Efficient XQuery Join Processing in Publish/Subscribe Systems

Ryan H. Choi[1,2] **Raymond K. Wong**[1,2]

[1] The University of New South Wales, Sydney, NSW, Australia
[2] National ICT Australia, Sydney, NSW, Australia
Email: {ryanc,wong}@cse.unsw.edu.au

Abstract

Efficient XML filtering has been a fundamental technique in recent Web service and XML publish/subscribe applications. In this paper, we consider the problem of filtering a continuous stream of XML data against a large number of XQuery queries that contain multiple inter-document value-based join operations in their `where` clauses. To perform efficient join operations, the path expressions from these queries are extracted and organized in a way that multiple path expressions can be joined simultaneously. The join operations are then pipelined to minimize the number of join operations and to share any intermediate join results as much as possible. Our system operates on top of many currently available XPath filtering engines as an add-on module to extend their features to support queries with join operations. Experiments show that our proposal is efficient and scalable.

Keywords: XML publish/subscribe, XML data stream, XML query processing

1 Introduction

XML has become the standard for data representation and exchange between large scale Web service applications on the Internet. We consider a Web service application that receives streams of XML messages from various data sources on the Internet, and forwards these messages to subscribed users or other applications. This type of application is called an XML publish/subscribe (pub/sub) system. One key feature of such an application is to support a large number of user subscriptions, and efficiently process XML messages coming from streams in real time. Furthermore, it is important that an XML pub/sub system is expressive enough to process complex subscriptions. Recently, there have been several research efforts on building scalable and expressive XML pub/sub systems (Chan et al. 2002, Diao et al. 2003, Gupta & Suciu 2003, Onizuka 2003, Rao & Moon 2004, Uchiyama et al. 2005, Kwon et al. 2005). In these systems, user subscriptions are expressed in XPath (Clark & DeRose 1999) queries. We use the term *queries* and *documents* to refer to user subscriptions and messages in this paper, respectively. Many previous works (Chan et al. 2002, Diao et al. 2003, Gupta & Suciu 2003, Onizuka 2003, Rao & Moon 2004, Uchiyama et al. 2005, Kwon et al. 2005) focus on how to efficiently report the set of matching query IDs for each streaming XML document. However, one feature that these previous works commonly lacks is to process queries that join multiple documents. In addition, these works are also limited in a way that, only the matching query IDs are reported—they cannot return the set of matching elements for each matching query. In this paper, we present how to process queries that contain inter-document value-based join operations efficiently. At the same time, we also present how to return all matching elements for each matching query.

Supporting queries that join multiple elements coming from multiple streams is important, since such technique allows us to deliver richer information than existing pub/sub systems. For example, a sudden price drop of a share can be detected, and a financial section of an online newspaper that contains news articles about the same share can be delivered by existing systems. However, users might be interested in what caused the sudden price drop of this share. To do that, existing systems require two queries from each user, one for detecting the price drop of the share, and the other is for selecting news articles about that share. Then, these two queries are postprocessed, and when there exists such matching news articles about that share, each subscribed user is notified. After that, the entire financial sections of the newspaper are optionally delivered. If a user prefers to receive only the related news articles about that share, this cannot be achieved using existing systems. This is because these systems are designed to be used as XML routers or brokers, where only the delivery of the complete documents to downstream routers are considered. Recent work by Hong et al. (2007) provides a partial solution to this problem. In conjunction with an XPath processor, it supports queries that join two documents. However, it is still not suitable for this situation, as it only reports matching query IDs, and can only forward the entire documents to users. Moreover, their join operations are limited in a way that only the leaves of a document can be joined. Furthermore, their technique is designed to process very small documents in size, which are not suitable for larger documents. On the other hand, our system provides a solution to this particular problem. Any nodes from documents can be joined with each other, and large documents in size can be efficiently processed. In addition, we return the complete set of matching elements for each matching query along with its ID, and they can be forwarded to users.

We consider the case where subscriptions are written in XQuery queries, and evaluate these queries against streaming XML documents whose sizes are small enough to fit into memory. In addition, the structures of streaming documents are already given to each pub/sub system. This is done during the initial handshake period between upstream and down-

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the 20th Australasian Database Conference (ADC 2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 92, Athman Bouguettaya and Xuemin Lin, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

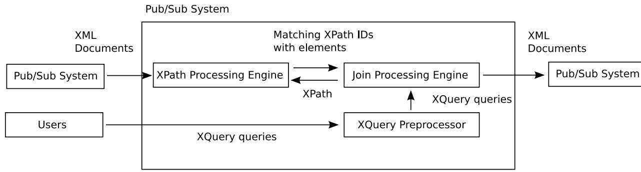


Figure 1: System Architecture

stream pub/sub systems. Once documents have been received from upstreaming pub/sub systems, n documents are stored temporarily in the buffer of size $b_{|n|}$, and are removed in a FIFO fashion when either the buffer becomes full, or after a certain period of time t_c. Join operations are performed between the documents in the buffer whose positions are in (1) between b_0 and b_i; or (2) their arrival times are in between t_i and t_j, where $t_i < t_j$. We currently support a subset of `for-where-return` clauses of XQuery. The description for each clause is as follows. A query contains multiple `for` clauses. A `for` clause expresses a node that is to be joined with a node in other `for` clause. The nodes in two `for` clauses can be from two different streams. A `where` clause expresses a list of conjunctive join operations between the nodes in `for` clauses. When a node is compared for equality, `text()` value is used if that node is a leaf node. If it is a non-leaf node, either `text()` or `string()` value can be used. A `return` clause expresses what node should be returned when a query matches the conditions in the `where` clause. Supporting `let` clauses is left as a part of our future work.

