RC24926 (W0912-136) December 31, 2009
Computer Science

IBM Research Report

To Parallelize or Not to Parallelize: XPath Queries on Multi-core Systems

Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, Bryant Wei-Lun Kok
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

To parallelize or not to parallelize: XPath Queries on Multi-core Systems

Rajesh Bordawekar
IBM T.J. Watson Research Center
bordaw@us.ibm.com

Lipyeow Lim
IBM T.J. Watson Research Center
liplim@us.ibm.com

Anastasios Kementsietsidis
IBM T.J. Watson Research Center
akement@us.ibm.com

Bryant Wei-Lun Kok
IBM Integrated Supply Chain Lab
kokwlb@sg.ibm.com

ABSTRACT

With wide availability of commodity multi-core systems, it is imperative to understand what, if any, changes are needed to existing software systems to harness the newly available computational power. In this context, this work explores acceleration of XML processing systems. Specifically, we investigate parallelization of individual XPath queries over shared-address space multi-core processors. Unlike past approaches that have considered a distributed setting or ad hoc parallel solutions, ours is the first methodical end-to-end proposal. Our solution first identifies if a particular XPath query should be parallelized and then determines the optimal way of parallelizing that query. This decision is based on a cost-base approach that relies both on the query specifics and data statistics. At each stage of the parallelization process, we evaluate three alternative approaches, namely, data-, query-, and hybrid-partitioning. For a given XPath query, our parallel cost model uses selectivity and cardinality estimates to compute costs for these different alternatives. The costs are then fed to parallel query optimizer that generates an optimal parallel execution plan. We have implemented a prototype end-to-end Parallel XPath processing system that integrates the XPath parser, cost estimator, query optimizer, and a parallel runtime library . We use this system to evaluate efficacy of our proposal by an extensive set of experiments using well-known XML documents. These results conclusively validate our parallel cost model and optimization framework, and demonstrate that it is possible to accelerate XPath processing using commodity multi-core systems.

1. INTRODUCTION

For a number of years, the evolution of hardware systems followed a rather *predictable* trend in terms of processing capabilities: the latest generation of processors was significantly faster than the previous, with the rate of speed increase following closely Moore's law. However, higher processor speeds did not always translate to corresponding gains in system performance (with memory speeds and instruction sets often becoming the new performance bottlenecks). This led hardware manufactures to consider alternative architectures in which multiple processing *cores* are used to execute instructions *in parallel*. So, whereas the trend before was to increase processor speeds between hardware generations, in the last few years a new trend has emerged where the difference between hardware generations is in the number of cores. Nowadays, it is not uncommon to find eight cores, even in commodity hardware.

Of course, to take advantage of these multiple cores, the parallelization of existing software systems comes at a cost. The systems often cannot be used as such and might need to be changed. Indeed, there has been a lot of interest in systems research, including database systems research [21], on how to harness this processing power. In this context, we investigate here the problem of how to parallelize the evaluation of XPath [9] queries over XML documents in a shared-address space multi-core system. To emphasize the importance of this problem we note that XML is the de facto data representation format used nowadays, and XPath queries are commonly used as such (or as part of XQuery expressions) to query XML data. Parallel query evaluation in this setting is as important, and not unlike, the early works on parallel evaluation of SQL queries over relational data [15]. In spite of sharing motivation however, the commonalities between the relational/SQL and XML/XPath settings are few and techniques from the former setting do not carry to the latter. In what follows, we review some of the main challenges in the parallelization of XPath queries.

Consider the XML document in Figure 1(a) (conforming to XMark [25]) whose document tree is shown in Figure 1(b). Assume that we want to evaluate the XPath query `/site/regions/*` to retrieve the names of all regions from our document (where `*` denotes the wildcard and can match any element). Should we parallelize the evaluation of the query or not? Clearly, this decision depends both on the query itself and on the characteristics of the document. In XMark, there is usually only a limited number of region nodes, each corresponding to a continent. Therefore, our query will access only a small number of nodes. Given that any form of parallelism is expected to also incur some cost in the evaluation, it seems that in this particular setting any benefits from parallelism are either insignificant or are alleviated by the cost of parallelism. Therefore, a serial execution seems preferable. However, what if a region node exists for each, say, county in the United States? Then, with approximately 3,000 possible region nodes, for the same query it seems reasonable to try to parallelize the evaluation of the query by considering, in parallel, all the regions, say, by state. In general, given a document and a query, our *first* challenge here will be to decide whether or not to parallelize the evaluation of the query.

For the simple example query, once the decision is made to parallelize, it is rather straightforward to decide *how* the query is parallelized: each core evaluates the initial query over a subset of the regions (i.e., document), which is an example of what we call a *data partitioning* [5] parallelization strategy (more on this later). In

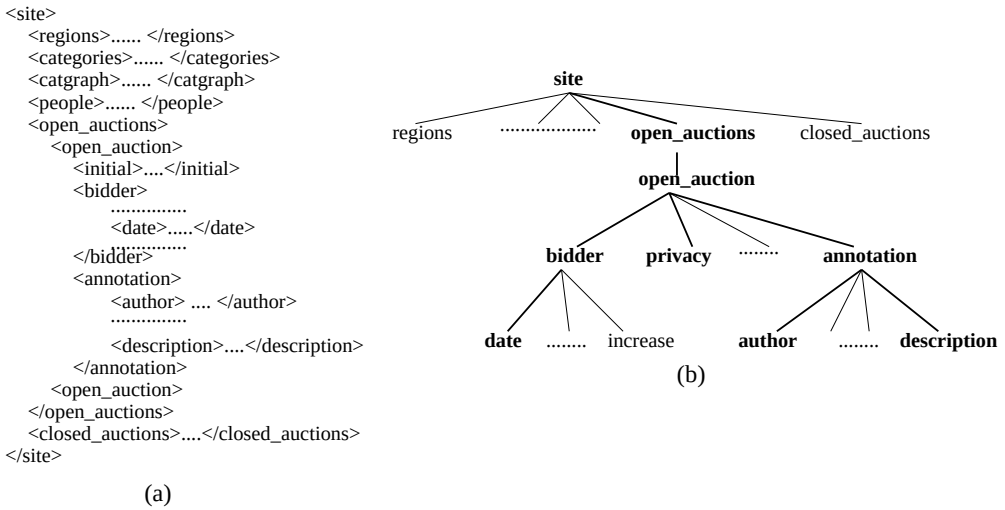

Figure 1: An Example XML Document: xmark.xml. The traversed edges of the tree by the XPath query are highlighted.

reality however, there will be multiple ways to parallelize a query, each of which might use a different strategy. To see this, consider for example the query in Figure 1(c). Assuming that there is a large number of `open_auction` nodes in the document, we might decide to parallelize on the third *step* of the query (hereafter referred to as a *partitioning point*). Data partitioning here dictates that we evaluate the first three steps of the query sequentially and then each core evaluates the predicate over a subset of the `open_auction` nodes retrieved by the serial evaluation. However, another parallelization strategy is also possible here. Using the *query partitioning* [5] parallelization strategy, we rewrite the initial query into three new queries, with each of the three predicates of the initial query appearing in only one of the resulting queries. For example, `/site/open_auctions/open_auction[annotation/author]` is one of the three queries. Then, each rewritten query is evaluated by a different core and the final result is computed by the intersection of results in all the cores.

Given the two alternative strategies, how can we choose which one to use to parallelize our query? Even if it is clear that one of the two strategies is the most promising, how can we be certain that parallelizing our initial query at a different step, say, in `annotation` might not give better response times. In general, for an XPath query with a total of n steps (including steps within predicates), each step could be a candidate partitioning point for parallelization. A parallel query plan might contain a subset of the partitioning points. Hence the number of possible parallel query plan is $\mathcal{O}(2^n)$. For a given subset of partitioning points, the parallelization strategy at each point, and the order of the partitioning points may further result in different *parallelization plans*. Coming up with a way to systematically search and find good parallelization plans in this huge search space is the *second* challenge we address in this paper. Unlike the work by [5] which builds such plans in an *ad hoc* manner, our objective here is to provide a solution that uses a cost-based approach to distinguish between alternative plans. Coming up with an appropriate cost-model for the parallelization of XPath queries is our *third* challenge.

The contributions of this paper are summarized as follows:

- We introduce and address the problem of optimizing XPath queries on shared memory, multi-core processors. To the best of our knowledge this is the *first* work that offers a systematic way to address the challenges in this domain.
- We adapt and extend current sequential XPath cost estimation models to model the cost of parallel XPath processing plans.
- We propose an optimization algorithm that uses the cost model together with several heuristics to find and select parallelization points in an XPath query. Once the parallelization points are selected, parallel query plans are generated and executed.
- We implement our optimizer in a prototype end-to-end XPath processing system. We provide experimental results on this system that validate the effectiveness of our optimizer on realistic XML workloads.

