Query Processing for High-Volume XML Message Brokering

Yanlei Diao
University of California, Berkeley
diaoyl@cs.berkeley.edu

Michael Franklin
University of California, Berkeley
franklin@cs.berkeley.edu

Abstract

XML filtering solutions developed to date have focused on the matching of documents to large numbers of queries but have not addressed the customization of output needed for emerging distributed information infrastructures. Support for such customization can significantly increase the complexity of the filtering process. In this paper, we show how to leverage an efficient, shared path matching engine to extract the specific XML elements needed to generate customized output in an XML Message Broker. We compare three different approaches that differ in the degree to which they exploit the shared path matching engine. We also present techniques to optimize the post-processing of the path matching engine output, and to enable the sharing of such processing across queries. We evaluate these techniques with a detailed performance study of our implementation.

1. Introduction

For distributed environments including Web Services, data and application integration, and personalized content delivery, XML is becoming the common *wire format* for data. In this emerging distributed infrastructure, XML *message brokers* [20][25][26] will play a key role as central exchange points for messages sent between applications and/or users. The main functions of such brokers are: *filtering*, *transformation*, and *routing*. *Filtering* matches messages to a large set of queries that represent the data interests of specific users, applications, or organizations. *Transformation* restructures matched messages according to recipient-specific requirements. *Routing* involves the transmission of the customized data to the recipients.

Recently, there have been a number of systems developed for XML filtering [1][3][7][8][13][14][17], where the queries typically involve *path expressions* that refer to the structure of the XML data items. The most efficient filtering systems exploit commonality among queries via shared processing of the path expressions.

The work we describe in this paper is aimed at developing the next level of functionality, i.e., transforming the XML messages on a query specified basis, in order to provide customized data delivery and to enable cooperation among disparate, loosely coupled services and applications. High-capacity brokering systems must be capable of supporting potentially tens of thousands of simultaneous queries. Thus, approaches that process queries individually are not adequate for our purpose. Since shared processing of path expressions has been shown to be an efficient and scalable foundation for the current generation of XML filtering systems, we start with such an engine, which we call a shared path matching engine, and develop alternatives for building customization functionality on top of it. We address the following fundamental questions:

- How, and to what extent can a shared path matching engine be exploited for customized result generation?
- What additional *post-processing* of path matching output is needed to support message customization, and how can this post-processing be done most efficiently?

By way of answering these questions, we have developed three techniques that differ in the extent to which they push work down into the matching engine. As we will show, there is an inherent tension between shared path matching and customized result generation. That is, aggressive path sharing requires more sophisticated post-processing.

Given an efficient shared path matching engine, it is easy for post-processing to become the dominant component of query processing cost. In order to reduce the cost of post-processing we have developed provably safe optimizations based on query and DTD (if available) inspection that enable us to eliminate unnecessary operations and choose more efficient operator implementations for post-processing of individual queries.

We have also developed a set of techniques for sharing post-processing work across multiple queries. These techniques are similar in spirit to approaches used in more generic Continuous Query processing systems, but as we will show, are highly tailored for the specific case of large-scale, high-volume XML message brokering.

We have implemented all of the above techniques on top of the YFilter shared path matching engine [8][11] and have evaluated their effectiveness with a detailed performance analysis of the implementation.

The paper proceeds as follows. Section 2 presents our problem definition. Sections 3 and 4 present three alternative solutions and a set of optimizations for them. Section 5 ad-

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599 and by Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

dresses shared post-processing. Section 6 presents our experimental results. Section 7 covers related work. Section 8 presents conclusions.

2. Background

2.1 Architectural Overview

Our proposed XML message broker architecture is shown in Figure 1. The primary inputs are the queries that represent subscriptions and the XML messages themselves.

Queries become active as soon as they arrive at the message broker. Inside the broker, an arriving query is parsed for use by the Query Processor, where the execution plan of the new query is merged with the existing queries without recompiling any of them.

Incoming messages are filtered and transformed on-the-fly for the entire set of queries. These messages need not conform to DTDs (*Document Type Definitions*) but, as we describe later, such conformance can be exploited[1]. Internally, the broker runs an incoming message through an event-based XML parser. Parsing events are passed to the query processor to drive the query execution. They are also used to incrementally construct a node-labeled tree, which provides materialization of the parsed message for later use. The nodes are assigned integer identifiers according to a pre-order traversal of the tree.

The query processor produces output in an intermediate format that contains identifiers of nodes in the parsed XML message organized for efficient translation into customized output messages. The intermediate output of the query processor is fed to the "message factory", which combines the element tags in queries with the intermediate output and forwards the resulting messages for delivery.

We describe the query processor in more detail, after first presenting our problem definition in the following section.

2.2 Problem Statement

We focus on user query specifications written in a subset of XQuery [2]. Consider "**Query 1**" below, which is based on the *Book* DTD from the XQuery use cases [6]:

```
<sections>
{
  for     $s in document("doc.xml")//section
  where   $s//figure/title = "XML processing"
  return  <section>
                { $s/title }
                { $s//figure }
          </section>
}
</sections>
```

This query specifies that for each section containing a figure whose title is "XML processing", a "section" element containing the title of that section and all of its figures should be returned. Note that in a message conforming to the *Book* DTD, section elements may contain other sections as well as figures and other elements. This query requires results to be returned for *all* sections matching the query in the same order that the matching sections appear in the message.

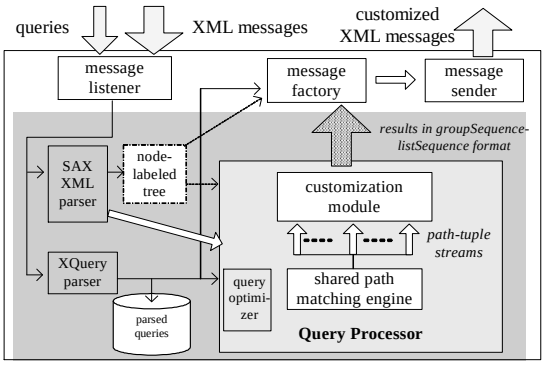

Figure 1: XML message broker architecture

More specifically, the queries we consider consist of a single *FLWR* (i.e., *For-Let-Where-Return)* expression enclosed in an element defined by a *constant tag*. The *FLWR* expression contains:

- A *for* clause containing a *variable name* and a *path expression*; followed by
- An optional *where* clause that contains a set of conjunctive predicates, each of which takes a form of a triplet: *path expression*, *op*, *constant*; followed by
- A *return* clause that contains interleaved *constant tags* and *path expressions*, where all constant tags have a matching close tag.

Our current implementation does not support the *let* clause.

The semantics of these queries is as follows: The *for* clause creates an *ordered* sequence of variable bindings to document elements (or in our case, to nodes in the parsed XML message). The *where* clause, if present, restricts the set of bindings passed to the *return* clause. The *return* clause is invoked once for each variable binding. At each invocation of the return clause, tags cause the construction of new XML fragments and path expressions select nodes from the current variable binding. The final result of the FLWR expression is an *ordered* sequence of the results of these invocations.

For conciseness, we refer to the path expression in a *for* clause as the "*binding path*", those in a *where* clause as "*predicate paths*", and those inside a *return* clause as "*return paths*". We require that the predicate and return paths of a query be relative to the binding path of that query (i.e., they are prefixed by the variable name used in the binding path).