Figure 1 shows the architecture of our system. A pub/sub system sends an XML document of type t to multiple downstream pub/sub systems that have subscribed to receive documents of the same type. Upon receiving the document, a pub/sub system processes its subscriptions against it, and returns a set of subtrees in the document. This set of subtrees becomes a set of new documents, and are sent to downstream pub/sub systems that have subscribed to receive documents of the same type. Similar process is repeated at every pub/sub system.

Our pub/sub system consists of three major components. XPath Processing Engine is an XPath processor that can return matching query IDs along with all matching nodes for all matching queries for each streaming XML document. We use our previous work (Choi & Wong 2009) as XPath Processing Engine, as it satisfies the above conditions and performs efficiently. XQuery Preprocessor is used to normalize user subscriptions written in XQuery. Transformed queries are then sent to Join Processing Engine. Join Processing Engine processes queries with join operators, and it works in conjunction with XPath Processing Engine. When queries are received from XQuery Preprocessor, the XPath expressions in `for` clauses of XQuery queries are extracted, and the expressions with '`//`' and '`*`' are expanded using the document structures that have been registered initially. These expanded XPath expressions are then transferred to XPath Processing Engine. At the same time, Join Processing Engine organizes the expanded XPath expressions in a tree structure. When an XML document arrives, XPath Processing Engine returns the matching XPath IDs along with matching nodes, and they are transferred to Join Processing Engine. Once Join Processing Engine has processed join operations, all matching nodes for each matching XQuery query are reported, and they are sent to downstream pub/sub systems.

This paper is organized as follows. Section 2 presents related work in the area of processing XPath and XQuery queries on streaming XML documents. Section 3 describes our technique to process join operators in XQuery queries. Section 4 presents our experiment results for our technique. Finally, we conclude our paper in Section 5.

2 Related Work

There have been many works done in the context of processing XPath queries against streaming XML documents. Diao et al. (2004) introduce an overall architecture of a scalable pub/sub system. XTrie (Chan et al. 2002), YFilter (Diao et al. 2003), XPush (Gupta & Suciu 2003), Onizuka (2003), PRIX (Rao & Moon 2004), Uchiyama et al. (2005) and FiST (Kwon et al. 2005) present how a large number of XPath queries can be evaluated against a streaming XML document. While all these works return the set of matching XPath IDs for each document, they do not return matching nodes for matching queries, and they do not support queries with join operations. Furthermore, it is not clear how these features can be implemented.

Approaches by Olteanu et al. (2004), FluX (Koch et al. 2004), Li & Agrawal (2005), XFPro (Huo et al. 2006) and Gou & Chirkova (2007) process a single XPath and/or XQuery query against a large XML document in size. Barton et al. (2003) present how a single XPath query with both forward and backward axes can be processed for each streaming document. Chen et al. (2006) use TwigM, which is an extension of the multi-stack approach in TwigStack (Bruno et al. 2002), for a compact representation of candidate elements during query processing to decrease the overall memory usage. However, non of these works support queries with join operations.

Diao & Franklin (2003) extend YFilter such that it can return matching elements for matching queries. In their system, only XQuery queries with conjunctive predicates of the form $e_1/text() = constant$ are supported, whereas we support conjunctive predicates of the form $e_1/text() = e_2/text()$ in `where` clauses, where e_1 and e_2 can be from two different documents. XSQ (Peng & Chawathe 2005) builds an NFA for an XPath query that contains nested paths, and maintains its own buffer to store potential matching elements. This allows them to return all matching elements for a given query. However, only one query can be processed at a time. The work by Hong et al. (2007) is the most related work to our paper. It uses a customized query language to define queries with join operators between elements. In its compilation phase, the system uses Query Templates to group queries in a way that, each group contains queries with similar join statements. In the runtime, it uses a relational database to join elements. Similar to our system, it works with an existing XPath processor as an add-on module to support queries with join operators. However, their template-based approach only works for the queries that join two leaves of documents, and unlike ours, queries that join non-leaves of documents are not supported. In addition, their approach is designed to process relatively very small documents in size (i.e., documents with 3 levels deep with 16 leaves as shown in their experiments). This is because the number of Query Templates rapidly increases as the number of leaves per document increases. Due to the same reason, their approach does not scale for the queries that join elements from multiple documents. Similar patterns are observed when the number of join operations per query increases. In our experiments, we used documents, each of which contained a few hundred leaves, and we could still provide better

performance. Finally, unlike ours, they only report matching query IDs, and cannot return matching elements for each matching query for each document. This feature is important in Web service applications, as it allows them to forward only the parts of documents in which the subscribers are interested.

