) and OFF (Unopt).

The throughput starts off as constant for all query sets, until it halves at every query doubling step. That point differs from one query set to another, based on the final query set size (query nodes), and is affected by the overall available resources on the GPU. When the the latter is under-utilized (small query sets), throughput is constant. Conversely, when over-utilized, the wall-clock time (thus throughput) is directly proportional to the query set size.

For a given optimization setup (on/off), throughput is higher for length 4 queries, as expected. Interestingly, the throughput is higher for optimized length 8 queries compared to length 4 unoptimized queries.

On average, applying the common prefix optimization increases throughput by a factor of 1.6x.

5.3 Speedup vs. Software Approaches

We measured the performance for two CPU-based approaches, namely YFilter and FiST, by measuring the running time for for queries of length 4 and 8. Our results show that YFilter and FiST have similar behavior for most test cases; thus, for the remainder of this paper, we will focus solely on YFilter.

We show in Figure 8 the GPU speedup over YFilter when filtering over a 50MB DBLP XML document, as the number of queries is doubled. Data is shown for length 4 (L4) and length 8 (L8) queries, with the common prefix optimization ON (Opt) and OFF (Unopt).

As the GPU is underutilized, the speedup over software is constant and highest. Beyond those respective points, speedup halves as the number of queries doubles, until slowdown is depicted.

Overall, software processing results in lower throughput for length 8 queries, as the GPU speedup is considerably higher for those corresponding query sets. Filtering length 8 queries results in up to 2.5 orders of magnitude speedup (300x), with an average of 40x; filtering length 4 queries results in up to 10x speedup, with an average of 2x.

6. CONCLUSIONS

In this paper, we presented an XML-based pub-sub filtering framework and algorithm fit for GPUs, exploiting the parallelism found in XPath filtering systems. GPUs are favorable platforms due to the massive parallelism found in their hardware architecture, alongside the flexibility and programmability of software. By utilizing properties of the GPU memory hierarchy, matching thousands of user profiles is achieved at high throughput, requiring minimal update time. Efficient common prefix optimizations are also applied to the query set. Through an extensive experimental evaluation, we show up to 2.5 orders of magnitude speedup (300x) for length 8 queries, with an average speedup of 40x versus the state of the art software approaches. Filtering for queries of length 4 results in up to 10x speedup, with an average of 2x.

7. ACKNOWLEDGEMENTS

This work was partially funded through a Cisco research grant, NSF CCR grants 0905509, 0811416, and NSF IIS

grants 0705916, 0803410, 0910859 and 1144158.

8. REFERENCES

[1] S. Al-Khalifa, H. Jagadish, N. Kodus, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE: Proceedings of the 18th International Conference on Data Engineering*, page 141, Washington, DC, USA, 2002. IEEE Computer Society.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB: Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[3] N. Ao, F. Zhang, D. Wu, D. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. In *VLDB: Very Large Databases*, 2011.

[4] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. Afilter: adaptable xml filtering with prefix-caching suffix-clustering. In *VLDB: Proceedings of the 32nd international conference on Very large data bases*, pages 559–570. VLDB Endowment, 2006.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[6] Z. Dai, N. Ni, and J. Zhu. A 1 cycle-per-byte XML parsing accelerator. In *FPGA: Proc. of the 18th Intl. Symposium on FPGAs*, pages 199–208, New York, NY, USA, 2010. ACM.

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

[8] F. El-Hassan and D. Ionescu. SCBXP: An efficient hardware-based XML parsing technique. In *SPL: 5th Southern Conference on Programmable Logic*, pages 45–50. IEEE, April 2009.

[9] G. Gou and R. Chirkova. Efficient algorithms for evaluating xpath over streams. In *SIGMOD: of the 2007 ACM SIGMOD international conference on Management of data*, pages 269–280, New York, NY, USA, 2007. ACM.

[10] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[11] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[12] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[13] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD: Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, 2010.

[15] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *VLDB: Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 217–228. VLDB Endowment, 2005.

[16] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units, 2008.

[17] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML accelerator engine. In *1st Intl. Workshop on High Performance XML Processing*. Springer Berlin / Heidelberg, 2004.

[18] A. Mitra, M. R. Vieira, P. Bakalov, W. Najjar, and V. J. Tsotras. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR: 4th Conference on Innovative Data Systems Research*. ACM, 2009.

[19] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Accelerating XML query matching through custom stack generation on FPGAs. In *HiPEAC: High Performance Embedded Architectures and Compilers*, pages 141–155. Springer Berlin / Heidelberg, 2010.

[20] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *ICDE: 2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011.

[21] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[22] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB: International Conference on Very Large Data Bases (VLDB)*, 2010.

[23] M. Salloum and V. Tsotras. Efficient and scalable sequence-based XML filtering system. In *WebDB: Proc. of 12th Intl. Workshop on the Web and Databases*. ACM, 2009.

