Adaptive Processing of Top-k Queries in XML

Amélie Marian
Columbia University
amelie@cs.columbia.edu

Sihem Amer-Yahia
AT&T Labs–Research
sihem@research.att.com

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

The ability to compute top-k matches to XML queries is gaining importance due to the increasing number of large XML repositories. The efficiency of top-k query evaluation relies on using scores to prune irrelevant answers as early as possible in the evaluation process. In this context, evaluating the same query plan for all answers might be too rigid because, at any time in the evaluation, answers have gone through the same number and sequence of operations*, which limits the speed at which scores grow. Therefore,* adaptive query processing *that permits different plans for different partial matches and* maximizes the best scores *is more appropriate. In this paper, we propose an architecture and adaptive algorithms for efficiently computing top-k matches to XML queries. Our techniques can be used to evaluate both* exact *and* approximate *matches where approximation is defined by* relaxing *XPath axes. In order to compute the scores of query answers, we extend the traditional* tf*idf *measure to account for document structure. We conduct extensive experiments on a variety of benchmark data and queries, and demonstrate the usefulness of the adaptive approach for computing top-k queries in XML.*

1. Introduction

The ability to compute top-k answers to XML queries is gaining importance due to the increasing number of large XML repositories[1]. Top-k query evaluation on exact answers is appropriate when the answer set is large and users are only interested in the highest-quality matches. Top-k queries on approximate answers are appropriate on *structurally heterogeneous* data (e.g., querying books from different online sellers). In both cases, an XPath query may have a large number of answers, and returning *all* answers to the user may not be desirable. One of the prominent querying approaches in this case is the top-k approach that limits the cardinality of answers by returning k answers with the highest scores.

The efficiency of top-k query evaluation relies on using intermediate answer scores in order to prune irrelevant matches as early as possible in the evaluation process. In this context, evaluating the same execution plan for all matches leads to a *lock-step* style processing which might be too rigid for efficient query processing. At any time in the evaluation, answers have gone through *exactly the same number and sequence of operations*, which limits how fast the scores of the best answers can grow. Therefore, adaptive query processing that permits different partial matches to go through different plans is more appropriate. Adaptivity in query processing has been utilized before [1, 4, 12, 25] in order to cope with the unavailability of data sources and varying data arrival rates, by reordering joins in a query plan. In this paper, we study adaptive techniques for efficiently computing exact and approximate answers to top-k queries in XML.

In order to compute approximate matches of XPath queries, we adopt the *query relaxation* framework defined in [3] where relaxations such as the ones proposed in [2, 11, 23] can be encoded in the query plan in order to permit structurally heterogeneous answers to match the original query in addition to exact answers.

Choosing the best k query matches is based on computing answer scores. Scoring query answers in the context of XML needs to account for two key aspects: (i) an answer to an XPath query may be any fragment of the input document and, (ii) an XPath query consists of several predicates linking the returned node to other query nodes, instead of simply "keyword containment in the document". Existing efforts in Information Retrieval (IR) such as [15, 24] have focused on extending the *tf*idf* (term frequency and inverse document frequency) measure to return document fragments. In our work, we extend the *tf*idf* measure to account for scoring on both structure and content predicates and return document fragments.

We make the following contributions:

- We present *Whirlpool*, a novel architecture incorporating a family of algorithms for processing top-k queries on XML documents adaptively.

  Whirlpool is used to compute both exact and approximate matches. It is adaptive in permitting partial matches to the same query to follow different execution plans, taking the top-k nature of our problem into account. The key features of *Whirlpool* are: (a) a partial match that is highly likely to end up in the top-k set is processed in a prioritized manner, and (b) a partial match unlikely to be in the top-k set follows the cheapest plan that enables its early pruning.

1 Library of Congress: http://lcweb.loc.gov/crsinfo/xml/
INEX: http://www.is.informatik.uni-duisburg.de/projects/inex03/

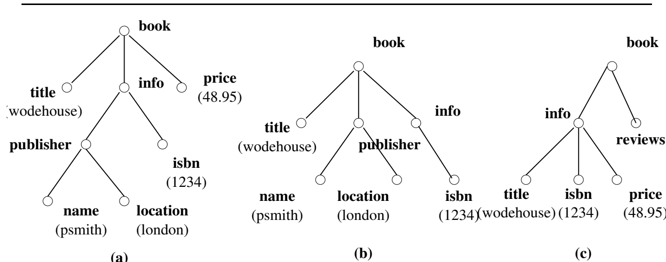

Figure 1. Heterogeneous XML Database Example

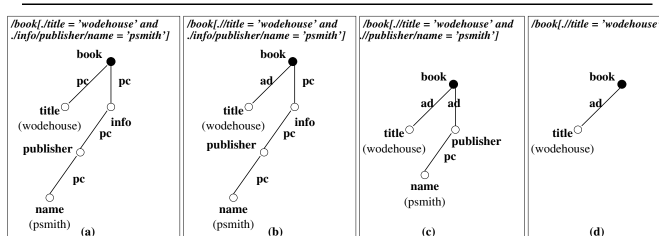

Figure 2. Query Tree Patterns and Relaxations

- We propose a novel scoring function for XML, inspired by *tf*idf*.
- We instantiate the *Whirlpool* strategy for a variety of routing alternatives (i.e., which operation does a partial match go through next?), and prioritization alternatives (i.e., given a number of partial matches waiting for a specific operation, how do we prioritize them?), to obtain a family of adaptive evaluation algorithms.
- We describe a real prototype, in which we implemented the *Whirlpool* architecture and algorithms. We perform a detailed experimental evaluation of our algorithms on a variety of benchmark data sets and queries, and (i) identify the tradeoffs between the different routing and prioritization alternatives among the *Whirlpool* algorithms, (ii) demonstrate that adaptivity pays off in processing top-k queries, and (iii) validate our scoring function.

To the best of our knowledge, this is the first work that explores per-answer adaptive evaluation strategies for computing top-k answers to XML queries.

The paper is organized as follows. Section 2 contains a motivating example for relaxation and adaptivity. Section 3 contains a summary of related work. Section 4 presents our scoring function. Section 5 describes the *Whirlpool* architecture and algorithms. Extensive experiments are given in Section 6. Finally, we conclude in Section 7.

2. Motivating Example

Relaxation: We consider a data model for XML where information is represented as a forest of node labeled trees. A simple database instance, containing a heterogeneous collection of books, is given in Figure 1. We represent XML queries as *tree patterns*, an expressive subset of XPath. Figure 2 contains examples of XPath queries and their corresponding tree pattern. A tree pattern is a rooted tree where nodes are labeled by element tags, leaf nodes are labeled by tags and values and edges are XPath axes (e.g., pc for parent-child, ad for ancestor-descendant). The root of the tree (shown in a solid circle) represents the returned node.

Different queries would match different books in Figure 1. For example, the query in Figure 2(a) would match book 1(a) exactly, but would neither match book 1(b) (since `publisher` is not a child of `info`) nor book 1(c) (since the `title` is a descendant, not a child, of `book`, and the `publisher` information is entirely missing). However, intuitively, it makes sense to return all three books as candidate matches, suitably ranked based on the extent of similarity of the books to the query in Figure 2(a).

In order to allow for such approximate answers, we adopt query relaxation as defined in [2, 11, 23] and formalized in [3]. We use three specific relaxations (or any composition of these relaxations): *edge generalization* (replacing a pc edge with ad), *leaf deletion* (making a leaf node optional) and *subtree promotion* (moving a subtree from its parent node to its grand-parent). These relaxations capture approximate answers but still guarantee that exact matches to the original query continue to be matches to the relaxed query. For example, the query in Figure 2(b) can be obtained from the query 2(a) by applying edge generalization to the edge between `book` and `title`. The query in Figure 2(c) is obtained from query 2(a) by composing subtree promotion (applied to the subtree rooted at `publisher`) followed by leaf deletion applied to `info`, and edge generalization applied to the edge between `book` and `title`. Finally, query 2(d) is obtained from query 2(c) by applying leaf deletion on `name` then on `publisher`.