There are some works that use XML algebras to optimize XPath and XQuery processing. Barta et al. (2005) use document summaries to calculate heuristics and statistics, and process (nested) XPath queries with that information. Gottlob et al. (2005) define and evaluate a subset of XPath queries called Core XPath, and they can process queries with aggregates and user defined functions. However, the approaches in both works are not suitable for streaming documents, as they need multiple passes of documents. In addition, they are designed to process a small set of queries against a large XML document in size. Nevertheless, they have introduced several optimization techniques that can be integrated to our system, and such work is left as a part of our future work.

Unlike all pub/sub systems above, Boncz et al. (2006) and Grust et al. (2007) process XPath and XQuery queries by building processing engines that operate on top of existing RDBMS. Boncz et al. (2006) translate XQuery queries into basic relational algebras, apply query optimizations, and evaluate queries against a large set of documents. Grust et al. (2007) use Range Encodings to preserve the tree structure of XML, and use a B-Tree to optimize and evaluate XPath queries. Systems in this category support (almost) full features of XPath/XQuery including updates. However, they are not suitable for pub/sub systems, as they process queries individually.

3 Methods

This section presents our technique and consists of five parts. The first part shows how queries are normalized prior to processing them. The second part shows how queries are prepared. The third part presents a data structure that is used to process join operations, and how to build it. The fourth part presents how join operations are processed using the data structure against a streaming document. The last part presents an optimization technique, which improves the overall performance of join operations. We illustrate our technique with running examples.

3.1 Rewriting Queries

```
for $z in docType("nasa3")//ref
for $y in docType("nasa2")//ref//name
for $x in docType("nasa1")//journal/name
for $w in docType("nasa4")//astroObject/name
where $z//other/title=$y AND $y=$x
return $y
```

(a) Q1: A query written by a user

```
for $x in docType("nasa1")//journal/name
for $y in docType("nasa2")//ref//name
for $z in docType("nasa3")//ref//other/title
where $x=$y AND $y=$z
return $y
```

(b) Q1′: A rewritten query

Figure 2: Rewriting a query

Since XQuery queries could be poorly written or auto-generated, before queries are registered to the pub/sub system, they are preprocessed such that all queries are in the similar format. First, we remove `for` clauses whose variables do not participate in neither `where` nor `return` clauses. Second, we rewrite any join operations and their path expressions in `for` clauses in a way that join operations in `where` clauses contain only variables without any path expressions. Third, we rearrange `for` clauses according to the names of document types. Lastly, a constraint table similar to Diao et al. (2003) is built using value constraints in `where` clauses. The main idea is to process constraints after finding structurally matching queries.

Streaming documents have the same document type if they share the same document structure, and are coming from the same stream. To obtain documents of the same type, we use a user defined function `docType()` in our queries. Figure 2 shows an example of how our system rewrites a query in another format. In this example, the `for` clause with `w` variable is removed, and both `$z//other/name` and `//ref` in `where` and `for` clauses are rewritten, respectively. Finally, `for` clauses are rearranged in ascending order of the types of NASA documents.

3.2 Preparing Queries

```
for $x in docType("nasa1")//src//*/year
for $y in docType("nasa2")//history//year
for $z in docType("nasa3")//revisions//yr
where $x=$y AND $y=$z AND $x/text()="1990"
return $x
```

(a) Q2

```
for $x in docType("nasa1")//src//*/year
for $y in docType("nasa2")//history//year
for $z in docType("nasa3")//journal//*/yr
where $x=$y AND $y=$z AND $x/text()="2000"
return $y
```

(b) Q3

```
for $x in docType("nasa1")//src//*/year
for $y in docType("nasa2")//history//year
where $x=$y
return $y
```

(c) Q4

Figure 3: Query examples

The rewritten queries are processed in order to compactly represent them in our system. Figure 3 shows some additional preprocessed queries that we use as running examples in this paper. The process is as follows. First, for each rewritten query, we extract all `for` clauses. For each path expression in a `for` clause, we assign a *PathID*, and store the document type *DocType* from which that path expression is extracted. For example, the path expression, `//src//*/year` and the document type, `nasa1` are extracted from the first `for` clause in Q2, and the path expression is given an $PathID = 2$. We repeat the process for all queries and collect all unique path expressions for each document type. These path expressions are then stored in a multi-hashtable.

Second, all paths with '`//`' or '`*`' are expanded so that we can efficiently evaluate them in a deterministic way in runtime (Onizuka 2003). They are expanded using a document structure of streaming documents, and we name such a structure *Structure Index*. Figure 4 shows an example of Structure Index. It is similar to DataGuide (Goldman & Widom 1997) and ViST (Wang et al. 2003), but Structure Index is used to extract data structures of documents in order to preprocess and expand queries, whereas DataGuide and ViST are used to index data to improve query processing. Structure Index is generated

```
 1: <dataset>
 2:   <history>
 3:     <revisions><year/></revisions>
 4:   </history>
 5:   <ref>
 6:     <other><name/></other>
 7:     <src>
 8:       <journal>
 9:         <date><year/></date>
10:         <name/>
11:       </journal>
12:     </src>
13:   </ref>
14: </dataset>
```

