02/18/03

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 or elements 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

XML continues to be a subject of intense activity throughout the computing industry. While many of the implications of XML in terms of data management are still the subject of debate, there is widespread agreement that XML is becoming the common *wire format* for data. In distributed environments including Web Services, data and application integration, and personalized content delivery, XML is the way that data to be exchanged is encoded.

In this emerging distributed infrastructure, XML *message brokers* will play a key role. Recently, there have been a number of systems developed for the more restricted problem of XML filtering [1][3][9][10][15][20]. These systems quickly match incoming XML data items to a large set of pre-indexed queries that represent the data interests of specific users, applications, or organizations. Due to the use of XML in this work, the queries typically involve *path expressions* that refer to the structure of the XML data items. The most efficient filtering systems use innovative path indexing techniques to narrow the set of possible matches, and exploit commonality among the queries to achieve scalability.

XML filtering is indeed a crucial component of a sophisticated XML message broker, but represents only the lowest level of functionality required. Such brokers must also be able to transform the XML data on a query-by-query basis, in order to provide customized data delivery and to enable cooperation among disparate, loosely coupled services and applications.

The work we describe in this paper is aimed at developing this next level of functionality. Our goal is to provide high-capacity brokering systems, capable of supporting potentially tens of thousands of simultaneous queries (i.e., subscriptions). Because shared path matching has been shown to be an efficient and scalable foundation for the current generation of XML filtering systems, we start with such an engine and develop alternatives for building customization functionality on top of it. In this work 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 [10][13] 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 addresses shared post-processing. Section 6 presents our experimental results. Section 7 outlines the extensions to support more general XQuery scenarios. Section 8 covers related work. Finally, Section 9 presents conclusions.

2. Background

2.1 Architectural Overview

XML message brokers [22][26][27] serve as central exchange points for messages sent between applications and/or users in loosely coupled, distributed environments. Three of the main functions of such brokers are: *filtering*, *transformation*, and *routing*. *Filtering* matches messages to predicates representing subscriptions or interest specifications. *Transformation* restructures matched messages according to recipient-specific requirements. *Routing* involves the transmission of the customized data to the recipients.

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. Incoming messages are processed on-the-fly as they arrive at the broker. These messages need not conform to DTDs (*Document Type Definitions*) but, as we describe later, such conformance can be exploited. Inside the broker, a message is parsed and represented as a node-labeled tree, which is a common view of an XML document disregarding IDREFs. The nodes in the tree are assigned integer identifiers according to a pre-order traversal of the tree. Incoming queries are also parsed for incremental addition to the shared query plan maintained by the Query Processor.

In this paper, we focus on the "Query Processor" component of the broker. The query processor consumes the parsed queries and messages, and produces output in an intermediate format. As we describe in the following section, this output 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 to the "message sender" for delivery.

In our system, as shown in Figure 1, the Query Processor consists of two main components: a shared path matching engine that produces "tuple streams" (which are described in Section 3) and a customization module that further processes these tuple streams 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. A number of groups have developed such engines, and have demonstrated excellent performance for large numbers of path expressions [1][3][9][10][13][15].

We chose to base our system on YFilter [10][13], a high-performance shared path matching engine that we built previously. YFilter employs a single *Non-Deterministic Finite Automaton* (NFA) to represent the full set of path expressions, sharing the common prefixes of the paths. For each incoming document, a pre-order traversal of the parsed representation of the document is used to drive the execution of the NFA and produce all path matches. A recent study by Bruno et al. in [3], showed that YFilter's approach is particularly effective for short documents and large numbers of queries; precisely the environment we anticipate for high-volume XML message brokers.

It should be noted that an alternative to using a path matching engine would be to extend a tree pattern matching approach (e.g., [4][18]) to support shared processing. Unlike path matching, however, there has been only limited prior work on *shared* tree pattern matching for simultaneously processing multiple queries. *MatchMaker* [20], one such system published recently, assumed a memory-constrained environment and reported execution times several orders of magnitude slower than state of the art path matching systems. *XTrie* [9] considers shared tree pattern matching in its design, but does not report on any experiments indicating its performance in doing so. A comparison of our solution with one based on shared tree pattern matching, once such technology is sufficiently developed, would be an interesting study, but is beyond the scope of this paper.

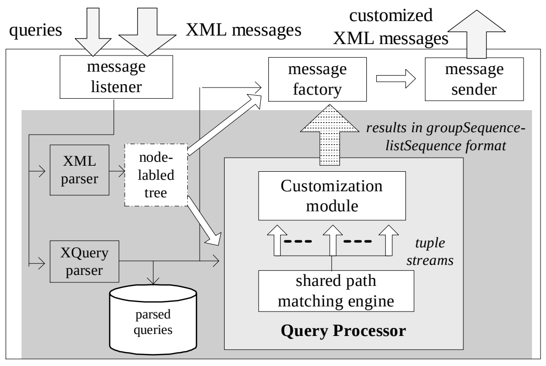

Figure 1: XML message broker architecture

2.2 Query and Output Specification

We focus on user query specifications written in a subset of XQuery [2] in which queries 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.