As a result, while queries 2(a) and 2(b) match the book in Figure 1(a) only, query 2(c) matches both book 1(a) and book 1(b) and query 2(d) matches all three books.

Exact matches to a relaxed query are the desired approximate answers to the original user query. In order to distinguish between different answers, we need to compute scores that account for query relaxation. We focus on this issue in Section 4. For now, we assume that scores are given and motivate the need for adaptive query processing.

Adaptivity: Suppose now that we are interested in evaluating a query that looks for the top-1 book with a `title`, a `location` and a `price`, all as children elements. Obviously, without applying query relaxation, this query would be empty if it is evaluated on our three books in Figure 1. Similarly to [2], we use a left-deep outer-join plan that encodes edge generalizations and subtree promotions in the query. Leaf nodes in the plan are query nodes and join predicates correspond to pc and ad edges in the query tree. Let us assume that we evaluate the query on our book collection enhanced with an additional book (d) having three exact matches for `title`, each one of them with a score equal to 0.3, five approximate matches for `location` where approximate scores are 0.3, 0.2, 0.1, 0.1, and 0.1, and one exact match for `price` with score 0.2. These scores are associated with the corresponding join predicate. Each book

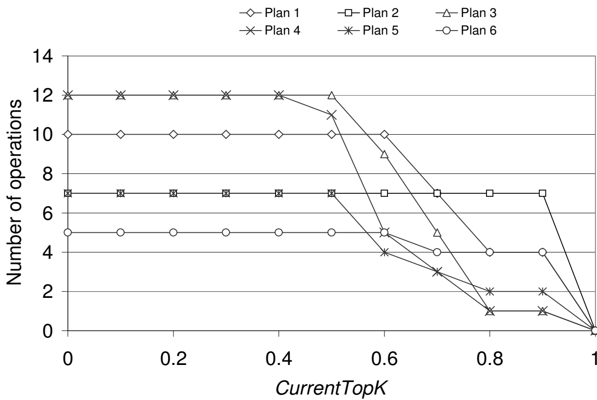

Figure 3. Adaptivity Example.

generates multiple tuples in the join plan (one for each combination of `book`, `title`, `location` and `price`). Thus, the score of a tuple is the sum of the individual predicate scores.

For simplicity, we focus only on the computation of tuples for book 1(d). During the evaluation of book (d), some tuples may be pruned based on their scores and the score of the current k^{th} best answer ($currentTopK$). This value depends on the values of previously computed tuples. Therefore, the number of pruned tuples at each step depends on previously computed tuples.

We consider six join plans that correspond to all permutations of `title`, `location` and `price` assuming that the root node `book`, is always evaluated first. Figure 3 shows the performance of each plan with increasing values of $currentTopK$. The performance is measured in terms of the total number of join operations (i.e., join predicate comparisons). The figure shows that no plan is the best (even with a small number of generated tuples – 15 tuples in this example). When $currentTopK < 0.6$, the best plan is Plan 6 (join `book` with `price` then with `title` then with `location`). However, when $0.6 \leq currentTopK \leq 0.7$, the best plan is Plan 5 (join `book` with `price` then `location` then `title`). Finally, when $currentTopK > 0.7$, Plans 4 (join `book` with `location` then `price` then `title`) and 3 (join `book` with `location` then `title` then `price`) are both best. Interestingly, Plans 3 and 4 are by far the worst if $currentTopK \leq 0.5$, but become the best later on, and Plan 6 becomes bad for higher values of $currentTopK$. Intuitively, joining `book` with `location` first creates the largest number of intermediate tuples (5), which is why Plans 3 and 4 are bad for low values of $currentTopK$. However, since `location` has only approximate matches, when $currentTopK$ is high, the tuples generated from the join with `location` can be pruned faster, leading to fewer alive intermediate tuples.

Since the value of $currentTopK$ changes during query evaluation, static join ordering (akin to selectivity-based optimization) would not be optimal. Query evaluation should *dynamically* decide which join predicate to consider next for a given tuple based on the value of $currentTopK$ using adaptive query processing.

3. Related Work

Several query evaluation strategies have been proposed for XPath. Prominent among them are approaches that extend binary join plans, and rely on a combination of index retrieval and join algorithms using specific structural (XPath axes) predicates [19]. In this paper, we adopt a similar approach for computing exact query answers.

Several query relaxation strategies have been proposed before. In the context of graphs, Kanza and Sagiv [18] proposed mapping query paths to database paths, so long as the database path includes all the labels of the query path; the inclusion need not be contiguous or in the same order which bears some similarities to edge generalization with subtree promotion. Rewriting strategies [9, 11, 15, 23] enumerate possible queries derived by transformation of the initial query. Data-relaxation [10] computes a closure of the document graph by inserting shortcut edges between each pair of nodes in the same path and evaluating queries on this closure. Plan-relaxation [2] encodes relaxations in a single binary join plan (the same as the one used for exact query evaluation). This encoding relies on (i) using outer-joins instead of inner-joins in the plan (e.g., to encode leaf deletion), and (ii) using an ordered list of predicates (e.g., if not child, then descendant) to be checked, instead of checking just a single predicate, at each outer-join. Outer-join plans were shown to be more efficient than rewriting-based ones (even when multi-query evaluation techniques were used), due to the exponential number of relaxed queries [2, 3]. In this paper, we use outer-join plans for computing approximate matches.

In relational databases, existing work has focused on extending the evaluation of SQL queries for top-k processing. None of these works follows an adaptive query evaluation strategy. Carey and Kossmann [6] optimize top-k queries when the scoring is done through a traditional SQL order-by clause, by limiting the cardinality of intermediate resuls. Other works [5, 8, 16] use statistical information to map top-k queries into selection predicates which may require restarting query evaluation when the number of answers is less than k.

Over multiple repositories in a mediator setting, Fagin et al. propose a family of algorithms [13, 14], which can evaluate top-k queries that involve several independent "subsystems," each producing scores that are combined using arbitrary monotonic aggregation functions. These algorithms are sequential in that they completely "process" one tuple before moving to the next tuple.

The Upper [20] and MPro [7] algorithms show that interleaving probes on tuples results in substantial savings in execution time. In addition, Upper [20] uses an adaptive per-tuple probe scheduling strategy, which results in additional savings in execution time when probing time dominates query execution time. These techniques differ from our approach in that all information on a tuple is retrieved through a unique tuple ID, whereas our operations are outer-joins that spawn one or more result tuples. Chang and Hwang [7] suggested an extension to MPro that evaluates joins as cartesian products, thus requiring to process a potentially huge

number of tuples. In contrast, our model allows for evaluation of all results of a join at once.

Top-k query evaluation algorithms over arbitrary joins have been presented for multimedia applications [21] and relational databases [17] but their ranking function combines individual tuple scores, whereas, in our scenario, the score of a top-k answer depends on the join predicate (e.g., child or descendant) used to produce the XPath approximate match (Section 4). Thus, a given node participates differently to the final score of the approximate answers it is joined with, depending on how good a match it is. In addition, existing top-k join algorithms require join inputs to be sorted, which is not the case in our setup.

Recently [19], top-k keyword queries for XML have been studied via proposals extending the work of Fagin et al., [13, 14] to deal with a bag of single path queries. Adaptivity and approximation of XML queries are not addressed in this work. Finally, in [2], the goal was to identify all answers whose score exceeds a certain threshold (instead of top-k answers). Early pruning was performed using branch-and-bound techniques. The authors explored a lock-step adaptive processing for relaxed XML queries while the present work explores adaptivity on a per-answer basis.