A path expression consists of a sequence of location steps. We support location steps with child "/" and descendent "//" axes and element name tests. Path expressions containing such location steps are referred to as *navigation paths* in this paper. We also allow location steps to contain simple predicates that compare the attributes or text data of elements to a constant. In this work, binding paths can contain an arbitrary number of simple predicates in any location step. A predicate path is a navigation path with a simple predicate attached to the last location step, and itself is a complex

[1] In applications such as web services, XML Schema is more widely used than DTDs. Since the structural information we exploit is provided by both types of definition, XML schema can be used under the same conditions as DTDs in this work.

predicate imposed on its binding path. A return path is simply a navigation path.[2]

As stated in Section 2.1, the output of the query processor is an intermediate representation that is passed on to the message factory component of the broker. In this representation, the nodes selected from the message are organized into a sequence of groups, such that each group corresponds to a single invocation of the *return* clause. Inside a group, nodes are contained in a sequence of lists. The sequencing of lists corresponds to the ordering of the return paths in the *return* clause. Each list contains the nodes matching the return path in their document order. For example, the output of Query 1 would have the following format:

….
$section_i$: [$title_{i1}$], [$figure_{i1}$, …]
$section_{i+1}$: [$title_{(i+1)1}$], [$figure_{(i+1)1}$, …]
….

where $section_i$ represents a group, and the numbering …, i, i+1, … represents the ordering of those groups. The sequence inside a group consists of a list of identifiers of title nodes (in our example there is only a single title per section) followed by a list of identifiers of figure nodes. In the remainder of this paper, we refer to this intermediate representation as the *groupSequence-listSequence* format.

Having described our model of queries and output, we can now formulate the core message broker functionality we provide as follows:

Given a large set of queries written in the specified query language, efficiently extract message components in the groupSequence-listSequence format for all queries for each message arriving at the message broker.

2.3 Query Processor Details

In our system, as shown in Figure 1, the query processor consists of two main runtime components: a *shared path matching engine* and a *customization module*. Given a parsed query, the optimizer in the query processor inserts navigation paths from the query into the shared path matching engine, and adds the execution plan for the remainder of the query to the customization module. For an incoming message, the shared path matching engine takes the parsing events to match its contained navigation paths. The customization module further processes the output of the path matching engine to generate customized results.

A key advantage of this design is that it leverages the prior work on building scalable, shared XML path matching engines [1][3][7][8][11][13]. We chose to base our system on YFilter [8][11], a high-performance shared path matching engine that we built previously. YFilter employs a single *Non-Deterministic Finite Automaton* to represent the full set of navigation paths, and supports shared processing of the common prefixes of all these paths. A recent study by Bruno et al. [3] shows that YFilter is particularly effective for short messages and large sets of queries; precisely the environment we anticipate for high-volume XML message brokers.

Our approaches to customized result generation are developed in the context of the particular *output format* provided by YFilter. For a navigation path matched by an incoming message, YFilter delivers a stream of "path-tuples" each of which represents a unique match of this path. A path-tuple contains one field per location step in the path, and the value of the field is the identifier of the message node bound to the location step. When multiple paths are matched by a message, YFilter delivers its output as streams of path-tuples, one stream for each path.

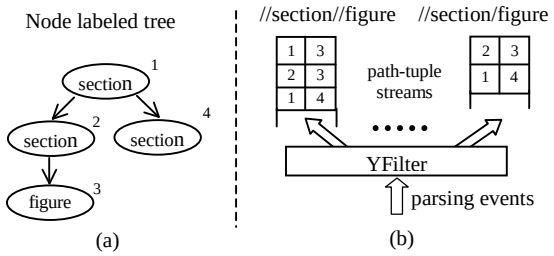

Figure 2: An example of YFilter output

Figure 2(a) shows a node-labeled tree for a message fragment, where nodes are annotated with their assigned ids. Path-tuple streams that are output from YFilter for different paths are illustrated in Figure 2(b). Take the stream for the path "//section//figure". It contains three path-tuples. Each path-tuple contains two node ids, representing a unique combination of the two location step bindings.

YFilter guarantees that path-tuples in each stream are produced such that the node ids in the last field of the path-tuples appear in monotonically increasing order. This stream order is exploited in our processing algorithms as described in the following sections. It is also important to note that ordering on other fields of path-tuples is not guaranteed by YFilter.

3. Basic Approaches

In this section, we present three different query processing approaches that differ in the extent to which they exploit the path matching engine. In all of them, a post-processing phase is applied to the output of the matching engine to generate the complete *groupSequence-listSequence* output. This post-processing is done via query plans using relational-style operators. In the approaches described in this section, we use one such query plan per XQuery query (i.e., the post-processing phase is not shared). We examine how to share post-processing work in Section 5.

It should be noted that much of the subtlety of developing solutions to this problem arises from the inherent tension between shared processing at the lower level (which is essential for good performance) and customized query result generation. The matching engine returns the path-tuples in a stream in a single, fixed order to all queries that include the corresponding path. The paths, however, may be used quite differently by the various queries, and thus potential inconsistencies such as unintended duplicates or ordering problems can arise with aggressive path sharing (we will discuss both of these cases in detail shortly). In the following, we describe our approaches in order of increasing path sharing, and focus on how the additional complications raised by increased sharing are addressed. The approaches are additive;

[2] The approaches we describe in this paper can be extended to support more general XQuery scenarios. Due to space limitations, we refer the interested reader to [9] for further details.

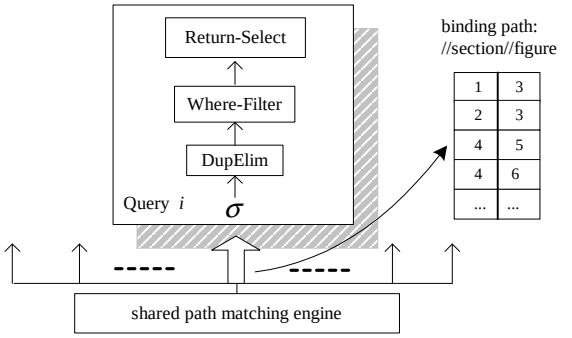

Figure 3: A query plan using PathSharing-F

that is, the approaches exploiting increased sharing incorporate those that use less.

3.1 Shared Matching of "For" Clauses

The first approach we describe uses the path matching engine to process only binding paths (i.e., paths that appear in for clauses). We begin by inserting the navigation part of the binding path from each query into the engine. Then, during the processing of a message, the output of the engine for each path is directed to the post-processing plans for its corresponding queries. We refer to this approach as *PathSharing-F*. Consider **Query 2**:

```
<figures>
{
  for $f in document("doc.xml")//section[@id<=2]//figure
 where  $f/title = "XML processing"
 return  <figure>  { $f/image }  </figure>
}
</figures>
```

Figure 3 highlights the post-processing plan for this query under *PathSharing-F*. In the figure, the multiple arrows above the matching engine represent the streams of path-tuples (note that queries that have a common binding path share a common stream). The thick arrow denotes the stream used by Query 2, which contains the path-tuples matching the binding path "//section//figure". In the following, we refer to the last field of these path-tuples as the *binding field*, because they contain the ids of the nodes that are actually bound by the binding paths. We refer to these nodes as the *BoundNodes*. The box above the thick arrow contains the post-processing execution plan. The operators in this plan are, from bottom-up:

Selection. A selection operator is placed at the bottom of a query plan to evaluate any simple predicates (i.e., comparisons of the attributes or text data of elements to a constant) attached to a binding path. The evaluation is done for each path-tuple by checking predicates against the nodes referenced by the path-tuple. Selection emits only those path-tuples for which all predicates evaluate to True.

Duplicate Elimination (DupElim). The XQuery specification requires that duplicate nodes bound to a path be eliminated based on the node identity [2]. Accordingly, we define duplicates in the stream for a binding path as path-tuples that contain the same node id in the binding field.