The rest of the paper is organized as follows: Section 2 discusses related work in parallel query processing of relational and XML data. Section 3 introduces the three strategies for parallel execution of XPath queries: data, query, and hybrid partitioning. Section 4 describes the models used to estimate costs of different parallel execution strategies. Section 5 presents the cost-based parallel query optimizer. Section 6 presents experimental evaluation of the optimizer using queries from realistic XML workloads. Finally, we conclude in Section 7.

2. RELATED WORK

Orthogonal to our work, and thus not the focus of this paper, the problems of XML cardinality and selectivity estimation have been extensively studied [2, 7, 23, 24, 16, 11, 29, 28, 17, 3, 27]. Our work adapts and uses many of the estimation models (e.g., the Markov model) proposed in the literature.

Cost-based query optimization in XML databases, although not as well covered in the literature as selectivity estimation, has been employed successfully in commercial databases like IBM DB2 pureXML [3, 4]. Balmin et al. [3, 4] outlines some of the cost models and optimization heuristics used in DB2 pureXML. Hidaka et al. [13] outlines a cost model for XQuery. Zhang et al. also proposed a statistical learning-based approach [30] for modelling the cost of XPath queries. These cost-based query optimization ap-

proaches deal solely with sequential execution plans. Our work addresses cost-based optimization issues associated with parallelization. We adapt some of the existing cost models for sequential portions of our parallel execution plans, but the cost models that we developed for evaluating parallelization decisions have not been addressed in previous literature.

Parallelization of SQL queries has been extensively studied in the context of both distributed and centralized repositories [14, 15, 18]. Most commercial database systems support parallel query processing in either shared-nothing or shared-everything architectures. Parallelization has been extremely effective in practice, for both OLTP, OLAP/data warehousing, and web applications. Parallelization of SQL queries differs from the XPath parallelization as follows: (1) The SQL workload supports in-place updates, while XPath processing is read-only; (2) The relational data has a regular 2-dimensional structure that is suitable for partitioning either along rows or columns. The rooted hierarchical structure of XML is not inherently suited for balanced data partitioning; (3) Using hash-partitioning, it is easier to physically distribute relational data across multiple storage nodes while maintaining data affinity. For XML documents, it is very difficult to effectively physically cluster related items; and (4) Unlike relational data, XML can be accessed and stored in many different ways, e.g., in-memory, streaming, relational or native storage. XPath parallelization algorithms need to be tuned to match the XML storage and access characteristics.

Past studies have evaluated XML processing either in distributed or concurrent scenarios. Most existing XML processing engines are thread-safe and allow multiple threads to issue concurrent XPath queries againts an XML document. Distributed XML processing is discussed in [6, 8]. The work in [6] considers initially Boolean XML queries expressed in a language containing forward axes, labels, text and the Boolean operators and, or and not. The algorithms are inspired by partial evaluation. In essence, the whole query and all its sub-queries are evaluated in each distributed fragment. During query evaluation, data unknown at some fragment is replaced by Boolean variables. Therefore, the computation at a fragment may result in a Boolean expression in terms of these variables, hence the relationship to partial evaluation. When all fragments complete computing, the final Boolean result may be resolved. The main advantage of the scheme is that computation at various fragments proceeds in parallel and incurs a computational overall cost similar to that of a centralized mechanism. The work in [8] extends the ideas from Boolean to node-returning queries. The idea is to normalize queries, and to treat separately the qualifiers in a query and the selection (main skeleton) part of the query. The various qualifiers are treated using the techniques of [6]. The evaluation of the selection path also uses partial evaluation ideas to "transmit" information between fragments. The overall scheme of [6, 8] is elegant and theoretically efficient. However, one of its limitations is that these fragments need be constructed statically. Issues of load balancing and performing the partition optimally or dynamically have not been addressed.

The work of [26] treats distributed query evaluation on semistructured data and is applicable to XML query processing as well. It treats three overlapping querying frameworks. The first is essentially regular expressions. The second is based on an algebra, C, and is aimed at restructuring. An algebraic approach based on query decomposition is provided for solving C queries. Here a query is rewritten into subqueries implied by the distribution. These queries are evaluated at the distributed fragments to produce partial results which are later assembled into a final result. The third is *select-where queries*, declarative queries combining patterns, regular expressions and some restructuring. Here, processing is done in two stages where the first is evaluating a related query that is expressible in C, and hence parallelizable, which produces partial results that are then used to form the final result at the client. The focus is on communication steps.

One may approach the problem of parallelizing XML query processing within the general framework of efficiently programming and coordinating multiprocessor computations (see [12] for a comprehensive treatment). This is the approach taken in [20, 22]. Execution of various XML processing tasks (not including query processing) appears in [20] in the context of multicore systems. The idea is to have a crew of processes each taking tasks out of its own work queue. Once tasks are exhausted, a process may *steal* tasks off queues of other processes. Tasks are ordered so that processing is done at the top whereas stealing is done at the bottom. This creates less contention. A scheme is presented for constructing the final result. The paper presents the idea of *region-based task partitioning* to increase task granularity. Parallel XML DOM parsing is presented in [19, 22]. The first paper uses a dynamic scheme for load-balancing among cores. The idea in the second paper is to statically load-balance the work among the cores. This latter work is targeted at large shallow files containing arrays and does not scale to many cores (beyond six).

3. PRELIMINARIES

3.1 XPath queries

We briefly review the fragment of XPath considered in this paper. We consider the class $\mathcal{Q}$ of XPath queries of the form:

$$\mathcal{Q} ::= \epsilon \mid t \mid * \mid \mathcal{Q}/\mathcal{Q} \mid \mathcal{Q}[p],$$
$$p ::= \mathcal{Q} \mid \mathcal{Q}/\textit{text()} = \text{`}c\text{'} \mid \mathcal{Q}/\textit{label()} = l \mid \mathcal{Q}/\textit{pos()}\ op\ i \mid \mathcal{Q} \wedge \mathcal{Q} \mid \mathcal{Q} \vee \mathcal{Q}$$

where ϵ is the empty path (*self*), t is a tag, $*$ is a wildcard (matches any tag), and '$/$' is the *child-axis*; $[p]$ is referred to as a *predicate*, in which $\mathcal{Q}$ is an expression, c and l are string constants, op is any one of $\leq, \geq, <, >, =$, i is an integer, and $\wedge, \vee$ are the Boolean conjunction and disjunction, respectively. Notice that in the paper we are currently considering only queries with downward modalities, since these are the most commonly used in practice. Parallelizing such queries is already challenging, as the following sections illustrate. For these queries, we do support complex nested predicates, which include boolean combinations of sub-predicates, and tests on label names, contents and positions.

In the next sections, we often distinguish the processing of a query $\mathcal{Q}$ from that of its predicates at the various query steps.

3.2 Partitioning Strategies

As mentioned in the introduction, ad hoc parallelization of individual XPath queries was first explored in [5]. The authors discussed various factors affecting the XPath parallelization and presented three strategies for parallelizing individual XPath queries: 1) Data partitioning; (2) Query partitioning; and (3) Hybrid partitioning. In what follows, we review these strategies in more detail.

The three parallelization strategies are defined over the abstract XML data model. As a result, they apply to any storage implementation of the XML data model. In this work, we assume that the pre-parsed XML document is stored using an in-memory, non-relational representation and it can be concurrently accessed by multiple application threads in a shared-address space environment. The three parallelization strategies differ in the way the shared XML data is logically partitioned across multiple processors and how the input query is executed on the partitioned data. All three strategies require some form of query re-writing.

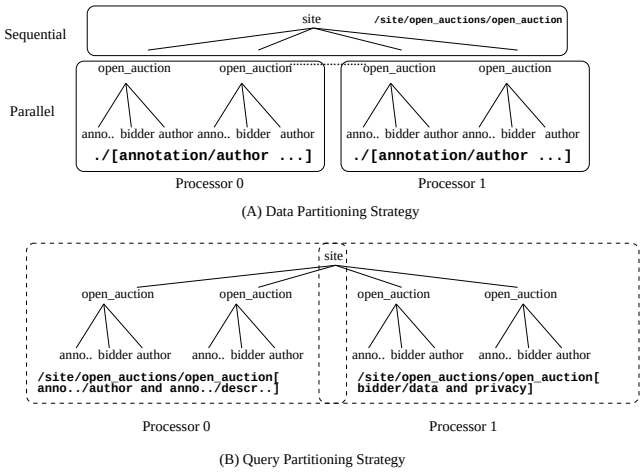

Figure 2: Data and Query Partitioning Strategies

In the data partitioning approach, the input XPath query is partitioned into serial and parallel queries. The serial part of the input query is executed by a single processor over the entire document. The resulting node set is then equally distributed across multiple processors. Each participating processor then uses the locally assigned node set as the set of context nodes and executes the parallel sub-query. This approach achieves parallelism by concurrently executing the *same* XPath query on *distinct* sections of the XML document. The scalability in the data partitioning scheme is determined by the sequential sub-query; an expensive sequential execution can degrade the performance of the entire query. Therefore, in the data partitioning approach, it is important to partition the query so that the serial portion performs the least amount of work. Figure 2(a) illustrate the execution of the XPath query presented in Figure 1 using the data partitioning approach. The original query is split into two sub-queries: a serial sub-query, `/site/open_auctions/open_auction` and the predicated sub-query, `./[anno.. and ..]`. The serial query is executed by a processor and the resulting node set of `open_auction` nodes is distributed over the participating processors. Each processor then executes the predicated sub-query on its assigned nodes. The result of the original query can then be computed by merging local results from participating processors.