While our idea of adaptive evaluation is similar to [4], we use adaptivity in the context of exact and approximate XML queries and focus on issues such as exploring different routing strategies (Section 6) that are appropriate when pruning intermediate query answers for top-k evaluation.

4. Scoring Function

The traditional *tf*idf* function is defined in IR, on keyword queries against a document collection. This function takes into account two factors: (i) *idf*, or inverse document frequency, quantifies the relative importance of an individual keyword (i.e., query component) in the collection of documents (i.e., candidate answers); and (ii) *tf*, or term frequency, quantifies the relative importance of a keyword (i.e., query component) in an individual document (i.e., candidate answer). In the vector space model in IR [22], query keywords are assumed to be independent of each other, and the *tf*idf* contribution of each keyword is added to compute the final score of the answer document.

In this section, we present a conservative extension of the *tf*idf* function to XPath queries against XML documents. The first point to note is that, unlike traditional IR, an answer to an XPath query need not be an entire document, but can be any node in a document. The second point is that an XPath query consists of several predicates linking the returned node to other query nodes, instead of simply "keyword containment in the document" (as in IR). Thus, the XML analogs of *idf* and *tf* would need to take these two salient points into consideration.

Existing efforts in IR [15, 24] have focused on extending *tf*idf* to return document fragments (instead of whole documents). In [24], the authors consider the use of semantic ontologies to compute scores on content predicates. In our work, we focus on a scoring method that combines predicates on both structure and content. Considering ontologies on content is beyond the scope of this paper.

Definition 4.1 [XPath Component Predicates] Consider an XPath query Q, with q_0 denoting the query answer node, and $q_i, 1 \leq i \leq \ell$, denoting the other query nodes. Let $p(q_0, q_i)$ denote the XPath axis between query nodes q_0 and $q_i, i \geq 1$. Then, the *component predicates* of Q, denoted $\mathcal{P}_Q$, is the set of predicates $\{p(q_0, q_i)\}, 1 \leq i \leq \ell$. ∎

For example, the component predicates of the XPath query `/a[./b and ./c[.//d and following-sibling::e]]` is the set $\{$`a[parent::doc-root]`, `a[./b]`, `a[./c]`, `a[.//d]`, `a[./e]`$\}$. The component predicates provide a unique decomposition of the query into a set of "atomic predicates". This is akin to decomposing a keyword query in IR into a set of individual "keyword containment predicates".

Definition 4.2 [XML idf] Given an XPath query component predicate $p(q_0, q_i)$, and an XML database D, p's *idf* against D, $idf(p(q_0, q_i), D)$, is given by:

$$\log\left(\frac{\mid \{n \in D : tag(n) = q_0\} \mid}{\mid \{n \in D : tag(n) = q_0 \& (\exists n' \in D : tag(n') = q_i \& p(n, n'))\} \mid}\right)$$

∎

Intuitively, the *idf* of an XPath component predicate quantifies the extent to which q_0 nodes in the database D additionally satisfy $p(q_0, q_i)$. The fewer q_0 nodes that satisfy predicate $p(q_0, q_i)$, the larger is the *idf* of $p(q_0, q_i)$. This is akin to the case in IR: the fewer the documents that contain keyword k_i, the larger is k_i's *idf*.

Definition 4.3 [XML tf] Given an XPath query component predicate $p(q_0, q_i)$, and a node $n \in D$ with tag q_0, p's *tf* against node n, $tf(p(q_0, q_i), n)$, is given by:

$$\mid \{n' \in D : tag(n') = q_i \& p(n, n')\} \mid$$

∎

Intuitively, the *tf* of an XPath component predicate p against a candidate answer $n \in D$ quantifies the number of distinct ways in which n satisfies predicate p. This is again akin to the case in IR: the more the number of occurrences of keyword k_i in a document d_j, the larger is the term frequency of k_i in d_j.

Definition 4.4 [XML tf*idf Score] Given an XPath query Q, let $\mathcal{P}_Q$ denote Q's set of component predicates. Given an XML database D, let $\mathcal{N}$ denote the set of nodes in D that are answers to Q. Then the score of answer $n \in \mathcal{N}$ is given by:

$$\Sigma_{p_i \in \mathcal{P}_Q}(idf(p_i, D) * tf(p_i, n))$$

∎

Note that, in defining the *tf*idf* score of an XPath query answer, we closely followed the vector space model of IR in assuming independence of the query component predicates. A *key* advantage of this approach, as we shall see later, is the ability to compute this score in an incremental fashion during query evaluation. More sophisticated (and complex)

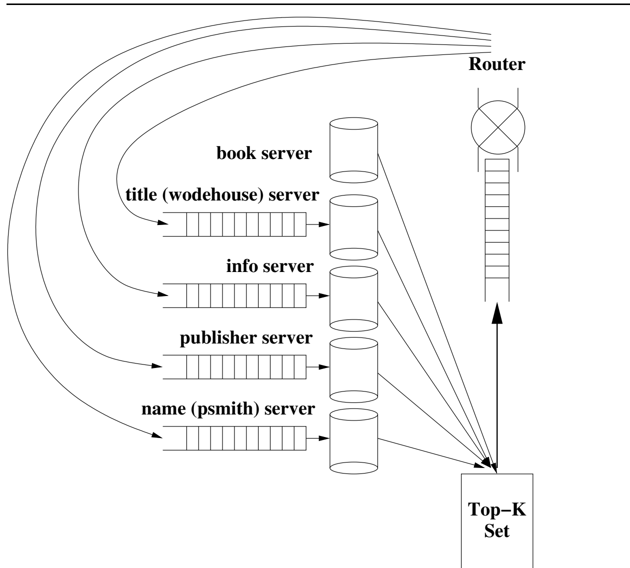

Figure 4. The *Whirlpool* Architecture

scores are possible if the independence assumption is relaxed, as in probabilistic IR models [22]. We do not pursue that avenue further in this paper.

As defined, different *exact* answers to an XPath query may also end up with different scores. This is no different from the IR case of having different documents that contain each of the query keywords having different scores. Once XPath query relaxations are permitted, an approximate answer to the original query Q is simply an exact answer to a relaxed query Q' of Q. Thus, our *tf*idf* mechanism suffices to score approximate answers to Q as well.

5. Whirlpool

We first describe the overall *Whirlpool* architecture and then present our adaptive top-k processing algorithms.

5.1. Architecture

Intuitively, the *Whirlpool* approach is an evaluation strategy of *controlled chaos*, which is extremely effective in cheaply and quickly identifying the top-k answers to relaxed XPath queries. The "chaos" is a consequence of permitting the possibility of different evaluation plans for different partial matches; this is in sharp contrast to the lock-step approach, where each partial match goes through the *same* sequence of operations. The "control" comes from making cost-based decisions, instead of choosing random evaluation plans.

The key components of the *Whirlpool* architecture are depicted in Figure 4, specialized for the XPath query in Figure 2(a), and its relaxations. These include servers, server queues, the top-k set, the router and router queue.

Servers and Server Queues. At the heart of adaptive query evaluation are servers, one for each node in the XPath tree pattern. Figure 4 depicts five servers, labeled with the query node labels.

One of these servers, the `book` server, is special in that it generates candidate matches to the root of the XPath query, which initializes the set of partial matches that are adaptively routed through the system.

Each of the other servers, e.g., the `publisher` server, maintains a priority queue of partial matches (none of which have previously gone through this server). For each partial match at the head of its priority queue, it (i) computes a set of extended (partial or complete) matches, each of which extends the partial match with a `publisher` node (if any) that is consistent with the structure of the queries, (ii) computes scores for each of the extended matches, (iii) determines if the extended match influences or is influenced by the top-k set.