Such duplicates arise when multiple path-tuples in a stream reference the same BoundNode. For example, consider Query 2 and the XML fragment:

"<section id=1> <section id=2> <figure> <title> XML processing </title> </figure> </section> </section>"

The matching engine outputs two path-tuples for the binding path. The first corresponds to "<section id=1> <figure>" and the second to "<section id=2> <figure>". These two path-tuples reference the same BoundNode, so the second could cause redundant work and produce a duplicate result.

The DupElim operator avoids these problems by ensuring that each BoundNode is emitted at most once. In this case, a simple scan-based DupElim operator can be used because as described in the previous section, path-tuples in the stream are ordered by their binding field. It should be noted, however, that DupElim cannot be pushed before the selection, because it is not known which (if any) of the path-tuples referencing the same BoundNode will pass the selection.

Where-Filter. This operator evaluates the *where* predicates on each path-tuple until a predicate evaluates to False or the entire *where* clause evaluates to True. Path-tuples in the latter case are emitted. For each path-tuple, a predicate path is evaluated with a tree search routine that uses a depth-first search in the sub-tree of the parsed message rooted at the *BoundNode* of the path-tuple. The search routine for a path returns True as soon as any node satisfying the predicate is found. The pseudo-code of this routine is omitted in the interest of space.

Return-Select: This operator applies the *return* clause to the *BoundNode*s of the path-tuples that survive the Where-Filter. It uses the tree search routine for each return path. Unlike the Where-Filter, however, the tree search routine here must retrieve *all* nodes matching a return path rather than stopping at the first match.

Return-Select generates results in the *groupSequence-listSequence* format. Each input path-tuple causes the creation of a new group. The ordering of return paths in the query defines the sequence of lists within each group. For each list, the matches of the corresponding return path are placed in the order that they appear in the message.

Recall that the results of a FLWR expression must be ordered in accordance with the order of the variable bindings of the *for* clause. Since the stream for the binding path is ordered in this way, and the remaining processing steps do not change that order, we are assured that the order produced by *PathSharing-F* is correct.

3.2 Shared Matching of "Where" Clauses

PathSharing-F only uses the path matching engine to process binding paths. The next approach, *PathSharing-FW*, in addition pushes the navigation part of predicate paths from the *where* clause into the matching engine to exploit further sharing. Recall that predicate paths are defined to be relative to the binding paths. Since the matching engine treats all paths as being independent, we must first extend the predicate paths by prepending their corresponding binding path. For example, consider **Query 3**:

```
<sections>
{
  for     $s in document("doc.xml")//section
 where   $s/title="XML"
      and $s/figure/title = "XML processing"
 return   <section>
```

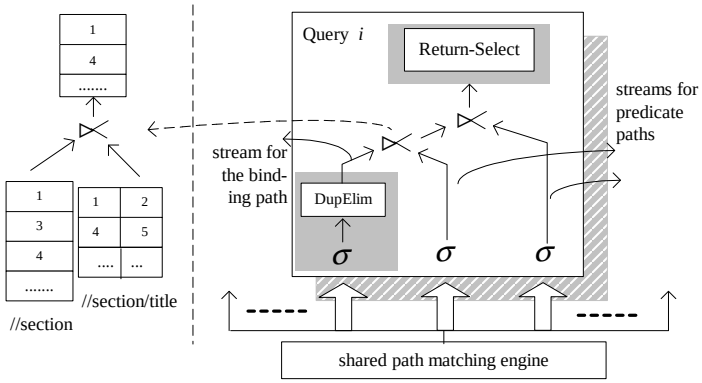

Figure 4: A query plan using PathSharing-FW

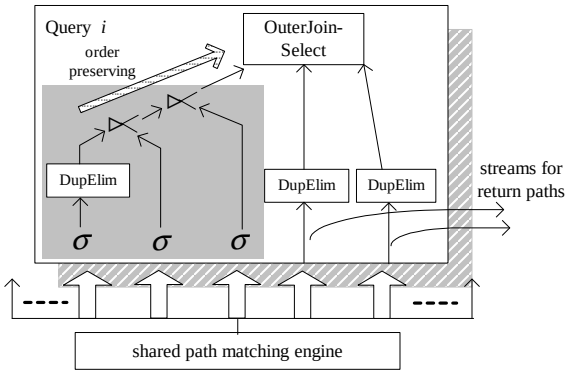

Figure 5: A query plan using PathSharing-FWR

```
            { $s//section//title }
            { $s/figure }
        </section>
}
</sections>
```

The first predicate path "/title" is transformed into "//section/title" and the second becomes "//section/figure /title". These extended predicate paths, along with the binding path, are inserted into the matching engine. Note that since common prefixes of paths are shared in the matching engine, the extension of these paths does not add significantly to their processing cost.

As in *PathSharing-F*, the path-tuple streams for each query are then post-processed by a query plan that executes the remaining portion of that query. This arrangement is shown in Figure 4. The stream corresponding to a binding path is passed through a selection operator and a DupElim operator as before. The output of the DupElim operator is then matched with the streams corresponding to the predicate paths. The path-tuples resulting from the matching process are piped to a Return-Select that works as described before.

In *PathSharing-FW*, the Where-Filter of *PathSharing-F* is replaced by a left-deep tree of *semijoins* with the binding path stream as the leftmost input. Recall that the predicate paths are extended by pre-pending them with the corresponding binding path. Thus, the common field on which each semijoin will match is the *binding field*, i.e., the last common field between the binding path tuples and the predicate path tuples. The result of a semijoin, therefore, is a stream containing only those binding path tuples that have matching predicate path tuples. Figure 4 shows an example for the leftmost semijoin.

The semijoin operators can be implemented using a simple merge-based algorithm, if it is known that the predicate path streams are delivered in monotonically increasing order of *BoundNode* id. In general, however, there are cases where such ordering cannot be assumed. Consider the execution of Query 3, when applied to the following XML fragment:

> "<section> <section> <figure> <title> XML processing </title> </figure> </section> <figure> <title> XML processing </title> </figure> </section>"

In this case, the stream for the predicate path "//section/figure/title" would contain a path-tuple corresponding to "$section_2$ $figure_1$ $title_1$" followed by a path-tuple corresponding to "$section_1$ $figure_2$ $title_2$", where the subscript indicates the first or the second occurrence of the tag name. This stream is not properly ordered by the *binding field* (i.e., section). In such cases, since the binding path stream is ordered properly, we can use a hash-based implementation of semijoin where the binding path stream is used as the probing stream. Sufficient conditions for determining when the more efficient merge-based approach can be used are discussed in Section 4. Note, however, that both approaches order the output correctly, resulting in semantics identical to those provided by *PathSharing-F*.

A final note is that duplicates in predicate path streams are not a concern, because these streams are only used to filter binding path tuples that have passed a DupElim operator.

3.3 Shared Matching of "Return" Clauses

Our third alternative approach, *PathSharing-FWR*, aims at further increasing sharing by also pushing the return paths into the path matching engine. Return paths differ from predicate paths in that they do not constrain the set of matching binding path tuples so the semijoin approach cannot be used for them. Instead, *outer-join* semantics are required.

We require a slightly more specialized operator than a generic outer-join, however, because results must be generated in the *groupSequence-listSequence* format. Thus, we have implemented our own n-way outer-join operator, which we call *OuterJoin-Select*. As Figure 5 shows, OuterJoin-Select takes as its leftmost input, the binding path stream resulting from the semijoins of the *PathSharing-FW* approach. It performs left outer joins on the *binding field* with each of the return path streams. Generation of the results in the required format is performed as part of the outer join processing. Each path-tuple in the binding path stream causes the creation of a new group. The outer join between this path-tuple and a return path stream results in a new list within the group, containing the BoundNode ids in the return path tuples that have matched the binding path tuple. If no such matches are found, an empty list is kept in the group for this return path.