Figure 4: Structure Index

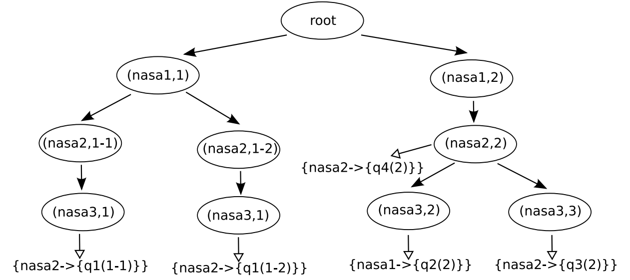


Figure 5: Join Tree

from a set of training documents that represent various structures of streaming documents. Table 1 shows expanded path expressions generated from the queries in Figure 2(b) and 3 using Structure Index shown in Figure 4. In this example, documents of type `nasa1` and `nasa2` share the same Structure Index. Structure Index for documents of type `nasa3` is similar, and can be obtained by replacing `year` and `name` nodes on line 3, 6, 9 and 10 with `yr` and `title`, respectively. It is omitted due to limited space.

Table 1: Expanded path expressions

DocType	PathID	XPath Path Expression
nasa1	1	/dataset/ref/src/journal/name
	2	/dataset/ref/src/date/year
nasa2	1-1	/dataset/ref/other/name
	1-2	/dataset/ref/src/journal/name
	2	/dataset/history/revisions/year
nasa3	1	/dataset/ref/other/title
	2	/dataset/history/revisions/yr
	3	/dataset/ref/src/journal/date/yr

Third, for each query, we replace all path expressions in `for` clauses with *PathID*s that are obtained from the multi-hashtable. A path expression is internally represented by a (*DocType*, *PathID*) pair. A query is represented by a list of (*DocType*, *PathID*) pairs along with the `return` path expression. Table 2 shows how queries in Figure 2(b) and 3 are stored in our system. This representation of queries are then used to build a tree called Join Tree. Lastly, the path expressions in Table 1 are now sent to an XPath processing engine as input at this stage.

Table 2: Query representations

QID	Path Representation	Return Path
1	(nasa1,1)(nasa2,1-1)(nasa3,1)	(nasa2,1-1)
1	(nasa1,1)(nasa2,1-2)(nasa3,1)	(nasa2,1-2)
2	(nasa1,2)(nasa2,2)(nasa3,2)	(nasa1,2)
3	(nasa1,2)(nasa2,2)(nasa3,3)	(nasa2,2)
4	(nasa1,2)(nasa2,2)	(nasa2,2)

3.3 Building Join Tree

The query representations generated in the previous section are rearranged in a tree structure in order to efficiently evaluate join operations. We name this tree structure *Join Tree*. Figure 5 shows an example of Join Tree built from the query representations shown in Table 2, and Algorithm 1 outlines how Join Tree is built.

Each node in Join Tree represents a path expression between the root and a node in an XML document of type *DocType*. For example, a node `(nasa3,2)` represents a path expression whose $PathID = 2$ in a document of type `nasa3`. In addition, a Join Tree node contains a mapping of $\{DocType \rightarrow \{QID(PathID)\}\}$. In Figure 5, empty mappings associated with Join Tree nodes are omitted for simplicity. The query ID in a mapping associated with a Join Tree node v indicates that, the query is evaluated by joining all path expressions represented by Join Tree nodes that are along the path between the root and v. In addition, this mapping is also used to find which path of a query should be processed next in order to return matching elements. For example, in Figure 5, $\{nasa1 \rightarrow \{q2(2)\}\}$ represents that if a join operation that joins `(nasa1,2)`, `(nasa2,2)` and `(nasa3,2)` nodes returns non-empty results, a query with $QID = 2$ is satisfied and should return the set of nodes expressed by the path with $PathID = 2$ from a document of type `nasa1`.

Join Tree is constructed as follows. We use the term *current context node* v_c to refer to a Join Tree node that is currently being examined and processed. We first set v_c to point to the root of Join Tree. For each query registered to the system, we retrieve the list of (*DocType*, *PathID*) pairs of the query from Table 2. For each pair from the list, we check whether a Join Tree node that represents the pair already exists as a child node of v_c. If it does, then that child node is retrieved. Otherwise, we create a new Join Tree node representing the pair, and it is added as a new child node. These steps are repeated for all (*DocType*, *PathID*) pairs. After processing the last pair, $\{DocType \rightarrow QID(PathID)\}$ mapping is created and stored in the last Join Tree node that we reach. Lastly, the above process is repeated for all queries.

3.4 Processing Join Tree

We first explain some additional data structures that we use when we process Join Tree, and explain how Join Tree is processed.