[24] J. Teubner, R. Müller, and G. Alonso. FPGA acceleration for the frequent item problem. In *ICDE: 26th International Conference on Data Engineering Conference*, pages 669–680, 2010.

[25] University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets.

[26] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with fpgas. In *VLDB: Proceedings of the 2010 Very Large Data Bases (VLDB)*, 2010.

[27] The XML Benchmark Projcet. http://www.xml-benchmark.org.

[28] XML Path Language Version 1.0. http://www.w3.org/TR/xpath.

APPENDIX

A. GPU ARCHITECTURES & PROGRAMMING MODEL

Graphics Processing Units (GPUs) are emerging as computational platforms comprising of several hundreds of simple processors operating in a parallel fashion. While intended to be used solely for graphic applications, they are generally employed to accelerate solving general purpose problems of SIMD (Single Instruction Multiple Data) type, thus referred to as General Purpose GPUs (GPGPUs).

GPGPUs are used as co-processors to which the main CPU passes a stream of data; the GPGPU then processes the data with minimal memory footprint, and returns the processing results to the CPU.

Figure 9 shows a high level view of a generic GPU architecture [5]. Streaming Processors (SPs) are simple processor cores that are clustered. Each cluster of SPs is referred to as a Streaming Multiprocessor (SM), such that all SPs within one SM execute instructions from the same memory block. When used with SIMD applications, all SPs on the GPU perform one common operation (at a time) on thousands of data elements.

Furthermore, all SPs within one SM communicate using a low latency shared memory structure. The SM also comprises of a *constant cache*, being a low-latency read-only memory, caching a (limited by size) read-only portion of the device global memory. The constant cache can be used for broadcast-type read operations, where all SPs require reading the same element from global memory. Finally, communication across SPs is achieved through the high latency global memory.

The programmer specifies the kernel that will be running on each of the SPs; however, when spawning the kernels onto the GPU, more instances of the kernel (threads) can be executed than the number of physical processing elements (SPs). The GPU manages switching threads on and off. Moreover, the number of physical cores is abstracted away from the programmer, and is only used at runtime.

Finally, the programmer specifies the number of instances of the kernel that are grouped to execute on a single SM. This group of kernels is referred to as the *block*. As the block size grows, the amount of shared memory available per block is reduced, and contention to computing resources increases; on the other hand, as the block size shrinks, the computational resources are under-utilized. The block size is determined per application basis.

B. XML OVERVIEW

An XML document has a hierarchical (tree) structure that consists of markup and content. Markup also referred to as tags begin with the character '<' and end with a '>'. There are two types of tags, 'start-tags' (for example <author>) and 'end-tags' (for example </author>). Nodes in an XML document begin with a 'start-tag' and end with a corresponding 'end-tag'. XML documents consist of a root node and sub-nodes, which can be arbitrarily nested. Figure 10(a) shows a small XML document example, while Figure 10 (b) shows the XML document's tree representation. Note that the document order corresponds to the preorder traversal of the document tree (shown by the numbers next to the tree nodes). In this example, the 'author' node in the XML tree representation corresponds to the start-tag $< author >$ and end-tag $< /author >$, while 'Yanlei Diao' corresponds to a value (the content of the 'author' node). In this paper, we shall use the terms 'tag' and 'node' interchangeably. For simplicity, Figure 10(b) shows the tags/nodes (i.e. the structural relationship between nodes) in the XML document of Figure 10(a), but not the content (values). The values can be thought as special leaf nodes in the tree (not shown).

[5]We are making use of the NVIDIA CUDA model and terminology.

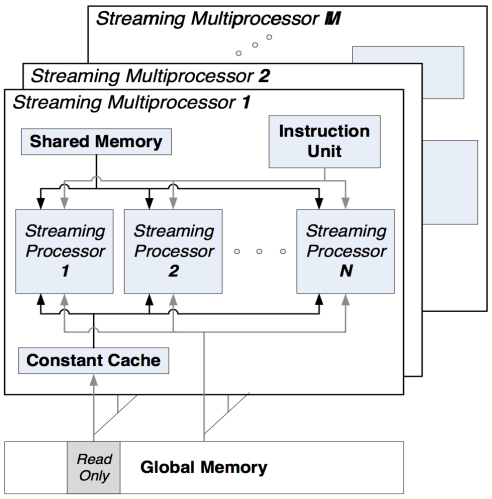

Figure 9: High-level GPU architecture overview.