Top-k Set. The system maintains a candidate set of top-k (partial or complete) matches, along with their scores, as the basis for determining if a newly computed partial match, (i) updates the score of an existing match in the set, or (ii) replaces an existing match in the set, or (iii) is pruned, and hence not considered further. Note that only one match with a given root node is present in the top-k set as the k returned answers must be distinct instantiations of the query root node. Matches that are complete are not processed further, whereas partial matches that are not pruned are sent to the router.

Router and Router Queue. The matches generated from each of the servers, and not pruned after comparison with the top-k set, are sent to the router, which maintains a queue based on the maximum possible final scores of the partial matches (Section 6.1.3) over its input. For the partial match at the head of its queue, the router makes the determination of the next server that needs to process the partial match, and sends the partial match to the queue of that server.

The top-k answers to the XPath query, along with their scores, are known when there are no more partial matches in any of the server queues, the router queue, or being compared against the top-k set.

5.2. Algorithms

We first describe how each server processes its input and then, we explain the overall top-k query processing.

5.2.1. Server Query Processing. Each server handles two distinct sources of complexity:

Query relaxations: A consequence of permitting XPath query relaxations is that the predicates at a server can involve a variety of nodes.

For example, given the query in Figure 2(a), and its *Whirlpool* architecture of Figure 4, the server corresponding to `publisher` needs to check predicates of the form $pc(\texttt{info}, \texttt{publisher})$ and $pc(\texttt{publisher}, \texttt{name})$ for the exact query. Supporting edge generalization on the edges

Algorithm 1 Server Predicates Generation

```
Require: Query Q, Query Node n {n is current server node}
 1: Relaxation_with_rootNode = getComposition(n, rootNode(Q));
 2: structuralPredicate = Relaxation_with_rootNode;
 3: for each Node n' in Q do
 4:   {isDescendant(a,b) evaluates to true if a descendant of b};
 5:   if (isDescendant(n',n)) then
 6:     Relaxation_with_serverNode = getComposition(n, n');
 7:     conditionalPredicate+=Relaxation_with_serverNode;
 8:   end if
 9:   if (isDescendant(n, n') AND notRoot(n')) then
10:     Relaxation_with_serverNode = getComposition(n', n);
11:     conditionalPredicate+=Relaxation_with_serverNode;
12:   end if
13: end for
```

$(\texttt{info}, \texttt{publisher})$ and $(\texttt{publisher}, \texttt{name})$ would require checking for the predicates $ad(\texttt{info}, \texttt{publisher})$ and $ad(\texttt{publisher}, \texttt{name})$. Allowing for subtree promotion on the subtree rooted at `publisher` would require checking for the predicate $ad(\texttt{book}, \texttt{publisher})$. Finally, the possibility of leaf node deletions means that the predicate comparing `publisher` with `name` is optional.

Adaptive query processing. Static evaluation strategies guarantee that all partial matches that arrive at a server have gone through exactly the same server operations. With adaptive strategies, different partial matches may have gone through different sets of server operations, and hence may have different subsets of query nodes instantiated.

For example, given the query in Figure 2(a), partial matches arriving at the `publisher` server may have previously gone through any of the other servers; only the node corresponding to the query root is guaranteed to be present. Dealing with each of the exponential number of possible cases separately would be very inefficient from the point of view of query processing.

Therefore, we use Algorithm 1 to generate the set of predicates to be checked for a partial match arriving at each server.

First, given a partial match at the head of the input queue, the server uses an index to quickly locate all matches at that server node that satisfy the *relaxation* of the predicate relating to the query root node of the partial match (which is guaranteed to be present) with the server node in the original XPath query. This predicate is obtained by composing the labels on the edges along the path from the server node to the root in the query.

Second, each element identified in the first step is compared with the input partial match by using a *conditional predicate sequence*. Such a sequence is created by examining the relationship between the server node and nodes that are either its ancestors or descendants in the original XPath query pattern. The predicates are obtained by composing the labels on the edges from the server node to the query tree node. For any node n_i of the partial match that corresponds to a query node represented in the *conditional predicate sequence*, we then check for validation of the relaxation of the conditional predicate with the server node n (i.e., `publisher` in the example). If it is validated, we check whether it is an exact predicate validation. This approach of using conditional predicate sequences at server nodes also

Algorithm 2 Whirlpool

```
Require: Query Q, k
 1: relaxedQ = relax(Q);
 2: plan = generateJoinPlan(relaxedQ);
 3: rootN = rootNode(plan);
 4: routerQueue = evaluate(rootN);
 5: servers = nonRootNodes(plan);
 6: for each server S in servers do
 7:   queueAtS = evaluate(S);
 8:   serverQueues += queueAtS;
 9: end for
10: while (nonEmpty(routerQueue)OR(nonEmpty(serverQueues))) do
11:   answer = nextAnswer(routerQueue);
12:   sendToNextServer(answer);
13:   for each server S in servers do
14:     answerAtS = nextAnswerAtS(queueAtS);
15:     computeJoinAtS(answerAtS);
16:     checkTopK(topKSet,answerAtS);
17:     if aliveAtS(topKSet,answerAtS) then
18:       backToRouter(answerAtS);
19:       serverQueues -= answerAtS;
20:     end if
21:   end for
22: end while
23: return topKSet;
```

enables incremental assignment of updated scores with extensions to the input partial match.

5.2.2. Top-k Query Processing. We synthesize two approaches for top-k query evaluation:

Lock-step: This algorithm is similar to the one proposed in [2]. Different variations of the lock-step algorithms can be obtained by varying the components implementations (Section 6.1).

Whirlpool: Algorithm 2 shows the top-k evaluation algorithm that is instantiated by *Whirlpool*. A few functions are worth highlighting in this algorithm: `nextAnswer` implements the router decision for picking the next answer according to some policy; `sendToNextServer` implements a routing decision (see Section 6.1.4 for implementation alternatives); `nextAnswerAtS` implements the priority queue strategy at each server (see Section 6.1.3 for implementation alternatives); `computeJoinAtS` computes the join predicates at a server. This function can implement any join algorithm. Finally, `checkTopK` checks if a partial match needs to be discarded or kept using its current score and decides to update the top-k set accordingly.

6. Experimental Evaluation

We now discuss the implementation of each component in the *Whirlpool* architecture. Then, we describe our experimental settings, and present the experimental results.

6.1. Implementation Alternatives

In this section, we discuss *Whirlpool*'s choices for priority queues and routing decisions.

6.1.1. Scheduling between components. There are two overall scheduling possibilities:

Single-threaded: The simplest alternative is to have a single-threaded implementation of *all* the components

in the system. This would permit having complete control over which server processes next the partial match at the head of its input priority queue.

Multi-threaded: One can allocate a thread (or more) to each of the servers, as well as to the router, and let the system determine how to schedule threads. The use of priority queues (Section 6.1.3) and adaptive routing strategies (Section 6.1.4) permits "control" of query evaluation. In addition, by using different threads, *Whirlpool* is able to take advantage of available parallelism.

6.1.2. Evaluation Algorithms.

Whirlpool-M: Our multi-threaded variation of *Whirlpool*. Each server is handled by an individual thread. In addition to server threads, a thread handles the router, and the main thread checks for termination of top-k query execution.

Whirlpool-S: The single-threaded scheduling variation of *Whirlpool*. Due to the sequential nature of *Whirlpool-S*, we slightly modified *Whirlpool*'s architecture (Figure 4) in our implementation of *Whirlpool-S*: a partial match is processed by a server as soon as it is routed to it, therefore the servers' priority queues are not needed, and partial matches are only kept in the router's queue. Note that *Whirlpool-S* bears some similarities to both *Upper* [20] and *MPro* [7]. As in both techniques, partial matches are considered in the order of their maximum possible final score. In addition, as in *Upper*, partial matches are routed to the server using an adaptive technique. While *Upper* does not consider join evaluation, *MPro* use a join evaluation based on Cartesian product and individual evaluation of each join predicate score. In contrast, our techniques use a different model for join evaluation where one single operation produces all valid join results at once.