In the query partitioning approach, the input query is rewritten into a set of 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, the modified query is an invocation of the original query using different parameters. Each processor executes its assigned query on the entire XML document. The final result of the query can be then computed using either the union or merge of the per-processor node sets. Unlike the data partitioning approach, this approach achieves parallelism via exploiting potentially non-overlapping navigational patterns of the queries. In this approach, the overall scalability is determined by the range of the concurrent queries. If their traversals do not overlap significantly, the query performance will scale as the number of processors is increased. Figure 2(b) illustrate the execution of the XPath query presented in Figure 1 using the query partitioning approach. In this scenario, the original query is re-written into two distinct predicated queries, each executing a part of the original predicate. Each new query is executed by a separate processor over the entire XML document. The final result is computed by intersecting two local result sets. Alternatively, the query partitioning approach rewrite a query using range partitioning. For example, consider the query, `/a/b`, where the node `a` has 20 `b` children. The query partitioning strategy can rewrite this query for 2 processors by partitioning the node `b`'s node set by 2, i.e., processor 0 will execute the query, `/a/b[position()<11]`, and processor 1 will execute the query, `/a/b[position()>10]`. Since the execution pattern of such plan is very similar to the data partitioning plan, we do not evaluate this query partitioning strategy any further.

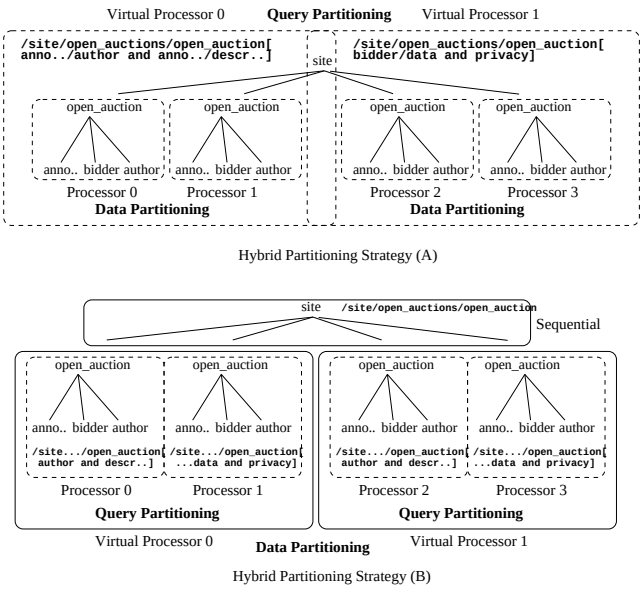

Figure 3: Hybrid Partitioning Strategy

The data and query partitioning approach can be integrated into a hybrid partitioning approach. Figure 3 illustrates two possible implementation of the XPath query using the hybrid partitioning approach. In the first implementation (Figure 3(a)), the input query is first re-written using the query partitioning approach for a set of *virtual* processors for the entire XML document. Each virtual processor is a set of physical processors and it executes its assigned query using the data partitioning approach. Specifically, if the virtual processor consists of two physical processors, one of the processors will first execute the serial portion of the assigned query and then the two processors will concurrently execute the parallel portion of the query using their allocated context nodes. Alternatively, the input query can be first re-written using the data partitioning strategy over a set of virtual processors and the parallel sub-query can be then executed using query partitioning strategy over the physical processors within a virtual processor (Figure 3(b)). The hybrid partitioning strategy is a generalized form of the query and data partitioning strategy and can be used recursively.

Experimental results presented in [5] have demonstrated that the three parallelization strategies are indeed very effective in practice. For a majority of XPath queries under evaluation, the performance scaled linearly as the number of threads was increased. However, there were a few cases where the performance *degraded* when the original query was parallelized. Further, the performance of queries parallelized using the data partitioning strategy depended on the way the original query was split into serial and parallel queries. In [5], the query splitting was performed in an *ad hoc* manner, without using any rule- or cost-based heuristics.

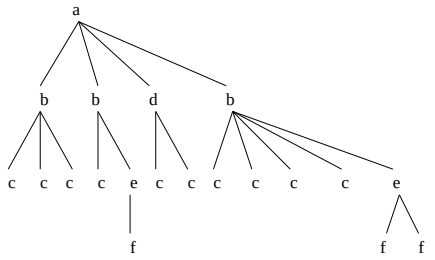

Figure 4: An example of an XML data tree.

4. COST MODEL

Our optimizer uses a cost-based model together with heuristics in order to find an efficient parallel query plan. Recall that the search space of all possible query plans (parallel and sequential) is super-exponential. A cost model is used to quickly evaluate a candidate plan (or relevant portions of a plan) to determine if it is likely to be an efficient plan. However, running the cost model on every possible plan in the search space is infeasible. Hence, we use heuristics in combination with the cost model to prune the search space.

The following factors affect parallelization decision:

• cardinality of a step: if there are too few node instances matching a particular step, performing parallelization via data partitioning at that step is not feasible.

• number of branches in the predicates of a step: If there are no predicates or very few branches in the predicate, performing parallelization via query partitioning at that step is not feasible.

• amount of work done via sequential and via parallel processing: For overall speedup, the sequential work should be minimized and the maximum amount of work parallelized.

Our cost model quantifies the processing cost of three basic ways of processing an XPath query: sequential, data partitioning, and query partitioning. The cost functions for data partitioning and query partitioning both rely on the cost function for sequential processing. Key components of these functions are the notions of *cardinality* and *selectivity*. While the two terms are sometimes used interchangeably in the literature, in our work there is a clear distinction between the two. There is a substantial body of work on cardinality and selectivity estimation. None of these problems is the main focus of our work. Instead, we rely and extend (where appropriate) existing definitions.

4.1 Statistics

XML statistics collection is a well-studied topic, and here we summarize the statistics needed by our cost model and optimizer. For details on the algorithms for collecting and storing these statistics we refer to [2, 16, 17, 3]. We collect three types of statistics:

Single tag count $f(t_i)$ counts the number of node instances in the XML data tree that matches the tag t_i,

Fanout count $f(t_i|t_{i-1})$ counts the average number of child node instances matching t_i for each parent node matching t_{i-1},

Children count $f(*|t_{i-1})$ counts the average number of child node instances (regardless of tag) for each parent node matching t_{i-1}.

Although we use a first order Markov model for our statistics, our optimizer is general and higher order Markov models or other models can be used as well. Under this simplifying assumption, to compute the above three statistics, it is sufficient to collect single tag and tag-tag pair counts, as described in [2, 16].

4.2 Cardinality

We first use the collected statistics to estimate the cardinality of each step in an XPath expression. The cardinality of a step in an XPath expression is the number of nodes in the XML data tree that satisfy the conditions of that step.

EXAMPLE 1. *Consider the XML data tree in Fig. 4. The cardinality of $/a$, $/a/b$ and $/a/b/c$ are 1, 3, and 8 respectively.*

Consider an XPath expression $\mathcal{Q} = /t_0/t_1/\ldots/t_i/\ldots/t_k$ (with no predicates for now), where each t_i is either a tag or the wildcard $*$. Let $\mathcal{Q}_i$ denote the sub-expression of $\mathcal{Q}$ up to step t_i. Then, the cardinality of $\mathcal{Q}_i$ is estimated by the recurrence relation,

$$card(\mathcal{Q}_i) = \begin{cases} 1 & \text{if } i = 0 \\ f(t_i|t_{i-1})card(\mathcal{Q}_{i-1}) & \text{otherwise} \end{cases} \quad (1)$$

Cardinality, as define here, is similar to the definition in [3].

EXAMPLE 2. *Consider the XML data tree in Fig. 4. The cardinality of $/a/b/c$ can be estimated as,*

$$\begin{aligned} card(/a/b/c) &= f(c|b)card(/a/b) \\ &= f(c|b)f(b|a)card(/a) \\ &= \frac{8}{3}\cdot\frac{3}{1}\cdot 1 = 8 \end{aligned} \quad (2)$$

Similarly, it is not hard to see that the cardinality of $/a/b/*$ is 10.

4.3 Selectivity

In order to estimate the cardinality of more complex XPath expressions that contain predicates, the notion of selectivity is needed. Selectivity is a fraction associated with a predicate that quantifies the filtering power of the predicate.