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*". Our current implementation does not support the *let* clause. We require that the predicate and return paths of a query be relative to the binding path of that query (i.e., that they are prefixed by the variable name used in the binding path). Each path expression consists of a sequence of location steps. We support location steps with parent-child "/" and ancestor-descendent "//" axes. All location steps can have element name tests. Furthermore, the binding paths may contain value-based predicates (i.e., comparisons to a constant value) addressing the attributes, text data, or the position of elements.[1]

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 representation of the XML message). The *where* clause, if present, restricts the set of bindings passed to the *return* clause. The *return* 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. Consider "**Query 1**" below, which is based on the *Book* DTD given in the XQuery use cases [8]:

<sections>

[1] The approaches we describe in this paper can be extended to support more general XQuery scenarios, which will be discussed in Section 7.

```
{
  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 document 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 document.

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:

```
section1:  {title11}, {figure12, figure13, figure14}
section2:  {title21}, {figure22, figure23}
….
```

where *section*$_i$ represents a group, and the numbering 1, 2, … 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.

2.3 Problem Statement

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.

This problem differs from previous work on XML path matching for two main reasons. First, the elements that are extracted by the *return* clause may be different than those used to match the message to the query (i.e., those identified by the *binding* and *predicate* paths). Second, *ordering* is an essential concern when extracting these elements for output. These two considerations add significant complexity to query processing.

Our solution also differs from work on XQuery query processing [4][18][29] in that it simultaneously processes a large set of queries — potentially tens of thousands or more. The sheer magnitude of the number of queries defeats any approach that processes each query individually. Rather, the key to performance and scalability in this environment is to exploit shared processing across queries.

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.

The approaches are described in the context of the particular *output format* provided by YFilter. YFilter delivers its output as streams of "tuples", one stream for each path expression matched by a document. A tuple contains a list of document node identifiers, where each identifier represents a binding for its corresponding location step in the matched path expression. For a given stream (i.e., for any particular matched path expression) the tuples are produced according to a pre-order traversal of the document. The important effect of this (which we exploit in our processing algorithms) is that since node identifiers are also assigned according to a pre-order traversal of the document, the tuples in a given stream are produced such that the node identifiers in the last (i.e. rightmost) position appear in monotonically increasing order. Examples of this output format are provided in the subsequent sections.

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 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; 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 binding path from each query into the engine. Then, during the processing of a document, 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>
```

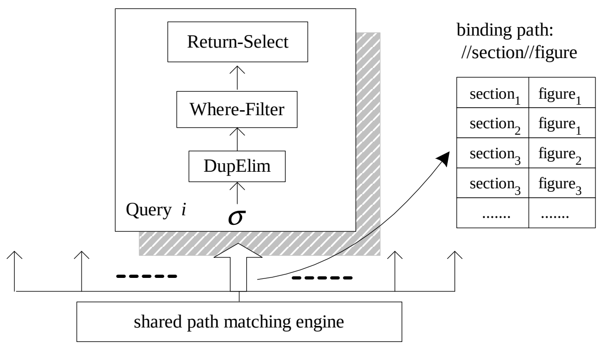

Figure 2: A query plan using PathSharing-F

Figure 2 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 tuples, one for each distinct path expression (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 tuples matching the binding path "//section//figure". In the following, we refer to the last field of these 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 value-based predicates attached to a binding path. The evaluation is done for each tuple by checking predicates against the nodes referenced by the tuple. Selection emits only those tuples for which all predicates evaluate to True.

Duplicate Elimination (DupElim). Duplicates can arise when multiple 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 tuples for the binding path. The first tuple corresponds to "<section id=1> <figure>" and the second to "<section id=2> <figure>". These two tuples reference the same *BoundNode* so the second tuple 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 earlier in this section, 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 tuples referencing the same *BoundNode* will pass the selection.

Where-Filter. This operator evaluates the *where* predicates on each tuple until a predicate evaluates to False or the entire *where* clause evaluates to True. Tuples in the latter case are emitted. For each tuple, a predicate path is evaluated with a tree search routine that uses a depth-first search in the sub-tree of the document rooted at the *BoundNode* of the 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 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 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 document.

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 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>
                { $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 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 3. 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 predicate paths. The tuples resulting from the matching process are piped to a Return-Select operator 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

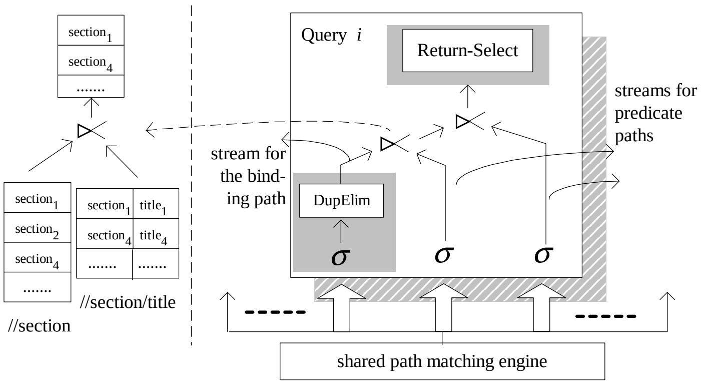

Figure 3: A query plan using PathSharing-FW

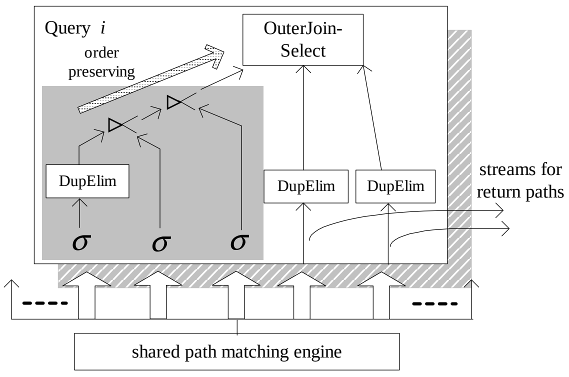

Figure 4: A query plan using PathSharing-FWR

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 3 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 tuple corresponding to "section_2 figure_1 title_1" followed by a 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*.