LockStep: *LockStep* considers one server at a time and processes all partial matches sequentially through a server before proceeding to the next server. Our default implementation of *LockStep* keeps a top-k set based on the current scores of partial matches, and discards partial matches during execution. We also considered a variation of *LockStep* without pruning during query execution, *LockStep-NoPrun*, where all partial match operations are performed, scores for all matches are computed, and matches are then sorted at the end so that the k best matches can be returned. Note that the *LockStep* algorithm is very similar to the *OptThres* algorithm presented in [2]. The relaxation adaptivity of *OptThres*, which decides whether a partial match will be considered for relaxation depending on its score, is included in the default server implementation of *Whirlpool*.

6.1.3. Priority Queues. Various strategies can be used for server prioritization:

FIFO: the simplest alternative is to process partial matches in the queue in their arrival order. This scheme is sensitive to the actual order in which partial matches are processed, and performance may vary substantially.

Current score: partial matches with higher current scores will be moved to the heads of their respective priority queues. This scheme is sensitive to the order in which partial matches are *initially* selected to be processed.

Maximum possible next score: the current score of a partial match is added to the maximum possible score it could receive from its current server, and partial matches with higher maximum possible next scores will be moved to the heads of their respective priority queues. This scheme adapts to the score that the current server could contribute to partial matches, making it less sensitive to the order in which partial matches are processed.

Maximum possible final score: the maximum possible final score determines which partial match to consider next. This scheme is less sensitive to the order in which partial matches are processed, and is the most *adaptive* of our queue prioritization alternatives. Intuitively, this enables those partial matches that are highly likely to end up in the top-k set to be processed in a prioritized manner akin to join ordering. Although not reported due to space constraints, we verified this conjecture experimentally.

6.1.4. Routing Decisions. Given a partial match at the head of the router queue, the router needs to make a decision on which server to choose next for the partial match. Obviously, a partial match should not be sent to a server that it has already gone through; maintaining a bit vector on the set of servers, with each partial match is used for this task. The routing choice could be made a few different ways:

Static: the simplest alternative is to route each partial match through the same sequence of servers. For homogeneous data sets, this might actually be the strategy of choice, where the sequence can be determined a priori in a cost-based manner.

Score-based: the partial match is routed to the server that is likely to impact its score the most. Two variations of this routing technique can be considered: routing the partial match to the server that is likely to increase its score the most (*max_score*), or the least (*min_score*), based on some precomputed, or estimated, information.

Size-based: the partial match is routed to the server that is likely to produce the fewest extended matches, after pruning against the top-k set. The intuition is that the overall cost of the top-k query evaluation is a function of the number of partial matches that are alive in the system. The size-based choice is a natural (simplified) analog of conventional cost-based query optimization, for the top-k problem, and can be computed using estimates of the number of extensions computed by the server for a partial match (such estimates could be obtained by using work on selectivity estimation for XML), the range of possible scores of these extensions, and the likelihood of these extensions getting pruned when compared against the top-k set.

In Section 6.3.1, we evaluate different partial match routing strategies for *Whirlpool*. In *Whirlpool-S*, the algorithm always chooses the partial match with the maximum possible final score as it is the one on top of the router queue. In addition, it is proven that this partial match will have to be processed before completing a top-k answer [20]. We tried several queue strategies for both *LockStep* and *Whirlpool-M* as described in Section 6.1.3. For all configurations tested, a queue based on the *maximum possible final score* performed better than the other queues. This result is in the same spirit

Query Size	Document Size	k	Parallelism	Scoring Function
3 nodes ($Q1$), ***6 nodes*** ($Q2$), 8 nodes ($Q3$)	1Mb, ***10Mb***, 50Mb	3, ***15***, 75	1, ***2***, 4, ∞	***sparse*** dense

Table 1. Evaluation parameters (defaults in bold).

as *Upper* [20] as it allows for partial matches that are likely to end up in the top-k set to be processed first. In the remainder of this paper, results that we report for *LockStep* and *Whirlpool-M* techniques assume server queues on *maximum possible final scores*.

6.2. Experimental Setup

We implemented the three top-k query processing strategies in C++, using POSIX threads for *Whirlpool-M*. We ran our experiments on a Red Hat 7.1 Linux 1.4GHz dual-processor machine with a 2Gb RAM and a Sun F15K running Solaris 8 with with 54 CPUs ranging from 900MHz to 1.2GHz, and 200Gb of RAM.

6.2.1. Data and Queries. We generated several documents using the XMark document generating tool[2]. We then manually created three queries by isolating XPath subsets of XMark queries that illustrate the different relaxations.

$Q1$: `//item[./description/parlist]`
$Q2$: `//item[./description/parlist and ./mailbox/mail/text]`
$Q3$: `//item[./mailbox/mail/text[./bold and ./keyword] and ./name and ./incategory]`

Edge generalization is enabled by recursive nodes in the DTD (e.g., `parlist`). Leaf node deletion is enabled by optional nodes in the DTD (e.g., `incategory`). Finally, subtree promotion is enabled by shared nodes (e.g., `text`).

When a query is executed on an XML document, the document is parsed and nodes involved in the query are stored in indexes along with their "Dewey" encoding[3]. Our server implementation of XPath joins at each server uses a simple nested-loop algorithm based on Dewey, since we are not comparing join algorithm performance. We discuss the effect of server operation time and its tradeoff with adaptive scheduling time in Section 6.3.3. Scores for each match are computed using the scoring function presented in Section 4.

6.2.2. Evaluation Parameters (x-axes) We measured the performance of our techniques for a variety of criteria (summarized in Table 1):
Query size: We consider 3 query sizes: 3 nodes, 6 nodes, and 8 nodes (see Section 6.2.1). The number of servers is

2 *http://monetdb.cwi.nl/xml/index.html*
3 *http://www.oclc.org/dewey/about/about_the_ddc.htm*

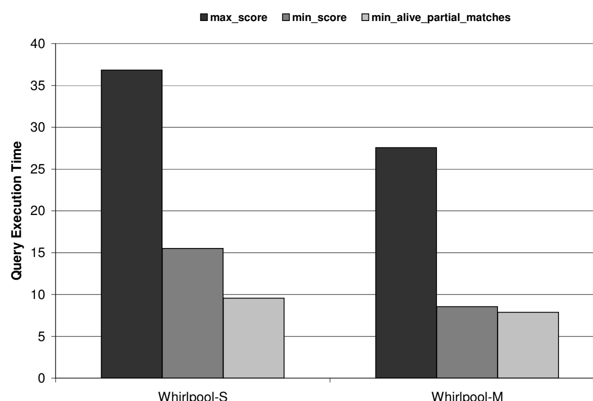

Figure 5. Query execution time for *Whirlpool-S* and *Whirlpool-M*, for various adaptive routing strategies.

equal to the number of nodes involved in a query. The number of partial matches and thus the number of server operations for a top-k strategy is, in the worst case, exponential in the number of nodes involved in the query.
Document size: We consider XMark documents of sizes ranging from 1Mb to 50Mb.
Value of k: We ran experiments for values of k ranging from 3 to 75. When the value of k increases, fewer partial matches can be pruned.
Parallelism: Our *Whirlpool-M* approach takes advantage of multiple available processors. We experimented with this strategy on different machines offering various levels of parallelism (from 1 to 48 processors).
Scoring function: We used the *tf*idf* scoring function described in Section 4. We observed that the *tf*idf* values generated for our XMark data set were skewed, with some predicates having much higher scores than others. Given this behavior, we decided to synthesize two types of scoring function based on the *tf*idf scores*, to simulate different types of datasets: *sparse*, where for each predicate, scores are normalized between 0 and 1 to simulate datasets where predicates scores are uniform, and *dense*, where score normalization is applied over all predicates to simulate datasets where predicate scores are skewed. (The terms *sparse* and *dense* refer to the effect of these functions on the distribution of final scores of partial matches.) We also experimented with randomly generated sparse and dense scoring functions. A sparse function allows for a few partial matches to have very high scores, resulting in high k^{th} score values, which enables more pruning. With a dense scoring function, final scores of partial matches are close to each other, resulting in less pruning. Using different scoring functions permits to study the impact of score distribution on our performance measures. Validating the scoring functions using precision and recall is beyond the scope of this paper and the subject of future work.