EXAMPLE 3. *Consider the XML data tree in Fig. 4. The selectivity of predicate $[e/f]$ in $/a/b[e/f]$ is $\frac{2}{3}$.*

Consider the XPath expression $\mathcal{Q} = /t_0/t_1/\ldots/t_i[t_{i,0}/t_{i,1}/\ldots/t_{i,j}/\ldots/t_{i,m}]/\ldots/t_k$, and let $\mathcal{Q}_i$ denote the sub-expression of $\mathcal{Q}$ up to step t_i. Also, let p_i denote the predicate $t_{i,0}/t_{i,1}/\ldots/t_{i,j}/\ldots/t_{i,m}$ of t_i and $p_{i,j}$ the sub-predicate of p_i up to step $t_{i,j}$. Then, the selectivity of $p_{i,j}$, denoted by $sel(p_{i,j})$, can be computed using the recurrence relation,

$$sel(p_{i,j}) = \begin{cases} \min(f(t_{i,0}|t_i), 1.0) & \text{if } j = 0 \\ \min(f(t_{i,j}|t_{i,j-1}), 1.0)sel(p_{i,j-1}) & \text{otherwise} \end{cases} \quad (3)$$

EXAMPLE 4. *Consider the XML data tree in Fig. 4. The selectivity of predicate $p = [e/f]$ in $/a/b[e/f]$ can be estimated as,*

$$\begin{aligned} sel(e/f) &= \min(f(f|e), 1.0)sel(e) \\ &= \min(f(f|e), 1.0)\min(f(e|b), 1.0) \\ &= \min(\frac{3}{2}, 1.0)\min(\frac{2}{3}, 1.0) = \frac{2}{3} \end{aligned} \quad (4)$$

When a predicate is a boolean combination of sub-predicates, the selectivity of the whole expression is computed from the selectivity of the component sub-predicates using the following rules,

$$sel(p \text{ AND } p') = \min(sel(p), sel(p')) \quad (5)$$

$$sel(p \text{ OR } p') = \max(sel(p), sel(p')) \quad (6)$$

where p and p' are the predicate sub-expressions.

Given the selectivity of predicates, we can now refine the cardinality estimation (of Eqn. 1) to account for the presence of predicates. This can be done by multiplying the cardinality of a step with the selectivity of the associated predicate.

$$card(\mathcal{Q}_i[p_i]) = \begin{cases} 1 & \text{if } i = 0 \\ f(t_i|t_{i-1}) card(\mathcal{Q}_{i-1}[p_{i-1}]) sel(p_i) & \text{otherwise} \end{cases} \quad (7)$$

Of course, not all steps in a query have predicates. For example, in the query $/a/b[e/f]$ only the second step has a predicate. In order to be able to use the above formula uniformly for all steps of any query, we introduce the notion of the *empty* predicate $[\epsilon]$ (note that the empty predicate is supported by the query grammar introduced in the previous section). We define the selectivity of the empty predicate to be equal to 1 and therefore any query in our grammar can be rewritten to an equivalent query where each step has a predicate. For example, query $/a/b[e/f]$ can be rewritten to query $/a[\epsilon]/b[e[\epsilon]/f[\epsilon]]$. Then, Eqn. 7 can be used to compute the cardinality of each step. Hereafter, whenever we compute cardinality, we will always use this formula on queries whose steps always includes (empty) predicates.

4.4 Sequential Cost

Consider the XPath expression $\mathcal{Q} = /t_0[p_0]/\ldots/t_{i-1}[p_{i-1}]/t_i[p_i]/\ldots/t_k[p_k]$, where each p_i is either a predicate of the query or an introduced empty predicate. Suppose the prefix $\mathcal{Q}_{i-1}[p_{i-1}]$ has been processed (all the steps and predicates up and including step t_{i-1}) resulting in a node set $\mathcal{N}_{i-1}$. For each node in the node set $\mathcal{N}_{i-1}$, the average cost of traversing the remaining suffix (starting with step t_i) of the XPath expression on a single processor model can be estimated by,

$$cost(t_i) = \begin{cases} f(t_i|t_{i-1}) & \text{if } i = k \text{ and } p_i = \epsilon \\ f(t_i|t_{i-1})\,[cost(p_i) & \\ \quad + f(*|t_i)C_{step}] & \text{if } i = k \text{ and } p_i \neq \epsilon \\ f(t_i|t_{i-1})\,[cost(t_{i+1}) & \\ \quad + cost(p_i) + f(*|t_i)C_{step}] & \text{otherwise} \end{cases} \quad (8)$$

where $cost(p_i)$ is the cost of processing the predicate p_i, and C_{step} is the overhead associated with processing a step. The intuition for the recursion is as follows. Starting from a single node matching t_{i-1} (henceforth the parent node), there are on average $f(t_i|t_{i-1})$ child nodes that match t_i. For each node that matches t_i (henceforth current node), the average cost can be computed as the sum of $cost(t_{i+1})$ (computed recursively), the cost $cost(p_i)$ of processing the predicate p_i associated with the current node, and an overhead associated with processing child steps from the current node. In order to process both the predicate and the t_{i+1} step, all the children of the current node need to be scanned once. The cost of this scan is captured by the average number of children of the current node multiplied by C_{step}, the overhead associated with processing a child step. In terms of $cost(p_i)$, in general, a predicate p_i is a boolean combination of XPath expressions. Hence, $cost(p_i)$ can be estimated recursively computing the cost of the constituent XPath expressions and summing the costs together.

EXAMPLE 5. *The cost for the XPath expression /a/b[c and e/f] can be estimated by essentially estimating the cost of the query root.*

$$\begin{aligned} cost(a) &= f(a|root)\,[cost(b) + f(*|a)C_{step}] \\ &= 1\,[cost(b) + 4C_{step}] \\ &= \{f(b|a)\,[cost(c \text{ and } e/f) + f(*|b)C_{step}]\} + 4C_{step} \\ &= 3\left[cost(c \text{ and } e/f) + \frac{10}{3}C_{step}\right] + 4C_{step} \\ &= 3\,[cost(c) + cost(e)] + 14C_{step} \\ &= 3\,\{f(c|b) + f(e|b)\,[cost(f) + f(*|e)C_{step}]\} + 14C_{step} \\ &= 3\left\{\frac{8}{3} + \frac{2}{3}\left[f(f|e) + \frac{3}{2}C_{step}\right]\right\} + 14C_{step} \\ &= 8 + 2\left[\frac{3}{2} + \frac{3}{2}C_{step}\right] + 14C_{step} \\ &= 11 + 17C_{step} \end{aligned}$$

Note that the cost computed by Eqn. 8 is for each instance of the node set matching the previous step. To obtain the total cost of traversing the suffix starting at t_i, the average cost $cost(t_i)$ needs to be multiplied by the cardinality $card(\mathcal{Q}_{i-1}[p_{i-1}])$ of the nodeset from the previous step.

4.5 Data Partitioning Cost

Once more, consider the XPath expression $\mathcal{Q} = /t_0[p_0]/t_1[p_1]/\ldots/t_i[p_i]/\ldots/t_k[p_k]$. The cost of evaluating the XPath fragment starting at t_i using data partitioning at t_i over n processors can be estimated as

$$\begin{aligned} DPcost(t_i, n) &= \frac{1}{n} \cdot card(\mathcal{Q}_{i-1}[p_{i-1}]) \cdot cost(t_i) \\ &\quad + card(\mathcal{Q}_i[p_i]]) \cdot tempResultOverhead \\ &\quad + n \cdot C_{par} \end{aligned} \quad (9)$$

Note that $DPcost(t_i, n)$ does not take into account the cost of traversing from the beginning of the XPath expression to t_i.

The first pass of our optimizer does not consider the number of processors when deciding whether a particular step should be a candidate for parallelization via data partitioning. Moreover, the data partitioning cost function (Eqn. 9) is non-monotonic. Hence, the candidacy decision is made based on the optimal data partitioning cost over any number of processors,

$$DPcost_{opt}(t_i) = \min_n DPcost(t_i, n) \quad (10)$$

$$n_{opt} = \arg\min_n DPcost(t_i, n) \quad (11)$$

4.6 Query Partitioning Cost

Consider the XPath expression $/t_0/\ldots/t_i[p_i]/\ldots/t_k$. The predicate p_i is a boolean combination of predicate XPath expressions of the form $p_{i,0}\ op\ p_{i,1}\ op\ \ldots\ op\ p_{i,n-1}$, where each op can be a conjunction or a disjunction. The cost of evaluating the boolean combination of predicates associated with t_i using query partitioning of the n predicates over n processors can be estimated as

$$\begin{aligned} predQPcost(p_i) &= card(\mathcal{Q}_{i-1}[p_{i-1}]) \cdot \left[\max_{0<j<n} cost(p_{i,j})\right] \\ &\quad + booleanOverhead(p_i) \\ &\quad + n \cdot C_{par} \end{aligned} \quad (12)$$

In fact the boolean combination is parenthesized into a binary tree and the overhead of merging the results after the parallelized predicate XPath expressions have completed is dependent on this

Algorithm 1 OPTIMIZER($xpath, S$)

Input: XPath Expression $xpath$, Data Statistics S
Output: multi-threaded query plan

1: $Q \leftarrow$ XPathParser($xpath$)
2: $\mathcal{P} \leftarrow \emptyset$
3: ANNOTATEQUERYTREE(Q, $FirstStep(Q)$, S, $\mathcal{P}$)
4: /* Choose Partitioning Points */
5: Sort $\mathcal{P}$ by depth and parallel cost
6: $\mathcal{P}_{opt} \leftarrow$ Pick top k points according to heuristics (Sec. 5.2)
7: $Plan \leftarrow$ Construct parallel plan using $\mathcal{P}_{opt}$

binary tree. The overhead is computed using the following recursive formula,

$$booleanOverhead(p) = \begin{cases} 0 & \text{if } p \text{ is atomic} \\ ANDoverhead\,[sel(lhs(p)) & \\ \quad +sel(rhs(p))] & \\ \quad +booleanOverhead(lhs(p)) & \\ \quad +booleanOverhead(rhs(p)) & \\ & \text{if } Op(p) \text{ is AND} \\ ORoverhead\,[sel(lhs(p)) & \\ \quad +sel(rhs(p))] & \\ \quad +booleanOverhead(lhs(p)) & \\ \quad +booleanOverhead(rhs(p)) & \\ & \text{if } Op(p) \text{ is OR} \end{cases} \quad (13)$$

Note again that the query partitioning cost at step t_i is computed as the average cost for each instance node matching t_i. Hence, the total query partitioning cost at t_i needs to be computed by multiplying with the cardinality of t_i.