In our implementation, OuterJoin-Select builds hash tables for each of the return path streams and then probes them in a pipelined fashion using a single scan of the stream emitted by the semijoin tree. In this way, the output of this operator is guaranteed to be ordered by the binding field.

Note from Figure 5 that, DupElim operators are required on each of the return path streams to prevent duplicate results from being generated by OuterJoin-Select. Here, the notion

of duplicates is defined on the combination of the *binding field* and the last field of the path-tuple, called the *return field*.

Recall that a return path stream is always ordered by the *return field*. If it also arrives ordered by the *binding field*, a scan-based approach suffices for DupElim. Otherwise, hashing is used.

As can be seen, PathSharing-FWR, the approach that exploits path sharing to the fullest extent, requires the most sophisticated post-processing. As we mentioned earlier, this complexity results from the tension between shared path matching and result customization. It is important to note that this problem cannot be easily solved in the path matching engine. Consider a path expression that is the binding path in one query and a return path in another. In this case, the path-tuple stream produced for that path expression will be used (by different queries) as two different types of streams. Since the two types of streams have different notions of duplicates, duplicate elimination cannot be done in the engine, but must be done in a usage-specific manner during post-processing. Similar issues arise with the ordering of path-tuples expected by the different uses of the stream.

4. Simplifying Post-Processing

Duplicates and stream ordering are two fundamental issues that complicate post-processing for customized result generation. With additional knowledge however, it is sometimes possible to infer cases when duplicates cannot arise, or when path-tuples will arrive in a needed order. In the first case, DupElim operators can be removed from the post-processing plans. In the second case, cheaper scan or merge-based operator implementations can be used in place of the more expensive hash-based ones.

4.1 Sufficient Conditions

We have derived a set of sufficient conditions that enable the detection of some situations where post-processing can be simplified. These conditions involve the presence of "//" axes in queries, and the potential for *recursive elements* (i.e. elements that have the same element name and contain each other) in the messages. The first type requires examining the queries, the second can be checked by examining a DTD, if present. The claims involving a DTD utilize a *DTD element graph* constructed as follows: Start at the root of the DTD and examine its child elements. If a node for a child element is not in the graph, create one. Then draw a directed edge from the parent element to each child element. Repeat this for all elements.

The conditions are described in the following five claims. Correctness proofs for these claims are given in [9]. Consider a path expression p of m location steps, and the stream of path-tuples that match the path, with fields numbered 1..m.

Claim 1: If p contains at most one "//" axis, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected on field m.

Claim 2: If p contains n, $n > 1$ "//" axes, then if the elements of the first $n-1$ location steps containing a "//" axis do not appear on a loop in the DTD element graph, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected on field m.

Claim 3: Partition p into two paths, one consisting of location steps 1 to i, $i < m$, and the other being a relative path consisting of the rest of the path. If claim 1 or claim 2 indicate that no duplicates exist for either path, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected onto fields i and m.

Claim 4: If there is no "//" axis from location steps 1 to i, $1 \leq i < m$ of p, then the stream of path-tuples matching p will be in increasing order when projected onto field i.

Claim 5: If p contains one or more "//"axes within location steps 1 to i, then if for all steps j, $j \leq i$ containing a "//" axis, the elements of location steps j and i do not appear on the same loop in the DTD element graph, then the stream of path-tuples matching p will be in increasing order when projected onto field i.

4.2 Optimization of Post-Processing Plans

The preceding claims enable optimizations of post-processing plans on a query-by-query basis as follows:

- Claim 1 (and 2, if a DTD is present) is used to check if there can be any duplicates in the path-tuple stream for a binding path. Recall that duplicates for binding path tuples are defined on the *binding field*, the last field of binding path tuples. If duplicates are not possible, we remove the DupElim operator for the binding path.
- Claim 3, in conjunction with Claim 1 (and 2, if a DTD is present) is used to check the possible existence of duplicates in the path-tuple stream for a return path. Recall (from Section 3.3) that for return paths, duplicates are defined based on the combination of the binding field and the return field. Thus, Claim 3, is tested with i set to the location of the binding field. If duplicates are not possible, we remove the DupElim operator for the return path.
- Claim 4 (and 5, if a DTD is present) is used to check if all input streams for a semijoin or OuterJoin-Select are guaranteed to be ordered by the binding field, with i set to the location of the binding field. If yes, the merge based versions of these operators can be used in place of the more expensive hash-based implementation. These claims are also used to determine if a scan-based DupElim operator can be used for each return path.

Consider the application of these claims for Queries 2 and 3 of the previous section using *Pathsharing-FWR*. Assume that the element "section" is on a loop in the DTD element graph, but the element "figure" is not. For Query 2 (see Section 3.1), the tests for Claims 1-3 fail, and in fact, duplicates can arise, as described in Section 3.1. The test for Claim 4 also fails because of the "//section//figure" in the binding path. The test for Claim 5, however, succeeds because although the two location steps in the binding path both contain "//" axes and the element "section" is on a DTD element loop, the element "figure" is not on any loop with "section". Therefore all predicate and return path streams are guaranteed to be ordered by the binding field. Thus, cheaper operators can be used for semijoin, Outer Join-Select and the DupElim on the return path stream.

For Query 3 (see Section 3.2), if we apply Claim 1 (or 2) with Claim 3 to its query plan, all DupElim operators except

the one for the return path "//section//title", can be removed. The remaining DupElim operator results from the presence of two "//"s in the return path and the fact that element "section" after the first "//" is on a DTD loop.

The performance impact of these optimizations can be quite significant, and is studied in the experiments presented in Section 6.

5. Shared Post-Processing

So far we have presented three ways to share path matching among queries. A common feature of these approaches is that they all require a separate post-processing plan for each query. In this section we describe an initial set of techniques that can further improve sharing by allowing some of the post-processing work to be shared across related but non-identical queries, in particular, ones that have path expressions (and hence, path-tuple streams) in common.

A prerequisite to the techniques we describe here is a way to determine which path expressions appear in multiple queries. The technique we use is to associate with each query a set of unique path identifiers corresponding to each of the paths that appear in it. These identifiers are returned by the path matching engine when the paths are initially inserted.

Our techniques are similar in spirit to techniques proposed for shared *Continuous Query* (CQ) processing over (typically non-XML) data streams [4][5][15][16][19]. Unlike the *generic* functionality provided in CQ systems, however, the approaches we use are highly tailored for large-scale XML filtering and customization. For ease of exposition, we focus the discussion on the post-processing plans used by *PathSharing-FWR* with DTD-based optimizations (as described in the previous section), which are shown in the experimental results to outperform the other approaches in most cases.

5.1 Query Rewriting

As a first step to enhance sharing among queries, whenever the appropriate DTD is available we rewrite path expressions into a canonical form before inserting them into the path matching engine. This rewriting collapses certain expressions that are semantically (but not syntactically) equivalent, allowing their corresponding queries to share a single path-tuple stream for the path. The rewriting focuses on removing superfluous "//" axes. A "//" axis is superfluous if the DTD shows that there is a single path from the element before "//" to the element after "//". If so, then we can replace "//" with the deterministic sequence of '/' steps. For example, a return path "figure//image" can be rewritten to "figure/image" if the DTD shows that an image element can only be the child but not the descendent of a figure element.

5.2 Sharing Techniques

Most work on CQ systems considers selection and join operators in a relational (or close to it) framework. In contrast, our work on XML message brokering is focused a subset of XQuery and involves a unique set of operators and a specific data flow through these operators, as presented in Section 3. The specialized nature of our work leads us to a particular set of sharing techniques, three of which are described below.