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*. OuterJoin-Select takes as its leftmost input, the stream resulting from the semijoins of the *PathSharing-FW* approach, and performs left outer joins on the *binding field* with each of the return path streams. It 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. The pseudo-code for OuterJoin-Select is provided in Appendix A.

PathSharing-FWR is shown in Figure 4. DupElim operators are required on each of the return path streams to prevent duplicate results from being generated by OuterJoin-Select. DupElim is performed on the combination of the *binding field* and the last field of the 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 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 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 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 documents. 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 Appendix B. Consider a path expression *p* of *m* location steps, and the stream of 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 tuples matching *p* when the 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 tuples matching *p* when the 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 tuples matching *p* when the 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 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 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 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 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 6.2.

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, 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 [5][6][17][19][21]. 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 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 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 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 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 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 [5] 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 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 [6][5][17][21]. 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

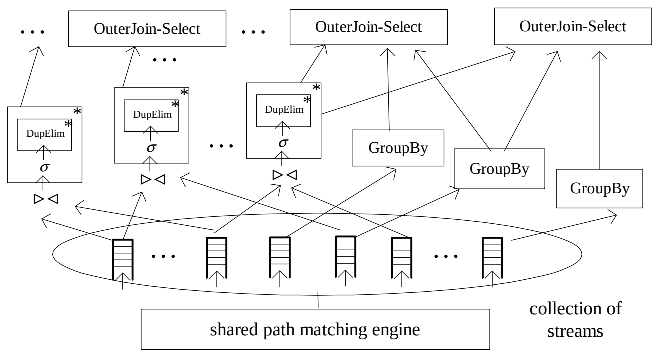

Figure 5: Shared post-processing example

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 tuples containing the same node id in the *binding field*. Fortunately, shared joins preserve 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 5. 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* [5][6] 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* [21] 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 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 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 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. After describing the experimental setup, we begin by comparing 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 [12], 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 three shown in Table 1. *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 2. 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.

Parameter	Value used	Description
DocDepth	4, 5	The maximum depth of XML documents
MaxRepeats	3 - 20	The maximum number of repeats of an element inside its parent element.
MaxValue	10 - 100	The maximum number of values an element or an attribute of an element can take

Table 1: Workload parameters for document generation

Parameter	Value used	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 2: Workload parameters for query generation

We report on experiments with two DTDs: the *Bib* and *Book* DTDs from the XQuery use cases[8]. 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.

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 6 (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 6, 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 3. 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.

The reduced cost of the three operators is further explained by the change in the resulting query plans, as shown in Table 4. The improvement of DupElim arises be-cause fewer such operators are needed with better optimiza-tion. 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	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 3: Costs (ms) of operators (PathSharing-FWR, *Bib*)

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 4: Profile for 5000 queries (PathSharing-FWR, *Bib*)

The above results demonstrate the effectiveness of the optimization techniques. In conjunction with these techni-ques, PathSharing-FWR provides significantly better perfor-mance than the other two alternatives, despite its more complicated 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 7 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 evalua-tion. 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 PathShar-ing-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 8 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 Path Sharing-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, Outer Join-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.

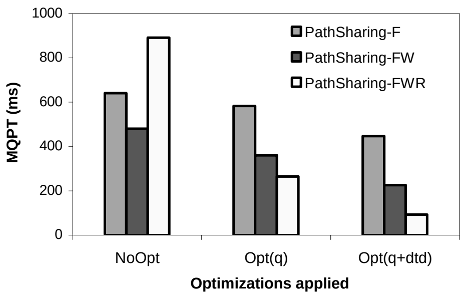

Figure 6: MQPT of three alternatives (Bib, Q=5000, PP=1, RP=2, DSProb=0.2)

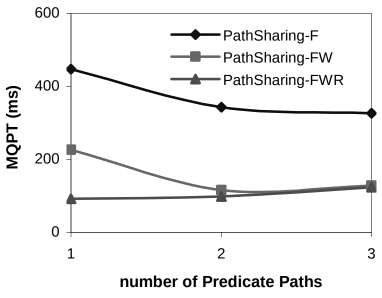

Figure 7: Varying PP (Bib, Q=5000, RP=2, DSProb=0.2, Opt(q+dtd))

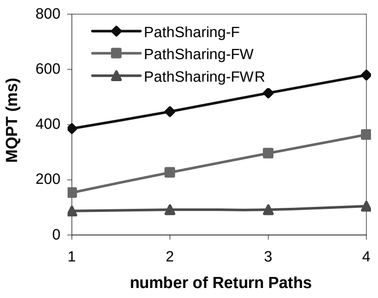

Figure 8: Varying RP (Bib, Q=5000, PP=1, DSProb=0.2, Opt(q+dtd))

Figure 9 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.