Algorithm 1 buildJoinTree(queries)

1: joinTreeRoot ← createRootNode()
2: **for** $q_i \in$ queries **do**
3: joinNode ← joinTreeRoot
4: docTypeToXPathId ← {}
5: **for** $p_i \in q_i$.getPaths() **do**
6: node ← joinNode.getChild(p_i.getDocType(), p_i.getXPathId())
7: **if** node = null **then**
8: node ← createNewChildNode(joinNode, p_i.getDocType(),p_i.getXPathId())
9: docTypeToXPathId ← docTypeToXPathId ∪ {(p_i.getDocType(), p_i.getXPathId()}
10: joinNode ← node
11: joinNode.addXQueryId(q_i.getId())
12: joinNode. setDocTypeToXPathId(docTypeToXPathId)
13: **return** joinTreeRoot

In order to distinguish different documents of the same type coming from the same stream, we assign each document a unique ID. We also keep a buffer of size b_n to store n documents of each type with path processing results. When a document of type $t(d)$ arrives, it is sent to an XPath processor as input, and the processor returns as output a set of $\{PathID \rightarrow \{v_1, \ldots, v_n\}\}$, where $\{v_1, \ldots, v_n\}$ represents a set of matching nodes for a path with *PathID*. This set is easily transformed to $\{(DocType, PathID) \rightarrow \{v_1, \ldots, v_n\}\}$, since we know the set is from a document of type $t(d)$. We use $P(d)$ to refer to this set of mappings for a document d. When a document d arrives, $P(d)$ is created and added to the buffer of type $t(d)$, and is removed from the buffer after the document has been processed. A $P(d)$ in a buffer is also sequentially removed in a FIFO fashion when the buffer becomes full or a $P(d)$ is stored for more than t_c units of time. We also maintain a global stack called Join Stack. An entry in Join Stack represents intermediate join results that have been produced so far by joining all matching nodes that are along the path between the root and a Join Tree node. Join Tree is traversed when a new document arrives from a stream and all buffers have at least one $P(d)$. Join Tree is traversed as follows. Algorithm 2 and 3 summarize the procedure.

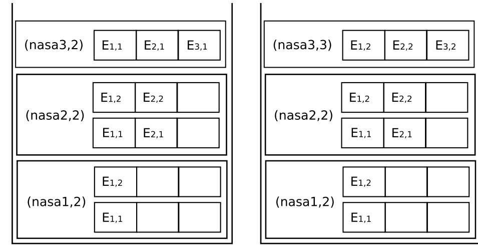
We initially set the root of Join Tree as the current context Join Tree node v_c. For each child of v_c, we obtain the type of the document that this Join Tree node represents. This can be done by checking (*DocType*, *PathID*) pair in the node. We then check whether *DocType* is the same as the type of the newly arrived document that is being processed. If *DocType* is the same, we retrieve the last $P(d)$ from the buffer that represents the document of type *DocType*. Otherwise, we obtain all $P(d)$s from the buffer of the same type. The reason for retrieving only the last $P(d)$ in the first case is to avoid duplicate join processing. For each $P(d)$ that we have obtained from the buffer, we search for the set of elements that match the path expression that v_c represents. This operation is efficiently done, as $P(d)$ is implemented in a hashtable. If an empty set is returned for the path expression, we stop traversing, and proceed to v_c's sibling nodes or backtrack. If a non-empty set is returned, we join the elements in the non-empty set with the sets of matching elements that we obtain from Join Stack, and push the newly generated sets of matching elements onto the stack. If a Join Tree node contains a mapping of $\{DocType \rightarrow QID(PathID)\}$ (see Figure 5), all query IDs are reported as matched.

Due to performance reasons, the matching elements are not immediately returned at this stage. Instead, a DOM representation of the currently streaming document is created, and each matching node from the DOM tree is decorated with the matching query IDs. We refer to such a DOM tree as a *document tree*. Matching elements are returned when Join Processing Engine finishes processing the current streaming document. Matching elements are returned in either the following two ways: (1) the decorated document tree is passed directly to an upper level application; or (2) the document tree is traversed once more to create $\{QID \rightarrow \{matching\text{-}elements\}\}$ mappings. It is possible that non-leaf nodes contain matching query IDs. In that case, subtrees rooted at these non-leaf nodes are returned as matching elements. Lastly, the above procedure is repeated for all child nodes of v_c.

Example 1. Figure 6(a) shows examples of matching elements $P(d)$ for some selected nodes in Join Tree, and Figure 6(b) and 6(c) show two instances of Join Stack after two Join Tree nodes, `(nasa3,2)` and `(nasa3,3)` in Figure 5 are traversed, respectively. In this example, Figure 6(a) shows that a Join Tree node `(nasa1,2)` has two sets of matching elements, namely $E_{1,1}$ and $E_{1,2}$, and each set contains the elements whose `text()` values are `1990` and `2000`, respectively, and all elements are from the document of type `nasa1`. Other nodes in Figure 6(a) are interpreted similarly. An entry in Join Stack contains a set of buffers (represented by a rectangle of three small boxes), and an entry in each buffer (represented by a small box) contains a set of matching elements so far. For example, the bottom entry in Figure 6(b) and 6(c) contains two buffers, each of which contains a set of matching elements denoted by $E_{1,1}$ and $E_{1,2}$. Other stack entries in Figure 6(b) and 6(c) are interpreted similarly. A buffer of sets of elements are used to represent intermediate join results. When we reach `(nasa3,2)` node in Join Tree in Figure 5, we find that the node contains $\{nasa1 \rightarrow \{q2(2)\}\}$. Since the top entry on Join Stack is not empty, we report $q2$ as matched. We retrieve matching elements $E_{1,1}$ from Figure 6(a), since $q2$ requires `text()='1990'`. The query ID=$q2$ is then added to the retrieved elements. Join Stack in Figure 6(c) is interpreted similarly. Figure 7 shows two document trees decorated with matching query IDs. Due to limited space, we use double dotted line notation between two nodes to represent an ancestor/descendant relationship in a document tree. This is similar to the double line notation commonly used in query trees.