Shared GroupBy for OuterJoin-Select: In the implementation as described so far, each OuterJoin-Select operator does its own hashing (or scanning) of the path-tuple streams it consumes for return paths (i.e., all but the leftmost stream). When multiple queries share a common return path, this approach incurs redundant processing. This redundancy can be expensive, because return paths are not constrained by predicates; thus, these streams may carry a large number of path-tuples.

We propose to remove this redundancy by placing GroupBy operators before OuterJoin-Selects on those streams that provide return path tuples. A GroupBy operator groups path-tuples in a return path stream by the *binding field*, so that the subsequent OuterJoin-Select can simply get all the return path tuples matching a binding path tuple by obtaining the matching group. Each GroupBy operator is shared by all OuterJoin-Selects that process the corresponding return path. Thus, their overhead is expected to be small. Implementationwise, if the stream of a return path is ordered by the binding field, the GroupBy is scan based. Otherwise, it is hash based. Duplicate elimination, if necessary, is performed in a scan-based manner in the GroupBy itself.

Having addressed return path processing, we now turn our attention to the post-processing of binding paths and predicate paths.

Selection-DupElim pull up: We first consider shared processing of semijoins among multiple queries. The common relational optimization of pushing selections below joins makes it difficult to share join processing. Pulling selection up over joins [4] avoids this problem. In our setting, we pull selections with their subsequent DupElim operators, if present, over semijoins, and turn semijoins into shared joins. We currently only implement this technique for queries with a single predicate path.

The technique works as follows. Our semijoins are said to have "signatures" consisting of the path ids for their two inputs (a binding path on the left and a predicate path on the right). We create a shared join for all semijoins with the same signature. When converting a semijoin to a join, we retain all path-tuple fields for later use in selections. To be consistent with semijoin semantics, our shared joins are also implemented to preserve the order of the left input stream. The decision on merge- or hash- based implementation carries over from semijoins to shared joins.

Shared selection: Above a shared join operator, selections can be grouped by their signatures [5][4][15][19]. In the XML setting for our problem, a predicate signature is a quadruplet (path id, level, attribute name, operator), where the level specifies the location step in the path containing the predicate. For sharing, we currently only consider a single predicate per path. Given this restriction, the signature for a selection above a join is simply the pair of predicate signatures from the joined paths. The constant of a selection signature is the pair of constants in the two predicates from the joined paths. Selections with the same signatures are replaced by a shared selection where different constants are merged into a single index. A shared selection can have multiple outputs, one for each constant of the selection signature matched by the XML data.

Shared joins may produce path-tuples containing the same node id in the *binding field*. Fortunately, shared joins

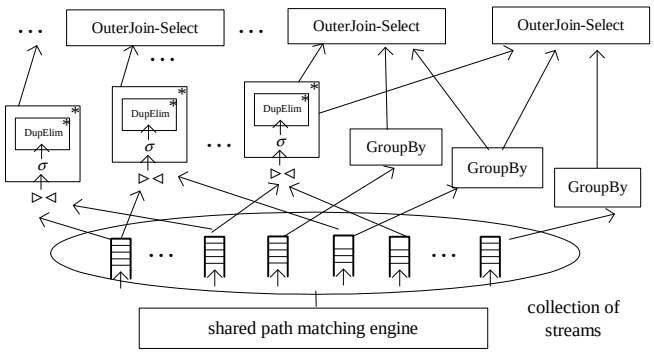

Figure 6: Shared post-processing example

pre-serve the order on the binding field in their output, so scan-based DupElim can be used on the selection outputs.

An example of a shared post-processing plan is given in Figure 6. Here a box annotated with '*' means there is a set of such operators. On top of the path matching engine there is a set of merged plans sharing joins and selections, and a set of GroupBy operators shared by OuterJoin-Selects. Each OuterJoin-Select takes the left input from one output of a merged plan and the rest of its inputs from the GroupBys.

5.3 Query Plan Construction and Execution

The construction of the shared post-processing plans is done incrementally. When a new query is entered into the broker, we first construct a standalone post-processing plan for the query. We then determine its relationship to the current shared plans by examining its path ids and signatures. Operators in the new plan are either merged with existing ones or result in the creation of new branches.

The execution of such large-scale shared query plans is a non-trivial issue. *NiagaraCQ* [4][5] placed a *split* operator to direct the output of one operator to all the subsequent operators. That operator, however, copies tuples (or pointers to tuples) when multiple subsequent operators require them. We experimented with a split operator copying tuple pointers in our initial implementation, and found that it imposed a significant performance overhead. *CACQ* [19] avoids this problem using *tuple lineage*, which records the operators that a tuple has passed or needs to pass inside the tuple itself. The overhead of tuple lineage, however, increases with the number of queries.

In our work, we used an alternative technique that places the pointers to path-tuples in each output of an operator in a data structure called *tpList*, and lets all the subsequent operators share the *tpList*(s) for their input. During query plan construction, each operator allocates one or more tpLists; each subsequent operator must remember which tpList to read from. Most operators have a single tpList. There are two exceptions, however. The path matching engine requires a tpList per path-tuple stream and a shared selection requires a tpList per constant of its signature. The tpLists in the latter cases can be instantiated lazily so they incur overhead only if they are actually used.

During post-processing execution, each operator places the pointer to each output path-tuple to one of its tpLists. Upon completion of an operator, all the subsequent operators read from the desired tpLists and start their execution. A possible disadvantage of this technique is that the scheduler has to check all the subsequent operators even though some tpLists are known to be empty. Our experimental results in Section 6.5 show that this overhead is quite small in practice.

6. Experimental Evaluation

We implemented the techniques described in the preceding sections using the YFilter shared path matching engine. In this section, we present the results of a detailed performance study of this implementation. We first compare the performance of the three basic approaches with and without optimizations when individual post-processing plans are used for distinct queries. We then examine the scalability of these approaches and the impact of shared post-processing.

6.1 Experimental Setup

Both YFilter and our message brokering extensions are written in Java. All of the experiments were performed on a Pentium III 850 Mhz processor with 768MB memory running IBM J2RE 1.3.0 on Linux 2.4. We set the JVM maximum allocation pool to 600MB, so that virtual memory activity had no influence on the results.

To test the system, we required generators for both documents and queries. For documents, we developed a document generator based on IBM's XML Generator [10], which takes a DTD as input, and produces documents that conform to that DTD, according to a set of workload parameters. We use the default settings for all those parameters except for the following three.

DocDepth bounds the depth of element nesting in the generated XML documents. In this work, we are less concerned with the absolute document depth, but rather, focus on the depth of recursive elements. This is because document depth mainly impacts path navigation, while deeply recursive data stresses the post-processing aspects of our solution by requiring DupElim and hash-based operators when "//" axes are used in queries.

The parameter *MaxRepeats* determines the number of times an element can repeat in its parent element. We have modified the generator so that *MaxRepeats* can be varied on an individual element basis. A large value of *MaxRepeats* produces more matches of a query within a document, generating a larger result set for each matched query.

The parameter *MaxValue* determines the number of values that the data of elements and attributes of elements can take, therefore affecting the selectivity of predicates.

We also developed a query expression generator that uses the query workload parameters shown in Table 1. We ensure that all generated queries are unique. To so do, predicates in the where clause are sorted lexicographically. We also sort return paths, since two queries that are the same except the ordering of return paths can share most processing with only some trivial reordering at the end. Hashing on the query after path sorting is used to determine if it is unique. Predicates in the generated queries take values from a range of size MaxValue, so this parameter determines the selectivity of predicates. A large value of MaxValue produces fewer matches per query, but also can increase the number of unique queries for scalability evaluation.