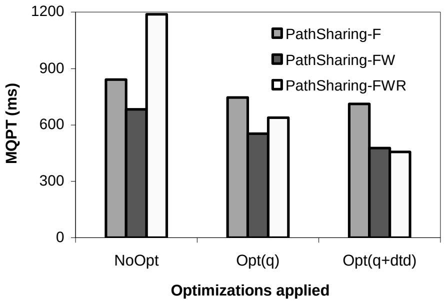

Figure 9: MQPT of three alternatives (Book, Q=10000, PP=1, RP=2, DSProb=0.2)

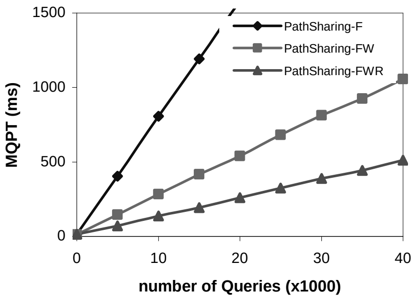

Figure 10: Varying Q (Bib, PP=1, RP=2, DSProb=0.2, Opt(q+dtd))

operators	PME	selection	DupElim	semijoin	outerjoin
No opt	36	101	560	225	287
Opt (q)	31	101	82	184	252
Opt (q+dtd)	21	93	30	137	163

Table 5: Costs (ms) of operators (PathSharing-FWR, *Book*)

operators	semijoin		outerJoin		DupElim
	#hash	# merge	#hash	#merge	#DupElim
No opt	10,000	0	10,000	0	30,000
Opt (q)	5963	4037	5963	4037	2833
Opt(q+dtd)	3968	6032	3968	6032	1500

Table 6: Profile for 10000 queries (PathSharing-FWR, *Book*)

The detailed performance of PathSharing-FWR is shown in Table 5. While optimization cuts down the cost of DupElim successfully, the costs of semijoin and OuterJoin-Select remain high. This is due to the recursive *section* elements and a fairly large number of "//" axes in queries (recall that *DS*=0.2 for each location step). In this situation, it is likely that tuples generated for predicate paths and return paths are not ordered by the *binding field*. Consequently, as shown in Table 6, many semijoins and outer joins must be hash based, even when the best optimization is applied.

To further investigate the impact of "//" axes in the presence of recursive elements, we ran another set of experiments varying *DSProb* from 0.05 to 0.4. The results can be summarized as follows: First, regardless of any optimizations applied, the MQPT for all alternatives increases with the *DSProb*. Second, the difference between Opt(q) and Opt (q+dtd) is more pronounced as *DSProb* is increased. Recall that Opt(q) only checks how many times the "//" axis appears in path expressions. In contrast, Opt(q+dtd) also checks if an element after "//" is allowed to be recursive, and thus can be more effective. Finally, with a very large *DSProb* value, even Opt(q+dtd) has a limited effect. As a result, PathSharing-FWR loses its performance gain over PathSharing-FW, as it requires more DupElims and hash based outer joins, which offsets the benefit of shared matching of return paths. For example, with *DSProb* = 0.4 (a very high value), PathSharing-FW outperforms PathSharing-FWR slightly (by about 4%).

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 10 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 7 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 7: 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 11 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 8 shows the cost breakdown of PlanSharing. A comparison with Table 5 provides insight into the reduction of the overall cost, which results from four major factors:

Q	10,000	20,000	30,000	40,000	50,000
(Unique plans)	**(8,232)**	**(16,482)**	**(24,576)**	**(32,736)**	**(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 8: Costs (ms) as *Q* varies - PlanSharing (*Book* 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

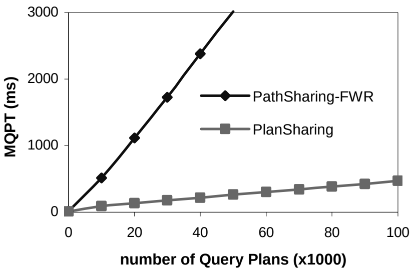

Figure 11: Varying number of unique query plans (Book, PP =1, RP=2, DSProb=0.2, Opt(q+dtd))

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.

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 sumarized 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 post-processing showed excellent scalability improvements, allowing the processing of 100,000 queries in less than half a second.

7. Extensions

The approaches developed in this paper can be extended to support more general XQuery scenarios. The extensibility of our approaches results from two factors. First, the output of YFilter, the path matching engine, carries sufficient information that enables all kinds of post-processing. Second, in all of our approaches, the post-processing for each query is modularized using a set of operators. The benefit of this modular design is that extending the current functionality can be achieved by modifying individual operators or adding more operators in each query plan for post-processing. In this section, we present how the techniques based on PathSharing-FWR can be extended to support a broader subset of the XQuery language.

First, we propose to modify individual operators, i.e. semijoin and OuterJoin-Select, to support more general path expressions in *for*, *where* and *return* clauses, as described below.