C. XPATH QUERY LANGUAGE

XPath [28] is a popular language for querying and selecting parts of an XML document. In this paper, we address a core fragment of XPath that includes node names, wildcards, and the /child:: and /descendant-or-self:: axis [6]. The grammar of the supported query language is given below. The query consists of a sequence of location steps, where each location step consists of a node test and an axis. The node test is either an node name, or a wildcard '*'(wildcards can match any node name). The axis is a binary operator that specifies the hierarchical relationship between two nodes. We support two common axes, the child axis and the descendant axis. The child axis, denoted by '/', specifies that an node must be a child of another node in the XML tree, such that two node are one level apart in the tree. The ancestor-descendant axis, denoted by '//' specifies that a node must be a descendant of another node in the XML tree representation.

Lets consider two examples, 'a/b' and 'a//b.' Let *level(x)* denote the level of a node 'x' in a XML tree.

In 'a/b', the child axis specifies:

- The start-tag and end-tag of node **'b'** must be contained *within* the start-tag and end-tag of node **'a'**, and
- level(a) - level(b) = 1

[6]We use '/' and '//' as shorthand to denote the /child:: axis and /descendant-or-self:: axis, respectively.

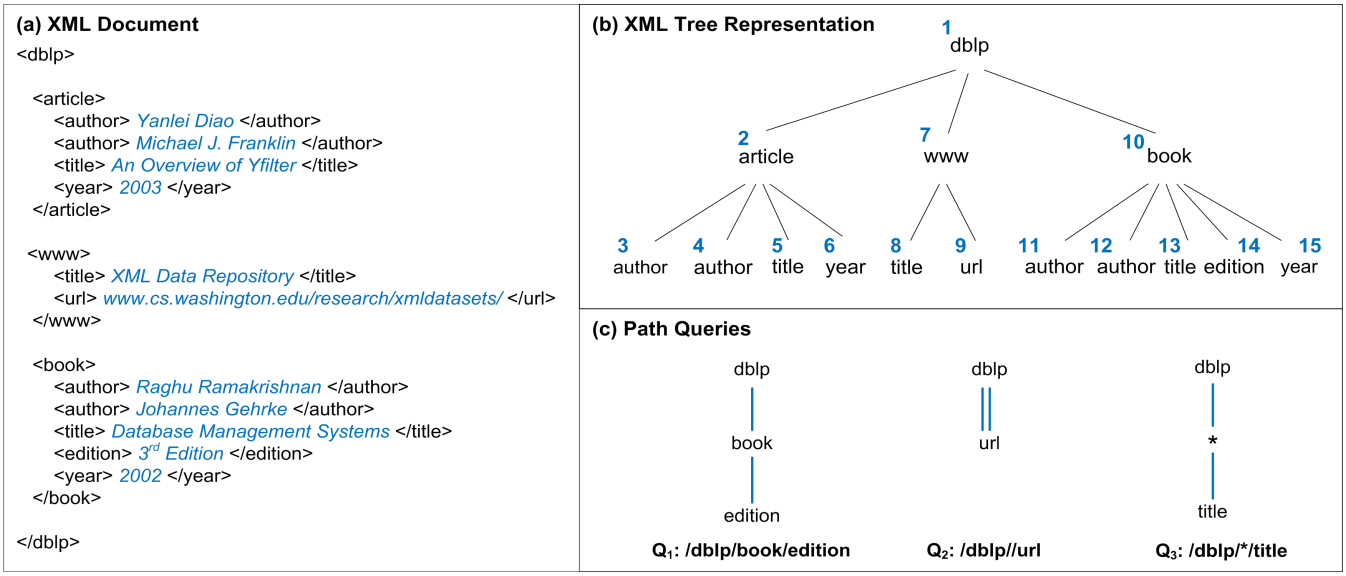

Figure 10: Example XML Document and XML Path Queries, (a) Example XML Document, (b) XML Tree Representation, (c) Example XML Path Queries.

While, in 'a//b', the descendant axis specifies:

- The start-tag and end-tag of node **'b'** must be contained *within* the start-tag and end-tag of node **'a'**, and
- level(a) - level(b) ≥ 1

Example path queries are shown in Figure 10 (c). Consider Q_1 (/dblp/book/edition) in 10 (c) which is a path query of depth three, and specifies a structure which consists of nodes 'dblp', 'book' and 'edition' where each node is separated by a '/' operator. This query is satisfied by nodes (dblp, 1), (book,10), and (edition, 14) in the XML tree shown in Figure 10(b). Q_2 (/dblp//url) is a path query of depth two, and specifies a structure which consists of two nodes, 'dblp' and 'url' are separated by the '//' operator. Q_2 specifies that the node 'url' must be descendant of the 'dblp' node, and the two nodes should be one or more levels apart. For example, the nodes (dblp,1) and (url,9) in Figure 10(b) satisfy this query structure. Q_3 (/dblp/*/title) is a path query of depth three, and specifies a structure that consists of two nodes and a wildcard (which matches any node). This query has two matches in the XML tree in Figure 10(b). The nodes (dblp,1), (article,2), and (title,5) satisfy one match, while nodes (dblp,1), (book,10), and (title,13) satisfy another match for Q_3.