5. COST-BASED PARALLELIZATION ALGORITHM

Our high level multicore query plan optimizer is outlined in Algorithm 1. The input XPath expression is first parsed into a query tree using a standard XPath parser. The optimizer then makes two passes over the query tree. In the first pass (line 3), the optimizer uses data statistics to estimate the cardinality and costs of each step in the XPath expression and identifies a set $\mathcal{P}$ of candidate points for parallelization based on local conditions. In the second pass (line 4-7), the optimizer evaluates each candidate parallelization points using heuristics that take into account global conditions, and picks the most promising parallelization point(s). Once the parallelization points have been chosen, a multi-threaded query execution plan can be constructed (line 7). We describe the two passes in greater detail next.

5.1 Finding Candidate Partitioning Points

The first pass in our optimizer identifies candidate partitioning points in the query tree using a cost model. Each node in the query tree is traversed and evaluated using two mutually recursive procedures ANNOTATEQUERYTREE and ANNOTATEQUERYTREEPRED. Conceptually, the procedure ANNOTATEQUERYTREE iterates over each linear step of an XPath expression, while ANNOTATEQUERYTREEPRED iterates over the boolean expressions contained in predicates.

Algorithm 2 outlines the logic of ANNOTATEQUERYTREE. The ANNOTATEQUERYTREE procedure takes as input the query tree for the XPath expression, a pointer to the current node in the query tree and data statistics, and returns the selectivity and the cost of the XPath fragment starting from the current node. The selectivity is used mainly in the case that the XPath fragment starting from the current node is part of a predicate. The cost is an estimate of the

Algorithm 2 ANNOTATEQUERYTREE(Q)

Input: Abstract Query Tree Q, Current Node t_i, Data Statistics S, Current set of candidates $\mathcal{P}$
Output: Selectivity at t_i, Cost estimate at t_i, adds candidates to $\mathcal{P}$

1: $minpredsel \leftarrow \infty$
2: $(sumpredcost, sumntwigs, QPcost) \leftarrow (0, 0, 0)$
3: **if** t_i is the end of list symbol **then**
4: return (1,0)
5: **if** t_i is root **then**
6: $sel \leftarrow 1$
7: $card \leftarrow 1$
8: **else if** t_i is a step **then**
9: $sel \leftarrow \min(f(t_i|t_{i-1}), 1.0)$
10: $card \leftarrow card(t_{i-1}) \cdot f(t_i|t_{i-1})$
11: **for all** $p \in Predicates(t_i)$ **do**
12: $(predsel, predcost, predQPcost, ntwigs) \leftarrow$ ANNOTATEQUERYTREEPRED(Q, p, S, $\mathcal{P}$)
13: $sumntwigs \leftarrow sumntwigs + ntwigs$
14: $QPcost \leftarrow \max(QPcost, predQPcost)$
15: $minpredsel \leftarrow \min(minpredsel, predsel)$
16: $sumpredcost \leftarrow sumpredcost + predcost$
17: **if** $sumpredcost > 0$ **then**
18: $cardWithPred \leftarrow card \cdot minpredsel$
19: $QPcost \leftarrow QPcost + sumntwigs * C_{par}$
20: $(rsel, rcost) \leftarrow$ ANNOTATEQUERYTREE(Q, t_{i+1}, S, $\mathcal{P}$)
21: $sel \leftarrow sel \cdot \min(rsel, minpredsel)$
22: $cost \leftarrow f(t_i|t_{i-1}) \cdot [sumpredcost + rcost + C_{step} \cdot f(*|t_i)]$
/* Logic for DP candidacy */
23: **if** $card \geq minCardForDP$ **then**
24: $DPcost \leftarrow$ Eqn. 10
25: **if** $DPcost < cost$ **then**
26: $\mathcal{P} \leftarrow \mathcal{P} \cup t_i$ {/* add t_i as a DP candidate */}
/* Logic for QP candidacy */
27: **if** $sumntwigs > 0$ **then**
28: $QPcost \leftarrow card \cdot [QPcost + rcost + C_{step} \cdot f(*|t_i)]$
29: **if** $QPcost < cost$ **then**
30: $\mathcal{P} \leftarrow \mathcal{P} \cup t_i$ {/* add t_i as QP candidate */}
31: $card(t_i) \leftarrow cardWithPred$
32: return (sel, $cost$)

amount of work required to traverse the XPath fragment starting from the current node using a single thread or processor. Conceptually, the algorithm consists of four main blocks: the base case (line 3), the pre-recursion processing (line 5), the recursive call (line 20), and the post-recursion processing (line 21). The base case of the recursion occurs when all the steps in the XPath expression has been processed and the current node is pointing at the end of list, i.e., beyond the last step (line 3).

In the pre-recursion processing block (line 5), there are two cases: the current node may be a root or a step. If the current node is a root, the contribution to the overall selectivity is always 1.0 and the contribution to the cost is 0. If the current node is a step, the contribution to the selectivity is dependent on the fan-out into the current node, and the contribution to the cost is proportional to the fan-out into the current node multiplied by the cardinality of the previous node. Moreover, one or more predicates may be associated with a step. The block from line 10 to line 17 handles the predicates associated with a step. Each predicate is processed by invoking the ANNOTATEQUERYTREEPRED procedure. The ANNOTATEQUERYTREEPRED procedure returns the selectivity of the predicate, the sequential cost of processing the predicate, the cost of processing the predicate if parallelization via query partitioning is used, and the number of twigs or branches in the predicate expression. Multiple predicates associated with a step are treated as if they are AND'ed together. Hence, the combined query partitioning cost is the maximum of the query partitioning cost of each predicate, the sequential cost of processing all the predicates is simply the sum of the individual predicate costs (line 16), and the resultant selectivity is estimated using the minimum. Once the resultant selectivity has been computed, the cardinality of the current step needs to be adjusted using the selectivity of the predicates (line 18).

Algorithm 3 ANNOTATEQUERYTREEPRED($Q, p_i, S, \mathcal{P}$)

Input: Abstract Query Tree Q, Current Predicate Expression Node p_i, Data Statistics S, Current candidates $\mathcal{P}$
Output: Selectivity sel, Sequential cost $cost$, QP cost $predQPcost$, QP branches $ntwigs$, adds candidates to $\mathcal{P}$

1: **if** p_i is a simple xpath expression **then**
2: $(sel, cost) \leftarrow$ ANNOTATEQUERYTREE(Q, $Expr(p_i)$, $S, \mathcal{P}$)
3: return $(sel, cost, cost, 1)$
4: **else if** p_i is a boolean expression **then**
5: $(lhssel, lhscost, lhsQPcost, lhsntwigs) \leftarrow$ ANNOTATEQUERYTREEPRED(Q, $LeftExpr(p_i)$, $S, \mathcal{P}$)
6: $(rhssel, rhscost, rhsQPcost, rhsntwigs) \leftarrow$ ANNOTATEQUERYTREEPRED(Q, $RightExpr(p_i)$, $S, \mathcal{P}$)
7: **if** boolean operator is AND **then**
8: $sel \leftarrow \min(lhssel, rhssel)$
9: $predQPcost \leftarrow \max(lhsQPcost, rhsQPcost) + (lhssel + rhssel) \cdot ANDoverhead$
10: **else if** boolean operator is OR **then**
11: $sel \leftarrow \max(lhssel, rhssel)$
12: $predQPcost \leftarrow \max(lhsQPcost, rhsQPcost) + (lhssel + rhssel) \cdot ORoverhead$
13: return $(sel, lhscost+rhscost, predQPcost, lhsntwigs+rhsntwigs)$

Line 19 adds the parallelization overhead (a tunable parameter) to the combined query partitioning cost.