Nested paths in binding paths. Nested path expressions can appear in a binding path as predicates specifying additional structural constraints. In our work, these paths can be treated in a similar way as predicate paths in a *where* clause. That is, they are first matched by the path matching engine and their corresponding tuple streams are then matched with the binding path stream using semijoin operators. The major difference is that these semijoins may perform on tuple fields, called *join field*, other than the *binding field* (i.e., the last field of the binding path tuples). Note that in this case, DupElim on the binding path stream need to be delayed until after all semijoins in a query plan. Optimization of a semijoin also relies on Claim 4 or 5 to identify the order on the join field in the right stream of this operator. Semijoins with the same signature are also shared using *Selection-Pullup*. As a special case, when a nested path is an absolute path, a semijoin operator for this path can be replaced by a simple check on if the right input stream is empty.

Relative predicate paths. Predicate paths that are relative to any location step in the binding path, can be written using a prefix containing the variable name given in the *for* clause followed by one or more parent axes "..". These predicate paths are semantically identical to nested paths in the binding path. For example, "*for* $a in document (…)//book/author *where* $a/../title='XML' return $a" is the same as "for $a in document(…)//book[title = 'XML']/author *return* $a". Thus, techniques for nested paths described above can be applied to these predicate paths.

Wildcard operators. In our implementation, we fully support matching of paths expressions containing wildcard operators '*'. '*' operators, however, bring additional complexity to query optimization, which we have not discussed in the previous sections. The complexity results from lack of information on the elements that can match a location step containing a '*' operator. In this situation, it becomes harder to analyze what properties of location steps in such a path expression can cause duplicates or affect the tuple ordering. Separate sufficient conditions need to be derived to remove DupElim or improve the implementation of operators, when '*' operators appear in path expressions.

Other axes and node tests. In our implementation, we also allow return paths to contain attribute axes, and functions text() and position() at the end of the paths. The only difference is that now, instead of placing node identifiers in the result list for such a return path, the values selected from an attribute, the data, or the position of those nodes are placed into the list. This support is enabled by a simply extension of the OuterJoin-Select operators.

After describing the extensions to support more general path expressions, we then present how to incorporate more functionality of XQuery, namely, *let clause* and *order by clause*, into our work. The extended functionality is achieved by adding additional operators or sub-query plans at appropriate places in the post-processing query plan.

Let clause. A *let* clause is used to specify aggregation based on each variable binding of the *for* clause. We currently only consider the case that *for* and *let* clauses address path expressions in the same document. With this restriction, we can build a sub-query plan that takes the binding path stream and the tuple streams for all matched path expressions in the *let* clause, and produces tuples that are pairs of bindings for the variable in the *for* clause and the variable in the *let* clause. In each tuple in the output, the binding of the *for* clause contains a single node id, while the binding of the *let* clause contains a sequence of node ids that are aggregated based on the *for* clause binding. This sub-query plan is applied to the binding path stream right before the tree of semi-join operators.

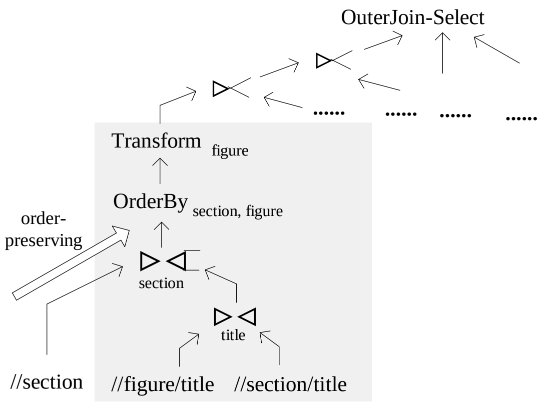

Figure 12: A query plan fragment processing a *let* clause

To illustrate such a sub-query plan, consider a *for* clause and a *let* clause below. A query plan fragment handling *the* let clause is shown in the shaded area in Figure 12.

```
…
  for      $s in document("doc.xml")//section
  let      $f := document("doc.xml")//figure[title=$s/title]
  where    …
  return   …
…
```

As this query plan shows, two path expressions in the *let* clause have been turned into paths "//figure/title" and "//section/title". When the tuple streams for these two paths are returned from the path matching engine, they are joined on the "title" field. This join operator is merge-based. The subsequent operator is a left outerjoin that takes the binding path stream as the left input, and the output of the join as the right input. The outerjion is performed on the "section" field and preserves the order of the left input stream. The output of this outerjoin is a stream of tuples each of which references a "section" node and may reference a "figure" node with identical "title" information as the "section" node (if the "figure" field in the tuple is not null). This outerjoin uses hashing on the right stream. The output of this operator is directed to an OrderBy operator that orders tuples by the "section" field and by the "figure" field. Since all tuples are already sorted by the "section" field, OrderBy simply takes adjacent tuples that reference the same "section" node, and sorts them by the "figure" field. Finally, the transform operator turns tuples referencing the same "section" node into a single tuple by collapsing the contents of their "figure" field into a sequence-valued field.

Several things need to be noted. When the binding path directly joins with a path expression in the *let* clause, such a sub-query plan can be simplified by avoiding the bottom join operator. When the *where* clause in a query does not address the variable binding using the *let* clause, the semijoin operators can be pushed down the sub-query plan processing the *let* clause. Furthermore, if the variable binding of the *let* clause is only used for the display in the output, this portion of query plan can even be partly merged with the OuterJoin-Select.