We report on experiments with two DTDs: the *Bib* and *Book* DTDs from the XQuery use cases[6]. The *Bib* DTD is

used to generate non-recursive documents; the *Book* DTD is used to generate documents that can contain multiple levels of recursion. For each DTD, we generated a set of 200 documents using one setting of the workload parameters. For each run, 20 of these documents are used to warm up the JVM runtime compiler. Thus, all reported experimental results represent the average over 180 documents. For each experiment, queries were generated according to a specific query workload setting. For a given experiment, each algorithm was run individually in a separate Java process.

Parameter	Values	Description
Q	5,000 – 100,000	The number of distinct queries.
D1	2, 3	The maximum depth of a binding path
PP	1 - 3	The number of predicate paths in a query
RP	1 - 4	The number of return paths in a query
D2	2	The maximum depth of predicate paths or the return paths
DSProb	0 - 0.4	The probability of a "//" axis occurring in any location step in a path expression

Table 1: Workload parameters for query generation

The main performance metric we report is ***Multi-Query Processing Time*** (*MQPT*), which is defined as the time from the scan of a parsed document starting until the last result in the *groupSequence-listSequence* format is returned to the calling program. The cost of parsing is not included in our reported results, but was usually below 100 milliseconds.

We also implemented a profiler that reports the cost of each operator for a run of an experiment. MQPT times reported here were taken with the profiler turned off. Where appropriate, we use data from runs with profiling turned on to explain the performance results. Due to the overhead of running the profiler, the costs reported in this manner are higher than those observed in the actual experiments.

6.2 Shared Path Matching – Non-recursive Data

We first report on tests with the *Bib* DTD, which contains no recursion. For document generation, *DocDepth* was set to 4 because the DTD allows at most four levels of element nesting. We varied *MaxRepeats* such that in each document a bib element contains 20 books and each book has up to five authors or editors. On average, each document contains 149 start/end element pairs. *MaxValue* is set to 10.

6.2.1 Expt. 1 – Basic performance

In the first experiment, we compare the performance of the three approaches for moderate query loads (i.e., $Q = 5000$). In this experiment, queries were generated using the settings $D1 = 2$, $PP = 1$, $RP = 2$, $D2 = 2$, and $DSProb = 0.2$. Under this workload, a single *where* clause predicate is applied to book elements bound by the *for* clause. The *return* clause identifies two types of sub-elements from each remaining book element.

We first ran the three approaches with no optimizations. The leftmost group of bars in Figure 7 (labeled "NoOpt") shows their MQPT (in msec). In this case, PathSharing-FW has the lowest cost and PathSharing-FWR has the highest. PathSharing-FW outperforms PathSharing-F due to the shared path matching for all the predicates. Our profiler reports that evaluating all predicate paths using tree search in PathSharing-F takes 386 ms, while for PathSharing-FW, the equivalent work takes only 231 ms (27ms for predicate path matching by the engine, 57ms for selection, and 147ms for semijoins). On the other hand, PathSharing-FW handles return paths using the tree-search based Return-Select operator, at a cost of 212ms, while PathSharing-FWR uses 648ms to perform the equivalent functionality using Outer Join-Selects (244ms) and DupElim for return paths (404ms) (note that there is almost no additional cost for processing the return paths by the engine).

Next, we apply the optimizations described in Section 4. The results are shown in the middle and right groups in Figure 7, where Opt(q) indicates optimizations based only on queries and Opt(q+dtd) indicates those also using the DTD. For this latter case we also apply the path rewriting described in Section 5.1 to speed up path matching in the engine and in Where-Filter and Return-Select operators. We make the following observations:

- The query-based optimizations improve performance for all alternatives, but particularly for those that exploit more path sharing. PathSharing-FWR benefits significantly, outperforming the other two in this case.
- More sophisticated optimizations using the DTD enable further improvements for all three approaches. With these optimizations, PathSharing-FWR outperforms the others by a wide margin.

More detailed results for PathSharing-FWR are shown in Table 2. Three operators, namely, DupElim, semijoin and OuterJoin-Select, particularly benefit from the optimizations. With opt(q), most of the DupElim cost is avoided and the costs of semijoin and OuterJoin-Select are more than halved. When the DTD is also utilized, DupElim is unnecessary, and semijoin and OuterJoin-Select only each require around 20 ms. Note that the matching engine denoted as PME in the table, is indeed a less dominant component of the overall cost.

Operators	PME	Selection	DupElim	Semijoin	OuterJoin
No opt	28	61	451	140	235
Opt (q)	27	51	15	67	112
Opt (q+dtd)	9	42	0	18	22

Table 2: Costs (ms) of operators (PathSharing-FWR)

The reduced cost of the three operators is further explained by the change in the resulting query plans, as shown in Table 3. The improvement of DupElim arises because fewer such operators are needed with better optimization. The reduction in time for semijoin and OuterJoin-Select results from the ability to use merge-based implementations more often. For the Bib DTD, since no elements are on a DTD loop, Opt(q+dtd) can completely avoid DupElim and hash based implementations (as described in Section 4.1).

Operators	Semijoin		OuterJoin		DupElim
	#hash	# merge	#hash	#merge	#DupElim
No opt	5000	0	5000	0	15000
Opt (q)	1966	3034	1966	3034	429
Opt(q+dtd)	0	5000	0	5000	0

Table 3: Profile for 5000 queries (PathSharing-FWR)

The above results demonstrate the effectiveness of the optimization techniques. In conjunction with these techniques, PathSharing-FWR provides significantly better performance than the other two alternatives, despite its more com-

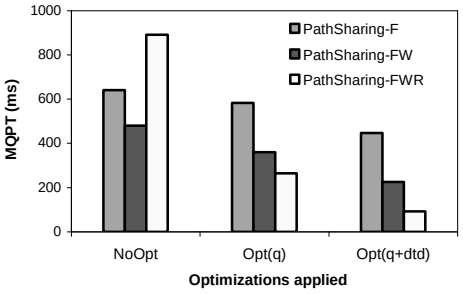

Figure 7: MQPT of three alternatives (Bib, Q=5000, PP=1, RP=2, DSProb=0.2)

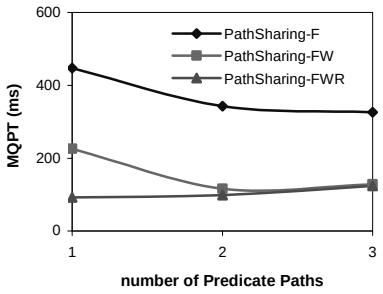

Figure 8: Varying PP (Bib, Q=5000, RP=2, DSProb=0.2, Opt(q+dtd))

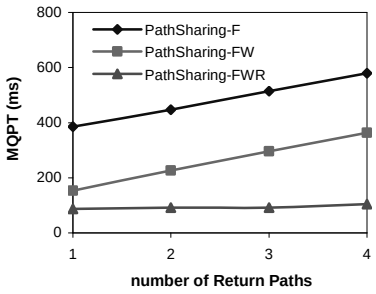

Figure 9: Varying RP (Bib, Q=5000, PP=1, DSProb=0.2, Opt(q+dtd))

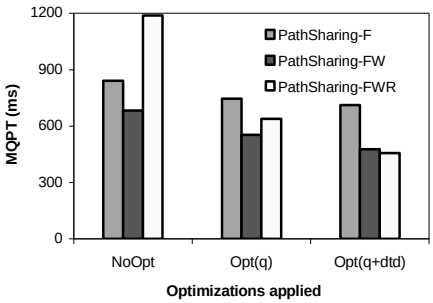

Figure 10: MQPT of three alternatives (Book, Q=10000, PP=1, RP=2, DSProb=0.2)

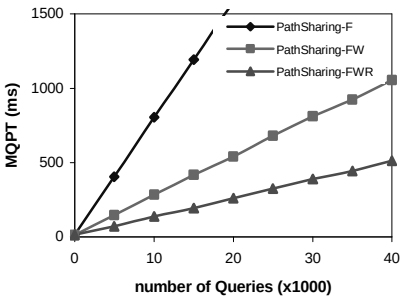

Figure 11: Varying Q (Bib, PP=1, RP=2, DSProb=0.2, Opt(q+dtd))

plicated post-processing. Thus, the post-processing optimizations help resolve the conflict between shared path processing and customized result generation.