In line 20, ANNOTATEQUERYTREE is called recursively on the next step in the XPath query tree. The recursive call returns the selectivity and estimated sequential cost of the XPath fragment starting from the next step.

The post recursion processing starts on line 21. The current node's contribution to the selectivity is multiplied with the selectivity from the recursive call. The current node's contribution to the sequential traversal cost is computed and incorporated with the cost from the recursive call. The procedure then evaluates whether it is feasible to parallelize the processing from the current node using either data partitioning or query partitioning. Finally, the cardinality associated with the current node is updated with the predicate selectivity and stored.

The logic of ANNOTATEQUERYTREEPRED is outlined in Algorithm 3 and we highlight the important details next. The procedure takes as input the XPath query tree, a pointer to a predicate expression node and the data statistics, and returns the selectivity, sequential cost estimate, query partitioning cost estimate and number of twigs of the input predicate expression. If the predicate expression is a simple XPath expression, ANNOTATEQUERYTREE is called to obtain the selectivity and estimated cost. If the predicate expression is a (binary) boolean expression, ANNOTATEQUERYTREEPRED is called recursively on the left and right operands of the boolean expression. In the subsequent post recursion processing, selectivity, sequential cost estimate and query partitioning cost estimate are updated and returned.

5.2 Choosing Candidate Partitioning Points

After the first pass of the optimizer has identified the set $\mathcal{P}$ of candidate partitioning points, the second pass iterates over this set of partitioning points to pick a subset of most 'optimal' partitioning points. Recall that the first pass identifies candidate partitioning points based on local information. Hence, in the second pass, we take into account information that is more 'global' in nature. For example, a candidate data partitioning point (eg. 'c' in /a/b/c/d/e) identified in the first pass does not take into account the cost of processing the query XPath up to the partitioning point (eg. /a/b). We call the query XPath up to the candidate partitioning point p the prefix $prefix(p)$ of p.

The prefix of a partitioning point represents work that needs to be done prior to the partitioning point in question and in full generality, the prefix can also contain other partitioning points. The number of ways that the prefix of a partitioning point could be parallelized is therefore exponential in the number of partitioning points it contains and hence leads to a combinatorial explosion of the search space of all possible parallel plans. We employ a greedy heuristic to deal with this problem: with respect to the partitioing point in question, we view the work associated with the prefix as sequential (not parallelized). Using this assumption, given two partitioning points, the partitioning point of which the prefix requires less traversal of the data tree is likely to result in a more efficient query plan.

The amount of traversal of a prefix can be quantified using the cost models decribed in Section 4. In the case where the prefixes are relatively simple XPaths without descendent axes, a simpler heuristic based on the length of the prefixes can be used. Comparing two candidate partitioning points of the same type (eg. both DP or both QP) becomes very straightforward if the prefixes are simple XPaths: the partitioning point with a shorter prefix results in a better query plan. Note that since we assumed that no parallel processing is done for the prefix, the overall processing time for the entire XPath is limited by the processing for the prefix: no amount of parallelism at the partitioning point can reduce the time required to process the prefix.

EXAMPLE 6. *Consider the XPath '/a/b/c/d[e and f and g]' and two data partitioning points at 'c' and 'd'. The partitioning point 'c' is likely to result in a better plan, because it probably takes less time to process the prefix '/a/b' sequentially than it does to process the prefix '/a/b/c'.*

A similar argument can be made when comparing a data partitioning point and a query partitioning point: the less work the prefix requires the more parallelism is exposed. Given two partitioning points of which the prefixes are the same, the parallel cost (estimated according to Eqn. 11 and Eqn. 12) of processing the XPath fragment starting from the partitioning will be used to distinguish the partitioning points. The parallel cost of a query partitioning point is limited by the number of branches and hence the amount of inherent parallelism. Data partitioning, on the other hand, is less limited, because the inherent limit on parallelism is the cardinality of the partitioning point which for most real data sets and queries is much larger than the number of processors. Hence, when the estimated parallel costs of a data and of a query partitioning point are equal (or very close), the former is prefered.

Using the heuristics described previously, the optimizer sorts the set of candidate partitioning points found in the first pass and picks the top k number of candidate partitioning points. The parameter k can be chosen based on the number of available processors ('cores'). At the time of writing of this paper, the number of cores in most multi-core processors have yet to reach the order of hundreds. Hence, in most cases, picking just one partitioning point is sufficient to produce an efficient parallel query plan. When the number of cores have reached the order of hundreds and beyond, a larger number of partitioning points can be picked. The resultant parallel query plan will contain nested partitioning points, and the processor assignment problem (mapping processors/cores to partitions) will become a significant problem. Discussion of the processor assignment problem is beyond the scope of this paper and we hope to address it as part of our future work.

5.3 Constructing the Parallel Query Plans

Once the top k candidate partitioning points have been chosen by the optimizer, the next step is to construct a parallel execution plan for the input query based on these points. This is done by an

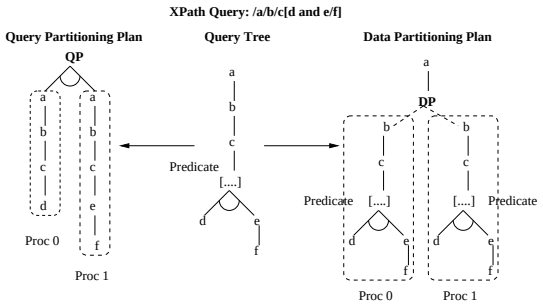

Figure 5: Query and data partitioning plans.

algorithm that accepts as input a query tree Q and the set of partitioning points $\mathcal{P}$, and builds the parallel execution plan iteratively. At each iteration the execution plan is built by considering the partially constructed plan of the previous iteration and by applying the following operations (we omit the pseudo-code of the algorithm, due to lack of space): It picks the next partitioning point $t \in \mathcal{P}$ and locates the position(s) of t in the partially constructed plan from the last iteration (in the first iteration, this partial plan coincides with the query tree Q). Then, if t is a data partitioning node, a new *special* DP node is inserted into the tree in place of t to denote that data partitioning occurs at this point. Then, the algorithm consider the subtree in the plan formerly rooted at t and creates as many duplicates of this sub-tree as the number of processors we are assigning to this partitioning point. All these sub-trees become children of the new QP node. Furthermore, to each instance of node t we add a new subtree corresponding to the predicate that defines the section of the XML document over which the query is to be executed (see Section on Partitioning Strategies).

In the case of query partitioning, we know that by partitioning the query at point t, we essentially rewrite Q into a set of queries whose expressions (and thus trees) differ only after step t. These differing trees of the rewritten queries become the children of a new *special* QP node and the new query tree rooted at QP replaces the partial plan from the previous iteration.

This concludes one iteration of the algorithm and the next partitioning point is considered. Notice that an iteration i might create multiple copies of the partitioning point at iteration $i+1$. Then, the above procedure must be applied to each one of these copies.

As an example of the above procedure, consider Figure 5. In the middle of the figure, we show the query tree corresponding to the query $/a/b/c[d$ and $e/f]$. Assuming that step c is a candidate partitioning point, then to the left of the figure we show the plan in the case that c is a query partitioning point, while to the right of the figure we show the plan in the case of data partitioning. Notice that in the latter plan, each of the subtrees of DP has an additional predicate specifying the section of the XML document over which the query is to be executed. What if an additional data partitioning point exists for step e? Then, for the right branch of the left plan node e is replaced by DP and two new subtrees for e/f are created as children of DP. For the right plan, we need to do a similar change in both subtrees of the (existing) topmost DP node.

6. EXPERIMENTS

We have performed extensive experiments on several types of XPath queries over many XML datasets. In this section, we describe our experiments and present a representative subset (due to space constraints) of our experiment results. The performance of both our optimizer and the parallel query plans are very similar on the other datasets.

Prototype Implementation. We implemented a prototype of our multi-core XPath processing system including our own implementation of an XPath processor, the optimizer, and the parallel query operators. The XPath processor leverages Xerces-C and XALAN DOM APIs. The optimizer is implemented in PERL and leverages the XML::XPath package to parse an XPath into an abstract syntax tree. The output of the optimizer is a (set of) optimal partitioning point(s). We then generate the query plans by hooking up a set of basic query operators implemented in C++. These operators rely on our XPath processor and include a sequential XPath operator, a parallel data partitioning XPath operator, a parallel query partitioning XPath operator, a parallel hybrid query and data partitioning operator, and sequential operators for merging node lists either via intersection or union. The query operators are parametrized by the number of threads.

	XMark-large	XMark-huge	FpML
Size (MB)	116	1172	833
Number of Nodes	514	514	466
Number of Docs	na	na	43,968
Depth	12	12	11

Table 1: Characteristics of the synthetic XMark dataset and the real FpML dataset

Datasets and Queries. We experimented with DBLP, Mondial, Treebank, Swissprot, XMark [25], and an FpML dataset from an investment bank. Due to space constraints, we present representative results from the XMark and FpML dataset. The FpML dataset is a collection of 43,968 real anonymized FpML [1] documents from an unnamed investment bank. Each document follows a proprietary extension of the FpML industry standard schema. We constructed a super root `fpmldocs` in order link the entire collection into a single XML document. In addition to the real FpML data, we also used XPath expressions extracted from a real query workload provided by the investment bank. Table 2 lists the representative XPath queries over the two XML datasets. Two of the XMark queries were drawn from the XPathMark [10] benchmark. The query XM1 was run against the XMark-huge document. The remaining XMark queries were run on the XMark-large document. The experiments were performed on a dual quad-core 2.66 GHz Intel Xeon system running Linux.