6.2.3. Evaluation Measures (y-axes). To compare the performance of the different techniques, we use the following metrics:

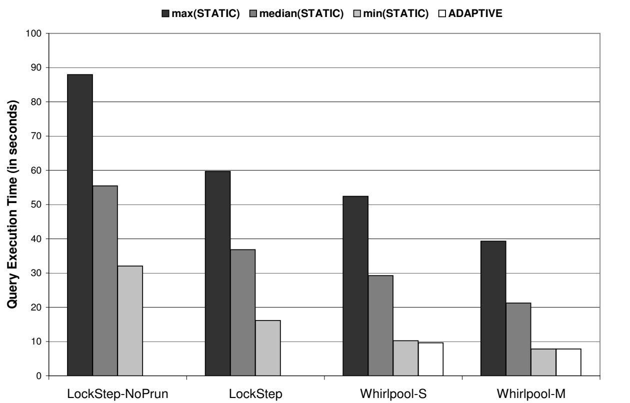

Figure 6. Query execution time for *LockStep-NoPrun*, *LockStep*, *Whirlpool-S* and *Whirlpool-M*, for static and adaptive routing strategies (linear scale).

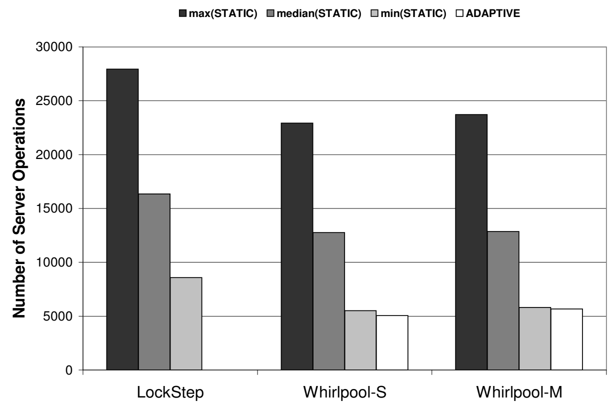

Figure 7. Number of server operations for *LockStep*, *Whirlpool-S* and *Whirlpool-M*, for static and adaptive routing strategies (linear scale).

Query Execution Time: Overall time needed to return the top-k answers.
Number of Server Operations: This measure allows us to evaluate the actual workload of the various techniques, regardless of parallelism.
Number of Partial Matches Created: The fewer the created partial matches, the better the top-k query processing technique is at pruning during query execution.

6.3. Experimental Results

We now present experimental results for our top-k query evaluation algorithms. We first study various adaptive routing strategies (Section 6.3.1), and settle on the most promising one. We then compare adaptive and static strategies (Section 6.3.2), and show that adaptive routing outperforms static routing when server operation cost dominates in the query execution time (Section 6.3.3), and that lock-step strategies always perform worse than strategies that let partial matches progress at different rates. We study the impact of parallelism (Section 6.3.4) and of our evaluation parameters (Section 6.3.5) on our adaptive techniques. Finally (Section 6.3.6), we discuss scalability.

6.3.1. Comparison of Adaptive Routing Strategies. We study the performance of adaptive routing strategies for our top-k techniques (Section 6.1.4). In particular, we considered the *max_score*, *min_score* and *min_alive_partial_matches* described in Section 6.1.4.

Figure 5 shows the query execution time for *Whirlpool-S* and *Whirlpool-M* for the three routing strategies for the default setting of Table 1. Choosing servers that increase partial match scores the most (*max_score*) does not result in fast executions as it reduces the pruning opportunities. In contrast, a score-based strategy that aims at decreasing partial matches scores (*min_score*) performs reasonably well. By basing routing decisions on the number of alive partial matches after the server operation, the size-based strategy (*min_alive_partial matches*) is able to prune more partial matches, and therefore decrease its workload (number of server operations), resulting in lower query execution times. Because *min_alive_partial_matches* performs better than all other tested routing strategies over all configurations tested for our adaptive *Whirlpool-S* and *Whirlpool-M* techniques, we will use *min_alive_partial_matches* as *Whirlpool*'s routing strategy in the rest of this paper.

6.3.2. Adaptive vs. Static Routing Strategies. We now compare adaptive routing strategies against static ones. Figures 6 and 7 show the query execution time and the number of server operations needed for *Whirlpool-S* and *Whirlpool-M*, as well as for both *LockStep* and *LockStep-NoPrun* using the default values in Table 1. For all techniques, we considered all (120) possible permutations of the static routing strategy, where all partial matches go through the servers in the same order. In addition, for *Whirlpool-S* and *Whirlpool-M*, we considered our adaptive strategy (see Section 6.3.1). For both *LockStep* strategies, all partial matches have to go through one server before the next server is considered, *LockStep* is thus static by nature. This implementation of *LockStep* is similar to the *OptThres* algorithm presented in [2].

For all techniques, we report the min, max and median values for the static routing strategy. A perfect query optimizer would choose the query plan that results in the min value of the static routing strategy. A first observation from Figures 6 and 7 is that for a given static routing strategy, *Whirlpool-M* is faster than *Whirlpool-S*, which in turn is faster than *LockStep*. Thus, allowing some partial matches to progress faster than others, by letting them being processed earlier by more servers, results in savings in query execution time and total number of server operations. The no-pruning version of *LockStep* is obviously worse than all other techniques, proving the benefits of pruning when processing top-k queries. In addition, for both *Whirlpool-S* and *Whirlpool-M*, we see that our adaptive routing strategy results in query execution times at least as efficient as the best of the static strategies. (For dense scoring functions, adaptive routing strategies resulted in much better performance

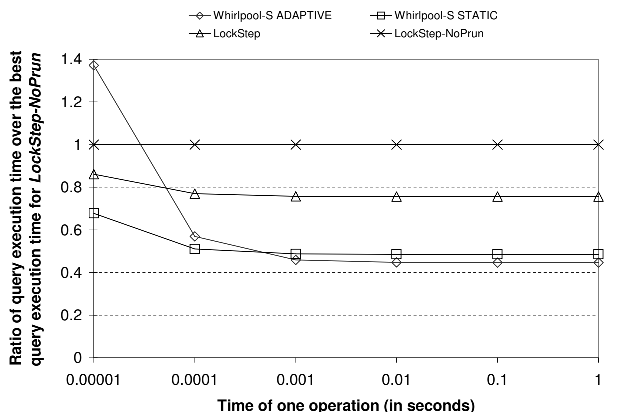

Figure 8. Ratio of the query execution time of the different techniques over *LockStep-NoPrun*'s best query execution time, for different join operation cost values.