Node	Matching Elements
(nasa1,2)	$E_{1,1} = \{e \in d(nasa1) \mid e/\texttt{text()} = \text{'1990'}\}$
	$E_{1,2} = \{e \in d(nasa1) \mid e/\texttt{text()} = \text{'2000'}\}$
(nasa2,2)	$E_{2,1} = \{e \in d(nasa2) \mid e/\texttt{text()} = \text{'1990'}\}$
	$E_{2,2} = \{e \in d(nasa2) \mid e/\texttt{text()} = \text{'2000'}\}$
(nasa3,2)	$E_{3,1} = \{e \in d(nasa3) \mid e/\texttt{text()} = \text{'1990'}\}$
(nasa3,3)	$E_{3,2} = \{e \in d(nasa3) \mid e/\texttt{text()} = \text{'2000'}\}$

(a) Matching elements



(b) At (nasa3,2) node (c) At (nasa3,3) node

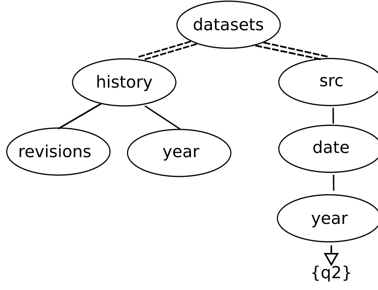
Figure 6: Examples of matching elements and the instances of Join Stack

Algorithm 2 processJoin(curDocType, joinTreeRoot)

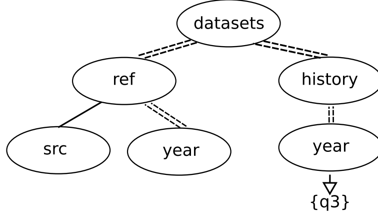
1: joinStack ← {}
2: **for** $\text{child}_i \in$ joinTreeRoot.getChildren() **do**
3: processJoinTree(curDocType, child_i, joinStack)

3.5 Optimizing Join Operations

We have presented how queries are processed using Join Tree. Up to this point, we have assumed that all publishers publish new documents at the same rate. In practice, however, it is most likely that some publishers produce documents more frequently than others, and therefore, some particular types of documents are sent more frequently than other types of documents. In addition, since we process documents



(a) `nasa1` document tree



(b) `nasa2` document tree

Figure 7: Two different types of document trees are shown. Each document tree shows which node should be returned as matching element for a query (Double dotted line represent ancestor/descendant relationship in a document tree similar to the double line notation commonly used in query trees).

sequentially as they arrive, we have found that there are some join operations that are common amongst the documents that are already in the buffer, and those join operations are repeatedly performed when a new document is arrived. In this section, we present an optimization technique that allows us to further reuse some intermediate join results that were already calculated when previous documents were processed. This technique considers how frequently a particular type of document arrives, and using that information, it determines the order in which join operations should be performed. The effect of this optimization is that, once more frequently arriving types of documents are identified, it improves runtime costs when these types of documents are processed. This is done by reusing some of the join results that were previously calculated.

In order to identify the rate of which a document arrives, we maintain simple statistics about how frequently each type of document arrives at our system while processing streaming documents. These statistics can also be reinforced when similar information is available from publishers. We refer to such statistics/information for each type of a document *Frequency Factor*.

After collecting Frequency Factor for each type of document, the nodes in Join Tree are rearranged in the order of Frequency Factor such that, the least frequent type of document is placed as child nodes of the root of Join Tree, and the most frequent type of document is placed as leaves of Join Tree. Figure 8 shows an example of Join Tree whose nodes are rearranged according to Frequency Factor. In this figure, `nasa2` has been identified as the most frequent type of document, and `nasa1` has been identified as the least frequent type of document.

Join Tree is now extended to include a buffer pool. A buffer pool contains bp_n number of partitions where $bp_n \leq$ the number of Join Tree nodes. The partitions in the buffer pool are allocated in runtime in a lazy way as needed. Each partition is used to store intermediate join results between a parent and a child node when a join operation is performed. Having a

Algorithm 3 processJoinTree(curDocType, node, joinStack)

```
1:  // curDocType: type of newly arrived document
2:  if curDocType = node.getDocType() then
3:     docsToProcess ←
              getLastDocFromBuffer(curDocType)
4:  else
5:     docsToProcess ←
              getDocsFromBuffer(curDocType)
6:  for doc_i ∈ docsToProcess do
7:     matchingNodes ←
           doc_i.getMatchingNodes(node.getXPathId())
8:     if matchingNodes ≠ {} then
9:        joinResultsSoFar ←
           joinStack.top().joinWith(curDocType,
                                     matchingNodes)
10:       joinStack.push(joinResultsSoFar)
11:       if joinStack.top() ≠ {} then
12:
13:          if node.containsMappings() then
14:             reportMatchingXQueries
                   (node.getMatchingXQueries())
15:          else
16:             for child_i ∈ node.getChildren() do
17:                processJoinTree(curDocType, child_i,
                                   joinStack)
18:       joinStack.pop()
```