Methodology. For each dataset and each query in the testing set for that dataset, we use our optimizer to identify the candidate partitioning points and strategies. Each of these partitioning point and strategy corresponds to a prebuilt parametrized query plan in our prototype. These query plans are executed on the dataset in question by setting the parameters with the appropriate XPath query fragments, the partitioning point and the number of processors to use. In general, we run the query plans over different number of processors and record the wall-clock execution time. We then check if the query plan (i.e., partitioning point and strategy) chosen by the optimizer is the among the fastest running query plan.

6.1 Evaluation Results

The query XM1 (Table 2) consists of a series of 6 child steps. As the query does not contain any predicates, this query can be parallelized only via the data partitioning approach. Figure 6 presents relative performance of five possible data partitioning query plans, **dp1** to **dp5** for the query XM1. These plans differ in the way the original query is partitioned into sequential and parallel queries. The query plan, **dp1**, uses the earliest partitioning point, the child step `/closed_auctions`, to partition the original query, while the query plan, **dp5**, chooses the later child

Document	Key	XPath Query
XMark.xml	XM1	`/site/closed_auctions/closed_auction/annotation/description/text/keyword`
	XM2	`/site/people/person[profile/gender and profile/age]/name`
	XM3	`/site/regions/asia/item[mailbox/mail/date and description/parlist/listitem/text and payment]/name`
	XM4	`/site/open_auctions/open_auction[annotation/author and annotation/description and bidder/date and privacy]`
FPML.xml	FM1	`/fpmldocs/Message/FpML/trade/creditDefaultSwap/generalTerms /effectiveDate/unadjustedDate`
	FM2	`/fpmldocs/Message/FpML[party/partyName and trade/creditDefaultSwap/generalTerms /effectiveDate/unadjustedDate]/trade/tradeHeader/tradeDate`

Table 2: XPath Queries used for Experimental Evaluation

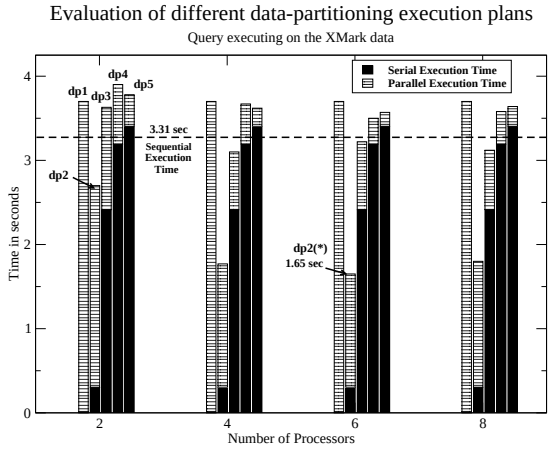

Figure 6: Impact of query splitting point on performance of data partitioned query plan. The graph compares performance of 5 different data partitioning query plans for the query XM1. We use the convention of annotating the plan chosen by the optimizer using (*).

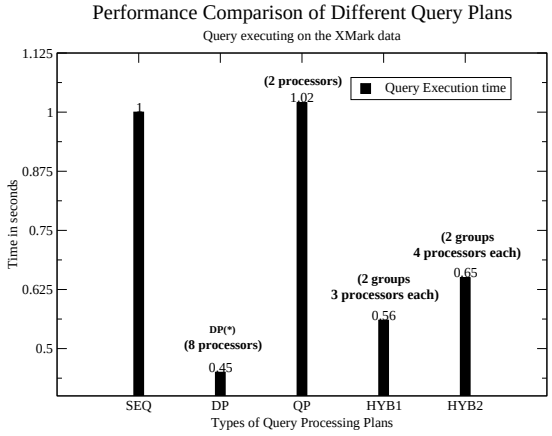

Figure 7: Comparing performance of data-, query- and hybrid partitioning plans for the query XM2.

step, `/text`, to partition the query. Our optimizer identified two plans, **dp2**, and **dp3**, that use the child steps, `/closed_auction` and `/annotation`, respectively, to partition the original query. Based on the recommendation, our system's heuristic (Section 5.2 finally chose the query plan, **dp2**, as the ideal plan for executing the query XM1. Note that the among the five evaluated plans, only **dp2** consistently performs better than the sequential execution, which requires 3.31 seconds (Figure 6). As Figure 6 illustrates, performance of query XM1 under the plan **dp2** is dominated by the parallel execution time which improves as the number of processors is increased. The best query performance is observed when the query is executed using **dp2** on 6 processors (1.65 seconds). For the plans **dp2** to **dp5**, the overall performance is dominated by the sequential execution costs. Hence, although the parallel execution costs improve as the number of processors is increased, the overall performance degrades. It is important to note that aggressive parallelization as implemented in the plan **dp1** performs poorly since each participating processor ends up doing the amount of work similar to the purely sequential execution (as there is only one `/closed_auctions` node and it gets replicated on all participating processors). In such setting, the amount of available processing power is inconsequential as demonstrated in Figure 6. The important lesson from this experiment is that the best execution plans result in from finding the proper mix of sequential and parallel execution.

Figure 7 presents performance of executing the query XM2 using different parallel execution plans. XM2 is a predicated query with a conjunction of two path predicates. The optimizer first evaluates whether the query should be parallelized or not. For XM2, the optimizer determined that the query would benefit from parallelization. It further evaluated the two different parallelization strategies: data- and query partitioning over different partitioning points in the original query. The optimizer recommended to first use the data partitioning strategy at the partitioning point, `/person`, followed by the query partitioning strategy by partitioning the query predicates at the partitioning point, `/person`, over 2 processors. In the query partitioning execution, the original query is rewritten into two sub-queries, each executing a distinct path predicate on the `/person` node. These two sub-queries are then executed on two different processors and their local results are intersected to compute the final result. We first evaluated these two options and observed that the query partitioning plan performs (1.02 seconds) as bad as the sequential execution (1 second). Still, as the optimizer chose the query partitioning as the second option, we decided to use the hybrid approach to provide the query partitioning plan more parallelism. The hybrid partitioning scheme first uses the query partitioning scheme to partition the original query and then follows the data partitioning strategy to execute individual sub-queries on two processor groups of 3 (HYB1) and 4 processors (HYB2) each. As Figure 7 illustrates neither HYB1 and HYB2 plans can match the performance of the data partitioning plan selected by our optimizer.

Figures 8 and 9 present the performance of different query execution plans for the queries XM3 and XM4, respectively. For the query XM3, the optimizer selects the data partitioning at the partitioning point `/item` to be the optimal execution plan, followed by a query partitioning plan at the same partitioning point by partitioning the predicates over 3 processors. Similar to the query XM1, the query execution plan performs (0.15 seconds) as bad as the sequential plan (0.16 seconds). We also tried applying the hybrid strategy to the query execution plan, where each sub-query was ex-

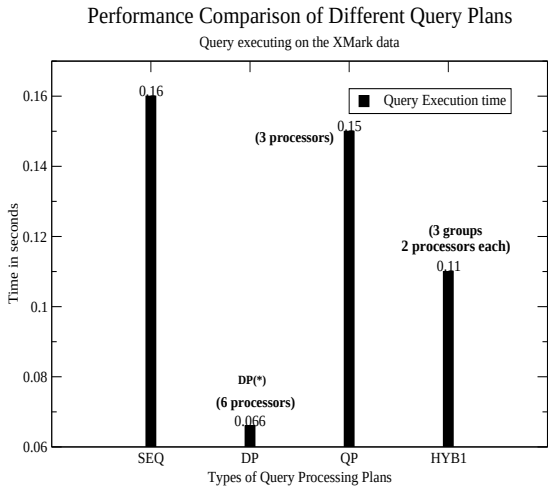

Figure 8: Comparing performance of data-, query- and hybrid partitioning plans for the query XM3.

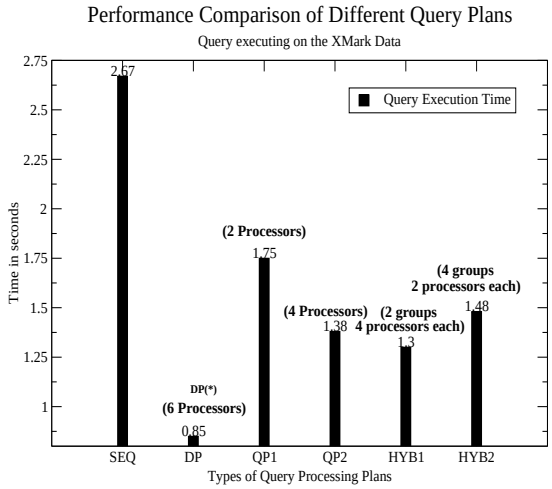

Figure 9: Comparing performance of data-, query- and hybrid partitioning plans for the query XM4.

ecuted over 2 processors. However, the hybrid approach was not able to provide significant improvement over the query execution plan (0.11 seconds) and the best performance was observed by the data partitioning plan chosen by the optimizer (0.066 seconds).