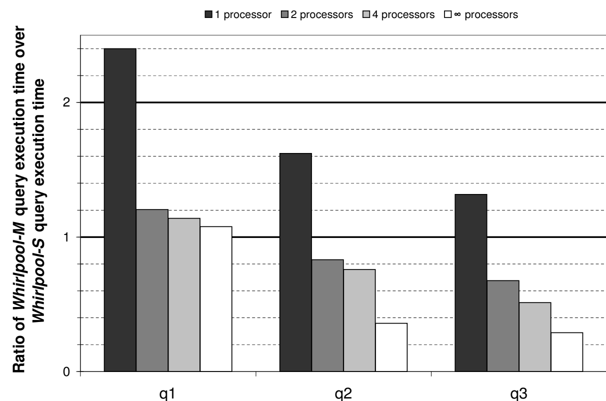

Figure 9. Ratio of *Whirlpool-M*'s query execution time over *Whirlpool-S*'s query execution time.

than the best static strategy.) Interestingly, for this default setting, *Whirlpool-M* performs slightly more server operations than *Whirlpool-S*. However, the better performance of *Whirlpool-M* is due to its use of parallelism (2 processors are available on our default machine) to speed up query processing time.

Since *Whirlpool* always outperforms *LockStep*, and *Whirlpool*'s adaptive routing strategy performs as well as or better than its static one, we will only consider the adaptive routing versions of *Whirlpool-S* and *Whirlpool-M* in the rest of this paper. The terms *Whirlpool-S* and *Whirlpool-M* will now refer to their adaptive versions.

6.3.3. Cost of Adaptivity. While adaptivity allows to reduce the number of server operations, and therefore leads to reduction in query processing time, it also has some overhead cost. In Figure 8, we compare the total query execution time of *Whirlpool-S* with both static and adaptive routing strategies to that of the best *LockStep* execution (both with and without pruning). Results are presented relative to the best *LockStep-NoPrun* query execution time. (We do not present results for *Whirlpool-M* in this section as it is difficult to isolate the threading overhead from the adaptivity overhead.) While for static routing strategies, an adaptive per-tuple strategy (*Whirlpool-S-STATIC*) always outperforms the *LockStep* techniques by about 50%, the adaptive version of *Whirlpool-S* performs worse than the other techniques if server operations are very fast (less than 0.5msecs). For query executions where server operations take more than 0.5msecs each, *Whirlpool-S-ADAPTIVE* is 10% faster than its static counterpart. (For larger queries or documents, the tipping point is lower than 0.5msecs, as the percentage of tuples pruned as a result of adaptivity increases.) Adaptivity is then useful when server operation time dominates in the query execution time. However, when server operations are extremely fast, the overhead of adaptivity is too expensive. These results are similar to what was observed in [12] and [20]. As a final observation, in scenarios where data is stored on disk, server operation costs are likely to rise; in such scenarios, adaptivity is likely to provide important savings in query execution times. In the future, we plan on performing adaptivity operations "in bulk", by grouping tuples based on similarity of scores or nodes, in order to decrease adaptivity overhead.

In this paper, we present results for the case where join operations cost around 1.8 msecs each.

6.3.4. Effect of Parallelism. We now study the effect of parallelism on the query execution time of *Whirlpool-M*. Note that in *Whirlpool-M*, the number of threads is equal to the number of servers in the query + 2 (router thread and main thread), thus *Whirlpool-M* is limited in its parallelism. To show the maximum speedup due to parallelism of *Whirlpool-M* we performed experiments over an infinite number of processors. (The actual number of processors used in the experiment is 54, which is much higher than the 10 processors that *Whirlpool-M* would use for $Q3$.)

Unlike *Whirlpool-M*, *Whirlpool-S* is a sequential strategy, thus its execution time is not affected by the available parallelism. To evaluate the impact of parallelism on *Whirlpool-M* execution time, we ran experiments on a 10Mb document for all three queries, using 15 as the value for k, on four different machines with 1, 2, 4 and ∞ processors respectively.[4] We then computed the speedup of *Whirlpool-M* over the execution time of *Whirlpool-S*, and report our results in Figure 9. When there is no parallelism, i.e., when the number of available processors is equal to one, the performance of *Whirlpool-M* compared to that of *Whirlpool-S* depends on the query size: *Whirlpool-M* can take more than twice the time of *Whirlpool-S* for small queries but becomes faster, when parallelism is available, for large queries. When multiple processors are available, *Whirlpool-M* becomes faster than *Whirlpool-S*, up to 1.5 times faster with two processors, up to 1.95 times faster with four processors, and up to a maximum of almost 3.5 times faster when the number of avail-

4 Our 4-processors machine is actually a dual Xeon machine with four "logical" processors.

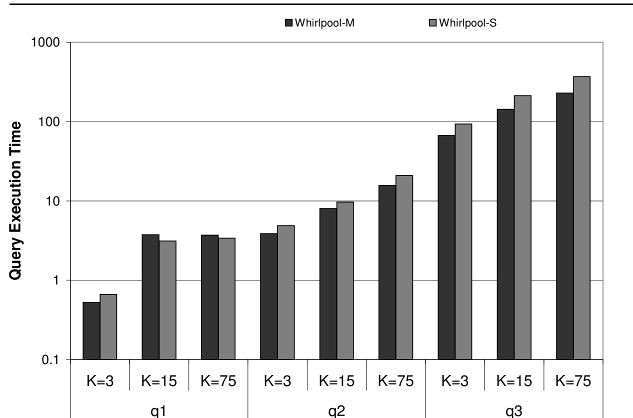

Figure 10. Query execution time for *Whirlpool-S* and *Whirlpool-M*, as a function of k and the query size (logarithmic scale).

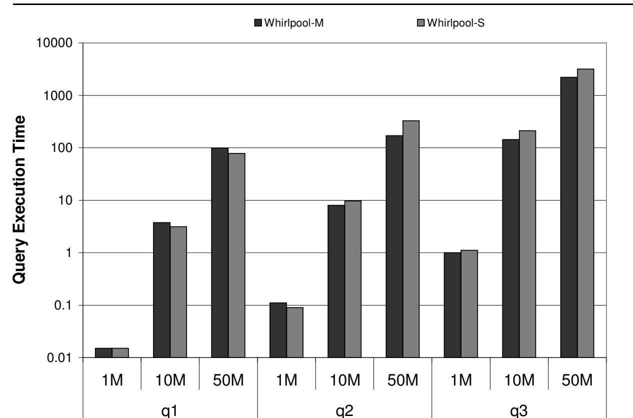

Figure 11. Query execution time for *Whirlpool-S* and *Whirlpool-M*, as a function of the document and query sizes (logarithmic scale, k=15).

able processors is unlimited. For $Q1$, *Whirlpool-M* is not faster than *Whirlpool-S*, even when parallelism is available, as $Q1$ only has three servers and does not take as much advantage of parallelism as $Q2$ and $Q3$, making the threading overhead expensive in comparison to the gains of parallelism. In addition, $Q1$ is evaluated faster than $Q2$ and $Q3$, and is thus more penalized by the threading overhead. For $Q2$ and $Q3$, *Whirlpool-M* takes advantage of parallelism, with better results for the larger $Q3$ than $Q2$, as it is a larger query.

The speedup stops increasing when the number of processors exceeds the number of threads needed to evaluate the query. Our example queries do not take advantage of parallelism greater than the number of servers involved in the query + 2 (router and main threads). Thus $Q1$ does not benefit from more than 5 processors, $Q2$ from more than 8 processors, and $Q3$ from more than 10 processors. If more parallelism is available, we could create several threads for the same server, thus increasing parallelism even more. This is the subject of future work.

6.3.5. Varying Evaluation Parameters. We now study the effect of our parameters from Section 6.2.2.

Varying Query size: Figure 10 shows the query execution time for both *Whirlpool-S* and *Whirlpool-M* for our three sample queries (Table 1), on a logarithmic scale. The query execution time grows exponentially with the query size. Because of the logarithmic scale, the differences between *Whirlpool-S* and *Whirlpool-M* are larger than they appear on the plot. The difference between *Whirlpool-M* and *Whirlpool-S* query execution time increases with the size of the query, with *Whirlpool-S* 20% faster for $Q1$ and *Whirlpool-M* 48% faster for $Q3$ (k=15), since the threading overhead has less impact on larger queries.