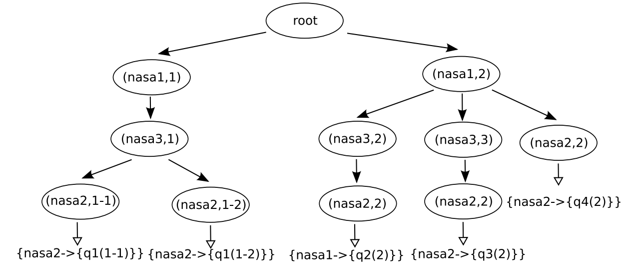


Figure 8: Rearranged Join Tree

buffer pool allows us to retrieve some previously calculated intermediate join results, which often prevent us from traversing the entire Join Tree from the root. Instead of traversing Join Tree when a new document arrives, we now select a group of small subtrees from Join Tree, and traverse these subtrees to perform join operations. A subtree is selected if the parent node of the subtree was previously visited, and *DocType* of the root of subtree is the same type as the streaming document. Previously visited nodes can be found in a hashtable. Checking the parent of a subtree is to ensure that, the join operations between the root of Join Tree and the parent node do not return an empty set. For each selected subtree, we do not traverse the path between the root of Join Tree and the root of subtree, if the join results for the nodes on the path between these two nodes can be retrieved from the buffer pool.

Example 2. If the currently streaming document is of type `nasa2`, and `(nasa2,1-1)` node from Figure 8 is the only node that has a parent node which was visited previously, we only perform one join operation between `(nasa3,1)` and `(nasa2,1-1)`, as the join operations between the root and `(nasa3,1)` can be retrieved from the buffer. Other join operations are not performed, as we know they will fail eventually.

Optimized join processing is described as follows. First, we find on which level l, the nodes that have the same *DocType* as the newly arrived document are

located in Join Tree, and their parent Join Tree nodes are extracted from Join Tree. Amongst these nodes, we select the nodes that were successfully traversed when a previous document had been processed. After that, we collect all child nodes from all such parent Join Tree nodes. These child nodes represent the roots of subtrees that must be traversed to perform join operations. One exception to the above selection process is when a parent node is the root of Join Tree. In that case, we simply choose all child Join Tree nodes of the root. This is the case where we do not benefit from our optimization technique, as we need to traverse Join Tree from the root. However, by carefully constructing Join Tree, this situation is minimized. Before any subtrees are traversed, all partitions in the buffer pool associated with the Join Tree nodes whose levels in Join Tree are $\geq l$ are cleared, as these partitions cannot be reused with these subtrees. To keep track of the last successfully traversed Join Tree nodes and to perform the above operations efficiently, we used an ordered hashtable of $\{DocType \rightarrow \{TraversedJoinTreeNodes\}\}$ in our implementation.

Second, for each subtree selected from the process above, we check whether the previous intermediate join results between the root of Join Tree and the parent of subtree can be found in the buffer pool. If so, that subtree is ready to be traversed. If not, we recursively check whether the intermediate join results between the root of Join Tree and any ancestor of the root of subtree can be found in the buffer pool. Once such a node is found, we start traversing back from that node to the root of the subtree. The buffer pool on the path are filled while traversing back. In the worst case, all nodes between the root of Join Tree and the root of subtree are traversed. Once the buffer pool for the root of subtree is filled up, we start traversing the subtree. While traversing a subtree, successfully traversed Join Tree nodes are marked, and our hashtable is updated accordingly in our implementation. With this approach, only a small set of subtrees are traversed, and by not traversing some upper parts of Join Tree, we skip performing some repeated join operations. Algorithm 4 summarizes the above procedures, and Algorithm 3 is modified in a way that the lines in Algorithm 5 are added to Line 12 in Algorithm 3.

Algorithm 4 processJoinOpt(curDocType, joinTreeNodeHashtable)

```
1:  subtrees ←
            joinTreeNodeHashtable.
                getJoinTreeNodes(curDocType)
2:  joinTreeNodeHashtable.
        clearJoinTreeNodes(curDocType)
3:  for subtree_i ∈ subtrees do
4:    parentOfSubtree ← subtree_i.getParent()
5:    bufferPool_i ←
          parentOfSubtree.getBufferPool()
6:    if bufferPool_i = {} then
7:      // recursively visit ancestor nodes
        // to find non empty bufferPool, and
        // start traversing back the same path
        // to fill up all ancestor bufferPools
8:      bufferPool_i ← calBufferPools(subtree_i)
9:    joinStack ← {}
10:   if parentOfSubtree ≠ rootOfJoinTree then
11:     joinStack.add(bufferPool_i)
12:   processJoinTree(curDocType,subtree,joinStack)
```

Algorithm 5 processJoinTreeMod(docType,node, joinStack)

```
1:  node.setBufferPool(joinStack.top())
2:  joinTreeNodeHashtable.
        addJoinTreeNodes(node.getDocType(), node)
```

4 Experiments

This section presents our experiment results in detail. All experiments were executed on a Core 2 Duo 2.33 GHz laptop with 2 GB ram running Mac OS X. The SAX parser we used was Xerces Java Parser 2.8.0 (The Apache XML Project 2007). Our approach was implemented in Java 1.5, and was built on top of our previous work (Choi & Wong 2009). This section consists of three parts. The first part shows how the documents used in the experiments were prepared. The second part shows how the queries used in the experiments were prepared. The last part presents how our approach performed under various conditions.