Order by clause. An order by clause imposes a new ordering of the binding path tuples. If these tuples are to be ordered by some attribute value or the data of the nodes referenced in one tuple field, we place an explicit OrderBy operator before the OuterJoin-select operator. If these tuples are to be ordered by addressing another path, we introduce a variant of join between the stream for the order-by path and the stream for the binding path, which outputs the binding path tuples in order of the tuple stream for the order-by path.

New techniques for optimizing the post-processing need to be explored. Some ideas of optimization for processing a *let* clause are mentions above. More efforts are needed to derive the rules guiding the optimizations. Techniques for shared query processing also need to be extended to support the new language aspects. We will investigate these issues in our future work.

8. Related work

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

CQ systems support shared processing of multiple standing queries. The concept of expression signatures was introduced by TriggerMan [17]. *NiagaraCQ* [6][5] uses such expression signatures to incrementally group query plans. The project also investigated tradeoffs among different selection placement strategies. *OpenCQ* [19] uses grouped triggers for CQ condition checking. *CACQ* [21] further combines adaptivity and grouping for CQ and supports sharing physical operators among tuples by attaching states to tuples. *PSoup* [7] combines the processing of ad-hoc queries and continuous queries symmetrically.

Among publish/subscribe systems, *Xlyeme* [23] uses a hierarchy of hash tables to index sets of events required in monitoring queries. *Le* Subscribe [14] efficiently matches incoming events, i.e., a set of attribute value pairs, with subscriptions by indexing predicates and clustering subscriptions using their common predicates. A common feature of these systems is the use of restricted profile languages and data structures tailored to them for high system throughput.

A number of XML filtering systems have been developed to efficiently match XPath 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* [10][13] has been described in section 2.1. *XTrie* [9] indexes sub-strings of path expressions that only contain parent-child operators, and shares the processing of the common sub-strings among queries using the index. In [15], 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]. To the best of our knowledge, *MatchMaker* [20] is the only published work reporting its performance on shared tree pattern matching. Using disk-resident indexes on pattern nodes and path operators, it labels document nodes with all matching queries. I/O invocations limited its matching efficiency. A pushdown automaton is proposed to process tree-pattern queries while sharing their common subexpressions [16]. Its algorithms, however, are not available at the moment when this paper is submitted.

In the context of XML query processing, efficient tree pattern matching has been recently studied. In [29], binary structure joins for single path operators are performed using an index on (*docID*, *startPos:endPos*, *level*) representations of elements and a multi-attribute merge join. This join algorithm was improved by using a stack mechanism to avoid the unnecessary inner rescanning in a merge join [18]. *PathStack* and *TwigStack* [4] further improve the join algorithms by evaluating *all* path operators in a path or a twig pattern. A stack for each pattern node is used to obtain compact encodings of the pattern matches. These techniques, developed for single query processing, have the potential of being leveraged in the multi-tree pattern matching scenario.

Multi-query processing [24][25][28] is marginally related to our work, since it identifies common subexpressions among queries and shares operators for these subexpressions. The work on MQP, however, considers a small number of queries (e.g., 10's), and uses heuristics to approximate the optimal global plan. In contrast, our work on high-volume XML message brokering handles a set of queries several orders of magnitude larger in a dynamic environment. Thus, scalability of the approach and incremental construction of query plans are the two major concerns.

9. 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, more functionality, e.g. aggregation and order by, will be supported in message customization. Second, efforts will be directed to building a message brokering system in the distributed environment which delivers customized messages using content-based routing. Last, as mentioned in the paper, it would be an interesting study to investigate shared query processing built on multiple tree pattern matching.

Acknowledgements

We would like to thank Joseph Hellerstein and Samuel Madden for their valuable comments on an earlier draft of this paper.

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] N. Bruno, N. Koudas, et al. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, Jun. 2002.

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

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

[7] S. Chandrasekaran, M. Franklin. Streaming queries over streaming data. In *VLDB*, Aug. 2002.
[8] D. Chamberlin, P. Fankhauser, et al. XML query use cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-use-cases/, 2002.
[9] C. Chan, P. Felber, et al. Efficient filtering of XML documents with XPath expressions. In *ICDE*, Feb. 2002.
[10] Y. Diao, P. Fischer, et al. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, Feb. 2002.
[11] Y. Diao, M. Franklin. Query processing for high-volume XML message brokering. *Technical report*, UCB//CSD-03-1228, 2003.
[12] A. L. Diaz, D. Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, Sep. 1999.
[13] Y. Diao, M. Altinel, et al. Path matching and predicate evaluation for high-performance XML filtering. http://www.cs.berkeley.edu/~diaoyl, 2002.
[14] F. Fabret, H. Jacobsen, et al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.
[15] T. J. Green, G. Miklau, et al. Processing XML streams with deterministic Automata. In *ICDT*, Jan. 2003.
[16] A. K. Gupta, D. Suciu. Streaming processing of XPath queries with predicates. SIGMOD 2003, to appear.
[17] E. N. Hanson, C. Carnes, et al. Scalable trigger processing. In *ICDE*, March 1999.
[18] S. Al-Khalifa, H. Jagadish, et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, Feb. 2002.
[19] L. Liu, C. Pu, et al. Continual queries for Internet scale event-driven information delivery. *IEEE TKDE* 11(4), Jul. 1999.
[20] L. V.S. Lakshmanan, P. Sailaja. On efficient matching of streaming XML documents and queries. In *EDBT*, March 2002.
[21] S. Madden, M. Shah, et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, Jun. 2002.
[22] Microsoft BizTalk Server 2002. http://www.microsoft.com/biztalk.
[23] B. Nguyen, S. Abiteboul, et al. Monitoring XML data on the Web. In *SIGMOD*, May 2001.
[24] A. Rosenthal, U.S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *VLDB*, Sep. 1988.
[25] P. Roy, S. Seshadri, et al. Efficient and extensible algorithms for multi-query optimization. In *SIGMOD,* May 2000.
[26] Salerio e2e middleware. http://www.one-ten.com/middleware.html
[27] Sybase financial fushion message broker. http://www.sybase.com/products/internetappdevttools/financialfusionmessagebroker
[28] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems,* Vol 13, No. 1, March 1988, Pages 23-52.
[29] C. Zhang, J. Naughton, et al. On supporting containment in relational database management systems. In *SIGMOD,* May 2001.