For the query XM4, we observe similar behavior. Our optimizer recommended the data partitioning plan at the partitioning point, `/open_auction`, as the top plan. It also suggested two query partitioning plans at the same partitioning point: the first plan creates two predicated sub-queries, first as a conjunction of two predicates, `annotation/author` and `annotation/description`, and the second as a conjunction of remaining predicates, `bidder/date` and `privacy`, the second plan creates four predicated sub-queries, each with a separate predicate. We ran the first query partitioned plan on 2 processors and other on 4 processors. However, both performed worse (1.75 seconds and 1.38 seconds) than the data partitioned plan (0.85 seconds using 6 processors). Application of hybrid partitioning strategy didn't help in improving the performance. The best performance was provided by the data partitioning strategy using 6 processors.

Figure 10 presents the comparison of three different data partitioning queries for the query FM1 on the FPML dataset. Our optimizer suggested only one possible partitioning point, `/Message`, to generate the serial and parallel sub-queries (**dp1**). To evaluate the efficacy of the suggested plan, we compared it with two different data partitioning plans, **dp2** and **dp3**, with partitioning points at `/FpML` and `/trade`, respectively. As illustrated in the Figure 10, the plan suggested by the optimizer provided the best performance as it had the smallest serial component.

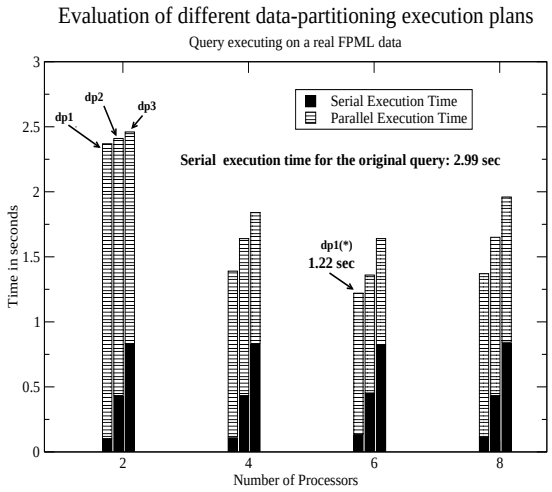

Figure 10: Impact of query splitting point on the performance of a data partitioned query plan. The graph compares performance of 3 different data partitioning query plans for the query FM1.

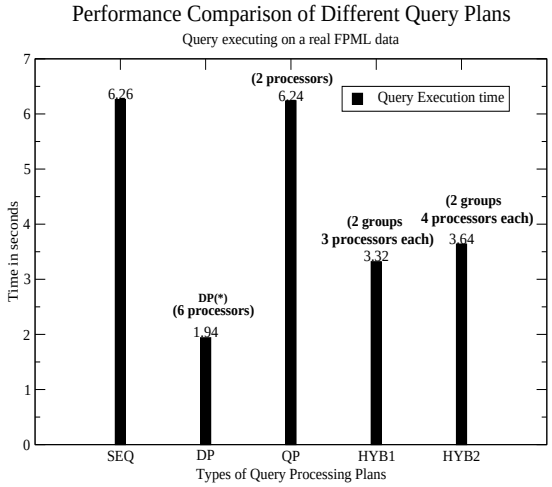

Figure 11: Comparing performance of data-, query- and hybrid partitioning plans for the query FM2.

Finally, Figure 11 presents comparison of multiple query plans for the Query FM2. The optimizer suggested two possible plans: data partitioning at the partition point, `FpML`, and query partitioning over 2 processors by creating two sub-queries for two different predicates. Similar to previous cases with predicated queries, the query partitioning plan (6.24 seconds) performed as bad as the sequential execution (6.26 seconds), and the hybrid partitioning was also ineffective. The best performance (1.94 seconds) was produced by the data partitioning plan recommended by our optimizer.

Our experiments have conclusively demonstrated that our optimizer precisely predicated the XPath execution costs and consistently recommended the optimal plan. In all cases, the query partitioning plan performed the worst. For predicated queries, query

partitioning requires the participating processors to traverse the entire tree. If the work involved in executing the predicates is not dominant, then individual processors perform the same amount of work as the sequential plan, and further their traversal patterns overlap. Hence, their execution times match the sequential counterpart (e.g., Figures 7, 8, and 11). Therefore, the query partitioning strategy that partitions the predicate work is suitable to only those queries whose overall work is dominated by the predicate execution.

7. CONCLUSION

Motivated by the recent trends in hardware and the emergence of multi-core processors in commodity systems, this paper presented the first systematic investigation towards parallelizing XPath queries. We considered alternative strategies of XPath parallelization and presented a series of cost functions to estimate the processing costs of different phases in the parallalization process that use these strategies. These cost functions take into account both the data statistics and query specifics, and form the basis of our parallel optimizer. For our optimizer, we used a number of heuristics that consider a subset of the exponentially large search space of possible parallelization plans. Our heuristics identified the most promising plans in this subset (with the help of the cost functions), which were then used to generate parallel execution plans. We have implemented a prototype end-to-end parallel XPath processing system that incorporates all the aforementioned functionalities. We presented a set of experiments using realistic XPath workloads that demonstrated the efficacy of our techniques in identifying optimal parallel execution plans. We believe that this work is only a start to the exploration of XPath parallelization and many future research topics exist, including, support for a wider fragment of XPath and the problem of choosing the optimal processor assignment.

8. REFERENCES

[1] Financial products markup language. http://www.fpml.org.

[2] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB 2001*, pages 591–600, 2001.

[3] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. E. Simmen, M. Wang, and C. Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[4] A. Balmin, F. Özcan, A. Singh, and E. Ting. Grouping and optimization of XPath expressions in DB2®pureXML. In *SIGMOD*, pages 1065–1074, New York, NY, USA, 2008. ACM.

[5] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of XPath Queries using Multi-core Processors: Challenges and Experiences. In *12th International Conference on Extending Database Technology*, 2009.

[6] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using Partial Evaluation in Distributed Query Evaluation. In *VLDB*, pages 211–222, 2006.

[7] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting twig matches in a tree. In *ICDE 2001*, pages 595–604, 2001.

[8] G. Cong, W. Fan, and A. Kementsietsidis. Distributed Query Evaluation with Performance Guarantees. In *SIGMOD Conference*, pages 509–520, 2007.

[9] W. W. W. Consortium. XML Path Language (XPath) 2.0, W3C Recommendation, 23 January 2007. www.w3.org.

[10] M. Franceschet. XPathMark: an XPath benchmark for XMark generated data. In *International XML Database Symposium*, 2005.

[11] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML count. In *SIGMOD 2002*, pages 181–191, 2002.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[13] S. Hidaka, H. Kato, and M. Yoshikawa. A relative cost model for XQuery. In *SAC*, pages 1332–1333, New York, NY, USA, 2007. ACM.

[14] M. F. Khan, R. A. Paul, I. Ahmad, and A. Ghafoor. Intensive data management in parallel systems: A survey. *Distributed and Parallel Databases*, 7(4):383–414, October 1999.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[16] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB 2002*, 2002.

[17] L. Lim, M. Wang, and J. S. Vitter. CXHist : An on-line classification-based histogram for XML string selectivity estimation. In *VLDB*, pages 1187–1198, 2005.

[18] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

[19] W. Lu, K. Chiu, and Y. Pan. A Parallel Approach to XML Parsing. In *Grid 2006: The 7th IEEE/ACM International Conference on Grid Computing*, pages 28–29, 2006.

[20] W. Lu and D. Gannon. Parallel XML Processing by Work Stealing. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38, 2007.

[21] Q. Luo and K. A. Ross, editors. *4th Workshop on Data Management on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008*. ACM, 2008.

[22] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *CCGRID*, pages 351–362, 2007.

[23] N. Polyzotis and M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD 2002*, pages 358–369, 2002.

[24] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB 2002*, pages 466–477, 2002.

[25] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The xml benchmark project. Technical report, Amsterdam, The Netherlands, The Netherlands, 2001.

[26] D. Suciu. Distributed Query Evaluation on Semistructured Data. *ACM Trans. Database Syst.*, 27(1):1–62, 2002.

[27] J. Teubner, T. Grust, S. Maneth, and S. Sakr. Dependable cardinality forecasts for xquery. *Proc. VLDB Endow.*, 1(1):463–477, 2008.

[28] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB 2004*, pages 240–251, 2004.

[29] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *EDBT 2002*, pages 590–608, 2002.

[30] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing xml queries. In *VLDB*, pages 289–300, 2005.