Varying k: Figure 10 reports the effect of varying the number of matches returned in the top-k answer. The query execution time is linear with respect to k. Interestingly, the difference in query execution time between *Whirlpool-S* and *Whirlpool-M* increases with k. This increase is more significant for larger query sizes, and *Whirlpool-M* is up to 60% faster than *Whirlpool-S* for $Q3$, k=75. The number of server operations exhibits a similar behavior (although at a smaller scale), with 8% fewer server operation for *Whirlpool-M* for the $Q3$, k=75 setting. This is rather counter-intuitive: [7] proved that sequential top-k algorithms based on probing the partial match with the highest possible final score, as does *Whirlpool-S*, minimizes the total number of operations *with respect to a given routing strategy*. Since our implementations of *Whirlpool-S* and *Whirlpool-M* use the same routing strategy, *Whirlpool-S* should always perform fewer server operations. The explanation lies in our adaptive routing strategy: *min_alive_partial_matches* relies on score estimates, server selectivity and current top-k values to make its choice. This last parameter, current top-k values, changes during query execution. Monitoring the executions of *Whirlpool-S* and *Whirlpool-M* show that top-k values grow faster in *Whirlpool-M* than in *Whirlpool-S*, which may lead to different routing choices for the same partial match, making the algorithms follow, in effect, different schedules for the same partial match. By making better routing choices, *Whirlpool-M* results in fewer partial matches being created than *Whirlpool-S*.

Varying Document Size: Figure 11 reports on the effect of the XML document size on the query execution time. The execution time grows exponentially with the document size; the larger the document, the more partial matches will have to be evaluated resulting in more server operations and thus longer query execution times. For a small document, the result is quite fast (less than 1.2 sec for all queries tested), making the threads overhead in *Whirlpool-M* expensive compared to *Whirlpool-S* execution time. However, for medium and large documents, *Whirlpool-M* becomes up to 92% faster than *Whirlpool-S* ($Q2$, 50M document, k=15).

Varying Scoring Function: We experimented with different scoring functions: both sparse and dense variations of the *tf*idf* scoring function, as well as randomly generated scoring functions that were designed to have either dense or

Document Size	1M	10M	50M
$Q1$	100%	93.12%	85.66%
$Q2$	100%	49.56%	57.66%
$Q3$	100%	39.59%	31.20%

Table 2. Percentage of partial matches created by *Whirlpool-M*, as a function of the maximum possible number of partial matches, for different query and doc. sizes.

sparse properties. We observed that sparse scoring functions lead to faster query execution times (due to faster pruning). In contrast, with dense scoring functions, the relative differences between *Whirlpool-M* and *Whirlpool-S* is greater with *Whirlpool-M* resulting in greater savings in terms of query processing time, number of server operations and partial matches created, over *Whirlpool-S*.

6.3.6. Scalability. A top-k processing technique over XML documents has to deal with the explosion of partial matches that occurs when query and document sizes increase. To measure the scalability of *Whirlpool*, we considered the number of partial matches created during query execution, as a ratio of the maximum possible number of such partial matches. The total number of partial matches is obtained by running an algorithm with no pruning (*LockStep-NoPrun*). Table 2 shows that the percentage of total possible partial matches created by *Whirlpool-M* significantly decreases with the document and query sizes. The benefits of pruning are modest for small queries. While all partial matches are created for $Q1$, for which tuples generated by the root server do not create "spawned" tuples in the join servers, pruning allows to reduce the number of operations of these partial tuples. For large queries ($Q3$), *Whirlpool-M* evaluates less than 40% of the partial matches on the 10M document, and less than 32% on the 50M document. By pruning partial matches based on score information, *Whirlpool-M* (and *Whirlpool-S*) exhibits good scalability in both query and document size.

7. Conclusion

In this paper, we presented *Whirlpool*, an adaptive evaluation strategy for computing exact and approximate top-k answers of XPath queries. Our results showed that adaptivity is very appropriate for top-k queries in XML. We observed that the best adaptive strategy focuses on minimizing the intermediate number of alive partial matches; this is analogous to traditional query optimization in RDBMS, where the focus is on minimizing intermediate table sizes. By letting partial matches progress at different rates, *Whirlpool* results in faster query execution times. In addition, *Whirlpool* scales well when query and document sizes increase. While the focus in this paper is not on evaluating XPath scoring functions, we show that *Whirlpool* adapts itself to environments where scores of intermediate answers are either sparse or dense. We studied the effect of parallelism on our *Whirlpool* approaches and observed that although *Whirlpool-M* is better for most cases, if parallelism is not available, or if query or document size is small, *Whirlpool-M* threading overhead may result in decreased performance. In contrast, for large queries and documents, *Whirlpool-M* exploits available parallelism and results in significant savings in query execution time over *Whirlpool-S*. We are investigating new directions such as increasing the number of threads per server for maximal parallelism.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. Zdonik Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal 2003, 120-139.

[2] S. Amer-Yahia, S. Cho, D. Srivastava. Tree pattern relaxation. EDBT 2002.

[3] S. Amer-Yahia, L. Lakshmanan, S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. SIGMOD 2004.

[4] R. Avnur, J. Hellerstein. Eddies: Continuously Adaptive Query Processing. SIGMOD 2000.

[5] N. Bruno, S. Chaudhuri, L. Gravano. Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems (TODS), 27(2), 2002.

[6] M. J. Carey, D. Kossmann. On Saying "Enough Already!" in SQL. SIGMOD 1997.

[7] K. C.-C. Chang, S.-W. Hwang. Minimal Probing: Supporting Expensive Predicates for Top-K Queries. SIGMOD 2002.

[8] C. Chen, Y. Ling. A Sampling-Based Estimator for Top-K Query. ICDE 2002.

[9] T. T. Chinenyanga, N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data. WebDB 2001.

[10] E. Damiani, N. Lavarini, S. Marrara, B. Oliboni, D. Pasini, L. Tanca, G. Viviani. The APPROXML Tool Demonstration. EDBT 2002.

[11] C. Delobel, M.C. Rousset. A Uniform Approach for Querying Large Tree-structured Data through a Mediated Schema. International Workshop on Foundations of Models for Information Integration (FMII-2001). Viterbo, Italy.

[12] A. Deshphande. An Initial Study of the Overheads of Eddies. SIGMOD Record Feb. 2004.

[13] R. Fagin, A. Lotem, M. Naor. Optimal Aggregation Algorithms for Middleware. PODS 2001.

[14] R. Fagin. Combining Fuzzy Information from Multiple Systems. PODS 1996.

[15] N. Fuhr, K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. ACM SIGIR Workshop on XML and Information Retrieval. Athens, Greece, 2000.

[16] V. Hristidis, N. Koudas, Y. Papakonstantinou. PREFER: A system for the Efficient Execution Of Multiparametric Ranked Queries. SIGMOD 2001.

[17] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. VLDB 2003.

[18] Y. Kanza, Y.Sagiv. Flexible Queries over Semistructured Data. PODS 2001.

[19] R. Kaushik, R. Krishnamurthy, J. Naughton and R. Ramakrishnan. On the Integration of Structure Indices and Inverted Lists. SIGMOD 2004.

[20] A. Marian, N. Bruno, L. Gravano. Evaluating Top-k Queries over Web-Accessible Databases. ACM Transactions on Database Systems (TODS), 29(2), 2004.

[21] A. Natsev, Y. Chang, J. R. Smith, C. Li, J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. VLDB 2001.

[22] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983

[23] T. Schlieder. Schema-driven evaluation of approximate tree-pattern queries. EDBT 2002.

[24] A. Theobald, G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. EDBT 2002.

[25] T. Urhan and M. Franklin. Cost-Based Query Scrambling for Initial Query Delays SIGMOD 1998, 130-141.