6.2.2 Expt. 2 - Varying the number of predicates.

In the next experiment, we vary the number of predicate paths (*PP*) from 1 to 3. Increasing *PP* makes each query more selective in addition to requiring more predicates to be evaluated. Figure 8 shows the results using Opt(q+dtd).

The main observation is that more predicates reduce the differences among three alternatives. For alternatives using Return-Select, more predicates improve their MQPT because the extra predicates reduce the number of query matches, resulting in much less work for Return-Select. These savings outweigh the modest increase in cost for predicate evaluation. An additional observation is that with three predicates in each query, only 116 matches were found for all 5000 queries, which explains why PathSharing-FW and PathSharing-FWR are so close at that point. In this workload, further increasing the number of predicate paths tends to result in no matches, so we stop increasing this parameter here.

6.2.3 Expt. 3 - Varying the number of return paths.

Figure 9 shows the results obtained when the number of return paths in the queries is varied from 1 to 4. Again, we show results only for the Opt(q+dtd) case. In this experiment, the MQPT of PathSharing-F and PathSharing-FW increases linearly because with the fixed query selectivity, more return paths require more executions of the tree search routine. PathSharing-FWR is much less sensitive to the increased workload, because the matching of the return paths is shared among 5000 queries. Also, by using a merge-based approach, OuterJoin-Selects are efficient even when the number of streams involved in the outer joins increases.

6.3 Shared Path Matching – Recursive Data

In the next set of experiments, we use the *Book* DTD to generate documents with recursive elements. *DocDepth* is set to 5 so that we obtain up to four levels of nesting of section elements. *MaxRepeats* is set such that there are 12 top-level section elements in each book, and in each section, p (i.e., paragraph), figure, and section elements are allowed to repeat four times. The average document length is 83 start-end element pairs. *MaxValue* is set to 10.

Figure 10 shows the MQPT of the three alternatives when queries were generated using the settings: *Q* = 10,000, *D1* = 3, *PP* = 1, *RP* = 2, *D2* = 2, *DSProb* = 0.2. Under this workload, the evaluation of the *for* clause can bind section, paragraph (p), or figure elements to the variable. The results are similar to those of the previous experiments except that with Opt(q), PathSharing-FWR is outperformed by Path Sharing-FW, and with Opt(q+dtd) the advantage of Path Sharing-FWR is less pronounced.

We do not show the detailed cost breakdowns due to lack of space. The key points are that for PathSharing-FWR, the optimizations successfully reduce the DupElim cost, but the costs for semijoin and OuterJoin-Select remain high. This is due to the recursive section elements and the presence of "//" axes in the queries. In this situation, it is likely that path-tuples generated for predicate paths and return paths are not ordered by the *binding field*. Consequently, many semijoins and outer joins must be hash based, even with Opt(q+dtd).

Note that we also ran experiments varying the number of predicates and number of return paths for the *Book* DTD. The results are similar to those reported for the *Bib* DTD so we do not show them here.

6.4 Scalability

Next, we ran experiments to test the scalability of the approaches in terms of the number of queries (i.e., *Q*). Figure 11 shows the MQPT for the three approaches with Opt(q+dtd), using *Bib* documents, as *Q* is varied from 5,000 to 40,000. In order to create a sufficient number of unique queries here, the *MaxValue* parameter was increased to 100 for both document and query generation; the other parameters are set as in the basic experiment, i.e., Expt. 1.

As can be seen in the Figure, the MQPT for all three approaches grows linearly with *Q*. Since the solutions studied in this experiment do not share any post-processing, such an increase is to be expected. Note also that the rate of increase is highest for PathSharing-F, which exploits the shared path matching engine the least.

Similar results were obtained using the *Book* DTD, but with an even sharper increase in MQPT due to the additional impact of recursive data on post-processing costs. Table 4 shows the detailed cost breakdown for PastSharing-FWR with Opt(q+dtd) in this case, as *Q* is varied from 10,000 to 50,000. The increasing semijoin and OuterJoin-Select costs become dominant as *Q* increases, while the costs of selection and DupElim also increase. As we explained in Section 6.3, post-processing is more expensive for the *Book* DTD because of the need for hash-based operators.

Q	**10,000**	**20,000**	**30,000**	**40,000**	**50,000**
Selection	93	191	267	380	498
DupElim	30	62	111	146	183
Semijoin	137	320	484	659	847
OuterJoin	163	364	592	810	1025
Others	…	…	…	…	…
Executor	73	152	182	314	384
Total	**516**	**1111**	**1715**	**2344**	**2985**

Table 4: Costs(ms) as Q varies - PathSharing-FWR (Book DTD)

6.5 On Shared Query Execution

The results reported in the previous section demonstrated the scalability limitations of approaches that share only path matching work. In this section we examine the additional benefits to be gained by applying the techniques for sharing post-processing described in Section 5.

In the following experiments, we first generated individual query plans for PathSharing-FWR with Opt(q+dtd). From these individual plans, we built shared execution plans using the three strategies from Section 5: pulling selections above joins, grouping selections, and using GroupBy on return paths for outer joins.

We also rewrote the queries to increase commonality as described in Section 5.1. Two effects of this optimization were noticed. First, as expected, it does reduce the number of unique paths. Furthermore, we found that some previously unique queries could completely share a query plan because their signatures became identical after this rewriting.

Here, we focus on results obtained using the (recursive) *Book* DTD (experiments with the *Bib* DTD tell a similar story). Figure 12 shows the MQPT of PathSharing-FWR without shared post-processing and with (labeled "Plan Sharing") as the number of unique query plans is varied from 10,000 to 100,000 (note that "*Q*" is roughly 20% higher than this, but those queries sharing query plans with others do not incur extra cost in both algorithms here). As shown in the figure, shared post-processing leads to dramatic reductions in cost and concomitant improvements in scalability; The results here show the PlanSharing approach handling 100,000 unique query plans in only 472ms.

Table 5 shows the cost breakdown of PlanSharing. A comparison with Table 4 provides insight into the reduction of the overall cost, which results from four major factors:

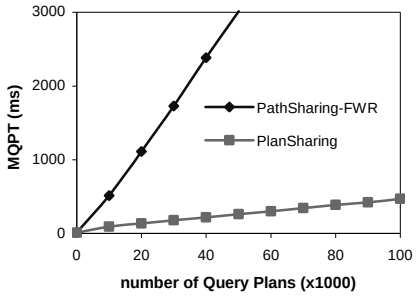

Figure 12: Varying number of unique query plans (Book, PP =1, RP=2, DSProb=0.2, Opt(q+dtd))

- The high cost of semijoins in PathSharing-FWR is reduced dramatically, because joins are now shared;
- Grouped selections reduce the selection cost (note that the cost of scan-based DupElim is included in the selection numbers, because it is folded into the selection operator.)
- OuterJoin-Selects are substantially cheaper, because the GroupBy technique removes redundant scanning and hashing at very little cost. Note that OuterJoin-Select is the only operator that exhibits a noticeable increase, as in our current implementation, the outer joins themselves are not shared.
- The cost of the Executor is also significantly reduced due to the reduction in query plan size.