Appendix A: Pseudo-code for OuterJoin-Select

```
OuterJoin-Select
Input: leftStream from the semijoin tree,
        rightStream_1, …, rightStream_n for return path_1 to path_n
Output: selected nodes in the groupSequence-listSequence format

List groups = new List();
for the next tuple l from leftStream {
    List seq = new List();
    for each right stream rightStream_i {
        List list = new List();
        for the next tuple r from rightStream_i matching l
            append r.lastField() to list;
        append list to seq;
    }
    append seq to groups;
}
return groups;
```

Appendix B: Proof of claims

Consider a path expression *p* of *m* location steps, and the **stream** of tuples that match the path, with fields numbered 1..*m*.

Proposition 4.1: If duplicates occur in field *m* of tuples in the stream, then in the path expression *p* which contains *m* location steps, there exist *i* and *j*, $i < j \le m$, such that both location steps *i* and *j* contain a "//" axis and the element in location step *i* is on a loop in the DTD element graph.

Proof: Let t_1 and t_2 denote the two tuples containing the duplicates, and let n^* be the common node identifier in their last field *m*, as illustrated in Figure 12. Since t_1 and t_2 are distinct tuples, there must exist at least one field where the two tuples differ. Let field *i* be the first field where they differ, $1 \le i < m$. The shaded fields before field *i* in Figure 12 represent the same content in both tuples. The node identifiers in field *i* in two tuples are denoted as n_{i1} and n_{i2}. Since they are in the same field, we know that n_{i1} and n_{i2} match the same element, i.e. the element in location step *i* of the path. Also, because $i < m$, n_{i1} and n_{i2} matching field *i* are both ancestors of n^* matching field *m* in the document tree. Since all ancestors of n^* lie on a single document path, one of n_{i1} and n_{i2} must contain the other. This shows the element in location step *i* must be on a loop in the DTD graph.

We then claim the axis in location step *i* is "//". To see why, let us assume the axis is '/' instead. Then a node in field *i* must be a child of the node in field *i-1*. Since we know n_{i1} and n_{i2} contain each other, they must have different parent nodes in field *i*-1, which conflicts our choice of field *i* (recall field *i* is the first field where the two tuples differ). A special case is that field *i* is the first field. If the axis in the first location step is '/', then n_{i1} and n_{i2} both have to be the root node, which also conflicts our choice of field *i*.

In addition, let field *j* be the field immediately after the last field where the two tuples differ, $j <= m$. The shaded fields from field *j* to the end in Figure 12 represent the same content in both tuples. We claim the axis in location step *j* is also "//". Again assume the axis is '/' instead. Then a node in field *j* must be a child of the node in field *j-1*. Since two tuples have the same node n_j in field *j*, they must have the same parent node of n_j in field *j-1*, which conflicts our choice of field *j*.

Claims 1 and 2 follow immediately from this proposition.

Proof of Claim 3: If claim 1 or 2 claims that there are no duplicates for p_1, we know two matches of p_1 can not have the same node identifiers for the last location step of p_1. In addition, if claim 1 or 2 also claims that are no duplicates for p_2, then we know for any node that matches the last location step of p_1, in the sub-tree rooted at this node, there won't be matches of p_2 that have the same node identifier for p_2's last location step. Combining the above two facts shows that there will no duplicates in the tuples matching the original path when they are projected onto field *i* and *m*.

Proposition 4.2: If tuples in the stream matching *p* are not in increasing order when projected onto field *i*, $1 \le i < m$ then
(a) there exist two tuples whose node identifiers in field *i* identify two nodes that contain each other;
(b) there exists a field *j*, $j \le i$, s.t.
- the axis of location step *j* in the path expression is "//";

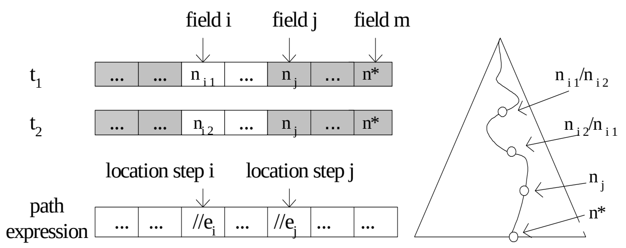

Figure 13: Two duplicate tuples, their path expression and the document tree

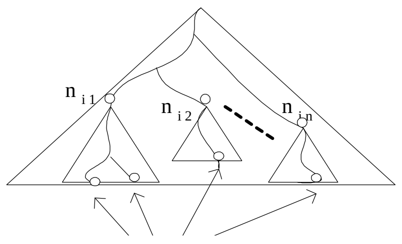

Figure 14: A document tree with no recursive nodes in field i.