Q **(Unique plans)**	**10,000 (8,232)**	**20,000 (16,482)**	**30,000 (24,576)**	**40,000 (32,736)**	**50,000 (40,392)**
Selection	18	18	24	18	21
GroupBy	4	3	5	6	5
Join	18	19	19	21	17
OuterJoin	29	58	81	117	138
Others	…	…	…	…	…
Executor	7	16	22	28	37
Total	**105**	**156**	**212**	**264**	**317**

Table 5: Costs (ms) as *Q* varies - PlanSharing (*Book* DTD)

6.6 Summary of Experiments

The experiments reported here have examined the performance of the three alternatives we proposed for exploiting a shared path matching engine to provide message broker functionality. We also investigated the performance of a suite of techniques to share for post-processing among queries. The results can be summarized as follows:

- PathSharing-FWR when combined with optimizations based on queries and DTD usually provides the best performance. This approach is the most aggressive of the three in terms of path sharing.
- Without optimizations, however, PathSharing-FWR performs quite poorly, due to high post-processing costs.
- Optimization of query plans using query information improves the performance of all alternatives, and the addition of DTD-based optimizations improves them further.
- For non-recursive data, DTD-based optimizations can remove all DupElim and hash-based operators. Recursive data, however, stresses the post-processing of queries containing "//" axes and limits the effectiveness of optimizations.
- Finally, experiments on extending PathSharing-FWR with shared postprocessing showed excellent scalability improvements, allowing the processing of 100,000 queries in less than half a second.

7. Related work

Our work on XML message brokering is related to Continuous Query (CQ) processing, publish/subscribe, XML filtering, XML stream processing, and multi-query processing.

CQ systems support shared processing of multiple standing queries over (typically non-XML) data streams. The concept of expression signatures was introduced by *TriggerMan* [15]. Using such expression signatures, *NiagaraCQ* [4][5] incrementally groups query plans, and *CACQ* [19] supports the sharing of physical operators among tuples. *OpenCQ* [16] uses grouped triggers for CQ condition checking. Our techniques for sharing post-processing, though similar in spirit to those used in some of these systems, are developed particularly for XQuery processing.

Publish/subscribe systems, e.g. *Le Subscribe* [12] and *Xlyeme* [21], match incoming events with a very large number of subscriptions each of which is typically a set of conjunctive predicates. These systems use restricted query languages and data structures tailored to the query languages to achieve high system throughput.

A number of XML filtering systems have been developed to efficiently match a large set of path queries with streaming documents. *XFilter* [1] builds a *Finite State Machine* (FSM) for each path query and employs a query index on all the FSMs to process all queries simultaneously. *YFilter* [8][11] has been described in section 2.3. *XTrie* [7] supports shared processing of the common sub-strings of path expressions which only contain parent-child operators. In [13], all path expressions are combined into a single DFA, resulting in good performance but with significant limitations on the flexibility of the approach. *YFilter* and *Index-Filter* are compared through a detailed performance study in [3]. *Match-Maker* [17] supports shared tree pattern matching using disk-resident indexes on the tree patterns, with limited filtering performance. *XPush* [14] builds a pushdown automaton for a subset of tree-pattern queries, sharing both path navigation and predicate evaluation among them. It requires some pre-computation of the machine to achieve good performance. As stated previously, these systems only provide the lowest level of functionality required by XML message brokers.

In the context of XML stream processing, some other recent work uses transducer based mechanisms for processing path expressions with qualifiers [22] or XQuery containing FLWR expressions [18]. These approaches, however, are developed for single query processing.

Multi-query processing [23][24][27] considers small numbers of queries (e.g., 10's) and uses heuristics to approximate the optimal global plan. In contrast, high-volume XML message brokering needs to handle sets of queries orders of magnitude larger in a dynamic environment. Thus, scalability of the approach and incremental construction of query plans are the major concerns unique to our work.

8. Conclusions

In this paper, we developed shared processing to support the customization of output in the context of high-capacity XML message brokering. We compared three different ways of exploiting a shared path matching engine for this purpose. Our results show that the most aggressive of the three in terms of path sharing performs best, when combined with optimizations based on the queries and DTD. Moreover, when post-processing is also shared among queries, excellent scalability can be achieved.

We plan to extend our work in the following directions. First, we plan to support additional features such as ordering and aggregation in result customization. Second, it would be useful to investigate customization solutions based on shared tree pattern matching, once such technology is sufficiently developed. Finally, we will address the third major component of the XML message broker through the investigation of content-based routing in an overlay network deployment.

References

[1] M. Altinel, M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, Sep. 2000.

[2] S. Boag, D. Chamberlin, et al. XQuery 1.0: An XML query language. W3C Working Draft. http://www.w3.org/ TR/xquery, 2002.

[3] N. Bruno, L. Gravano, et al. Navigation- vs. index-based XML multi-query processing. ICDE 2003, to appear.

[4] J. Chen, D. DeWitt, et al. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, Feb. 2002.

[5] J. Chen, D. Dewitt, et al. NiagaraCQ: A scalable continuous query system for Internet databases. In *SIGMOD*, May 2000.

[6] D. Chamberlin, P. Fankhauser, et al. XML query use cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-use-cases/, 2002.

[7] C. Chan, P. Felber, et al. Efficient filtering of XML documents with XPath expressions. In *ICDE*, Feb. 2002.

[8] Y. Diao, P. Fischer, et al. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, Feb. 2002.

[9] Y. Diao, M. Franklin. Query processing for high-volume XML message brokering. *Technical report*, UCB//CSD-03-1228, 2003.

[10] A. L. Diaz, D. Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, Sep. 1999.

[11] Y. Diao, M. Altinel, et al. Path matching and predicate evaluation for high-performance XML filtering. http://www.cs.berkeley.edu/~diaoyl, 2002.

[12] F. Fabret, H. Jacobsen, et al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[13] T. J. Green, G. Miklau, et al. Processing XML streams with deterministic Automata. In *ICDT*, Jan. 2003.

[14] A. K. Gupta, D. Suciu. Streaming processing of XPath queries with predicates. In *SIGMOD*, June 2003.

[15] E. N. Hanson, C. Carnes, et al. Scalable trigger processing. In *ICDE*, March 1999.

[16] L. Liu, C. Pu, et al. Continual queries for Internet scale event-driven information delivery. *IEEE TKDE* 11(4), Jul. 1999.

[17] L. V.S. Lakshmanan, P. Sailaja. On efficient matching of streaming XML documents and queries. In *EDBT*, March 2002.

[18] B. Ludascher, P. Mukhopadhyay, Y. Parakonstantinou. A transducer-based XML query processing. In *VLDB*, Aug. 2002.

[19] S. Madden, M. Shah, et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, Jun. 2002.

[20] Microsoft BizTalk Server 2002. http://www.microsoft.com/biztalk.

[21] B. Nguyen, S. Abiteboul, et al. Monitoring XML data on the Web. In *SIGMOD*, May 2001.

[22] D. Olteanu, T. Kiesling, F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *ICDE*, 2003.

[23] A. Rosenthal, U.S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *VLDB*, Sep. 1988.

[24] P. Roy, S. Seshadri, et al. Efficient and extensible algorithms for multi-query optimization. In *SIGMOD*, May 2000.

[25] Salerio e2e middleware. http://www.one-ten.com/middleware.html

[26] Sybase financial fushion message broker. http://www.sybase.com/products/internetappdevttools/financialfusionmessagebroker

[27] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, Vol 13, No. 1, March 1988, Pages 23-52.