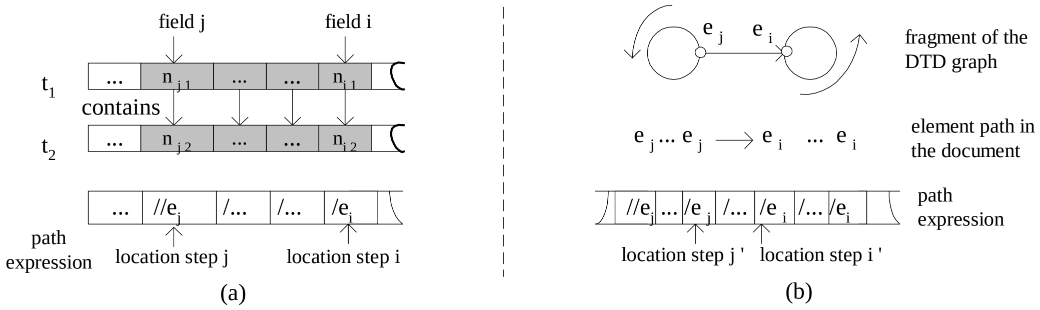

Figure 15: Two tuples with recurive nodes in field *i*, their path expression, related DTD graph, and the element path in the document

- the element of location step *j* is on a loop in the DTD element graph; and
- the element in location step *i* is on the same loop.

Proof. (a) Let us look at the nodes identified by field *i* of the tuples. Assume there do not exist two nodes that contain each other. Then the sub-trees rooted at these nodes are disjoint, as illustrated by nodes n_{ij} ($j \geq 1$) in Figure 13.

Also, location steps *i* to *m* are evaluated in these disjoint subtrees. Since the tuples are in increasing order in field *m*, we know that matches of the path expression are returned from one disjoint subtree to another in their document order. This implies that nodes for field *i* are also returned in increasing document order, which conflicts our assumption that tuples are not in increasing order in field *i*.

(b) Based on the existence of two tuples whose node identifiers in field *i* identify two nodes that contain each other, we move to investigate location step *i* in the path expression which is matched by these nodes. We already know that the element in this location step is on a loop in the DTD graph (otherwise the nodes matching the location step cannot contain each other.) What to be checked is the axis of the location step.

Case 1: Location step *i* contains a "//" axis. Then this is a special case of claim (b) with *j* = *i*.

Case 2: Location step *i* contains a "/" axis. Let t_1 and t_2 denote the two tuples, and n_{i1} and n_{i2} denote the nodes in their field *i*. Without loss of generality, assume n_{i1} contains n_{i2}. Since the axis of location step *i* is "/", the parent node of n_{i1} in t_1 must also contain that of n_{i2} in t_2. See Figure 14(a) for the illustration. Let field *j* be the last field before field *i* whose location step contains a "//" axis. Using the same argument, we know node $n_{j,1}$ in field *j* in t_1 also contain the node $n_{j,2}$ in field *j* in t_2. This implies the element in location step *j* is an element on a loop. Note that such a *j* location step containing "//" must exit, otherwise by using induction, we reveal conflicting facts that the first location step of the path contains a '/' axis but the two nodes in the first field of the two tuples contain each other.

Last let us check why the elements in location step *j* and location step *i* share one DTD element loop. Let e_j and e_i be the two elements in the two location steps. Assume that e_j and e_i are not on any common loop. Given the additional facts that (1) there is a path from e_j to e_i in the document tree (which is in the content of field *j* to *i* of either tuple) and (2) both e_j and e_i are elements on loops, we know that in the DTD graph, they are on different loops with one or more directed bridges going from the loop containing e_j to that containing e_i.

Assume location step *j'* and location step *i'* are such location steps in the path expression that $j \leq j' < i' \leq i$ and any location step between *j'* and *i'* do not contain element e_j or e_i. Then elements in location steps *j'* to *i'* define a unique bridge between the two loops in the DTD graph. Figure 14(b) illustrates the bridge determined by these location steps.

Then it is important to note that, any path from e_j to e_i through this bridge in a document tree must contain one and only one path fragment corresponding to crossing the bridge (note that it goes one direction). Since the content of field *j'* to field *i'* in t_1 matches location steps *j'* to *i'*, this content must be identical to that path fragment. Also, as the content of field *j'* to field *i'* in t_2 matches location steps *j'* to *i'* on the same document path as t_1, it also must correspond to the unique path fragment. This conflicts with our knowledge that the nodes in any field between *j* and *i* in the two tuples contain each other. So the assumption that e_j and e_i are not on any common loop is false.

Claims 4 and 5 follow immediately from this proposition.