4.1 Document Preparation

We used NASA dataset obtained from UW Database Group (2002), and this dataset was preprocessed in a similar way as Kwon et al. (2005) to generate sample XML documents. In this setting, the dataset was split into many smaller documents, and randomly selected to form three sets of documents of size [10 KB, 20 KB), [20 KB, 30 KB) and [30 KB, 60 KB). We refer to each dataset as 10k, 20k and 30k respectively in this section. In addition, we removed all `text()` values from all non-leaf nodes to make $e_1/text() = e_2/text()$ always true for all non-leaf nodes with the same names. This is to increase the number of matching queries. Finally, each dataset generated above was added to a list, and this list was duplicated to create m identical lists of documents. These duplicated lists of documents were used to represent the documents coming from different streams.

For the experiments, the documents were streamed in two ways. First, a document from each list was randomly selected and streamed until all lists became empty. Second, all the first documents from the lists were removed and streamed sequentially until all lists became empty. This is to further increase the number of matching queries in the experiments, as many join operations in queries result in joining with the copies of the same documents. We will refer to each streaming method as a random and sequential streaming method, respectively.

4.2 Query Preparation

All queries we used in the experiments were generated as follows. First, the document structure of NASA dataset was extracted. Second, we randomly chose an element name e from the dataset, and selected n_p elements from the document structure whose names are the same as e. For each selected element, the path between the root and the element was scanned. While scanning the nodes on the path, we chose a node with a probability of 80%, as well as maintaining the element orders. For the case where a node was not chosen, we replaced `/` of the next node with `//`. Amongst the chosen nodes, we replaced nodes with `*` with a probability of 10%. Moreover, we replaced `/` with `//` with a probability of 10%. This step was repeated until n_p path expressions were created. Third, the path expressions were grouped such that each group contained the path expressions whose last elements had the same names. Fourth, j number

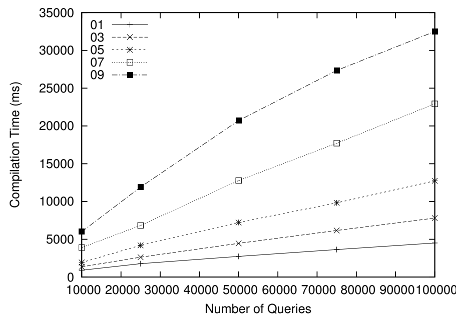


Figure 9: Compilation time vs. Number of queries that have various numbers of join operations

of path expressions were randomly selected from one path expression group, and combined to create a list of `for` clauses. A `where` clause was created by joining all path expressions found in the `for` clauses. A `return` clause was created by randomly choosing a path from a `for` clause. Lastly, the fourth step was repeated until n_q distinct queries were generated.

4.3 Results

Figure 9 shows compilation times when instances of Join Tree are built for various numbers of queries with various numbers of join operations. The compilation time increases as the number of queries increases, as the Join Tree building process is proportional to the number of queries to process. However, the rate of increase slowly drops as the number of queries increases. This is because, the number of new Join Tree nodes that must be created decreases, as nodes are shared amongst similar queries. Similarly, the compilation time increases as the number of join operations per query increases. This is because, as the depth of Join Tree increases, the number of Join Tree nodes that must be created and traversed increases.

In the following experiments, the Join Tree created in this phase was used, and all matching nodes for each query were computed as a part of join process. In addition, we used the following default values unless specified otherwise—the size of document buffer for each type of document was set to 1, each query had 5 join operations, 10k dataset was used, and documents were streamed in sequential order. Moreover, all experiments were executed with our join optimization. The processing times reported here are the average running times taken to perform join operations for each document. It also includes the times taken to calculate all matching elements for all queries. The processing times taken by an XPath processor were not included, and these can be found in our previous work. (Choi & Wong 2009)

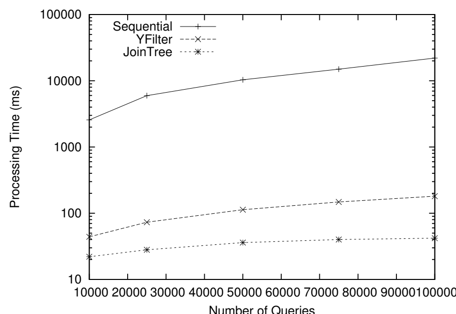


Figure 10: Processing time vs. Number of queries using various methods

Figure 10 shows how our Join Tree approach is performed against both naive Sequential Join approach and YFilter (Diao et al. 2003). In this experiment, 30k dataset was used. Compared to Sequential Join approach, the processing time and the rate of which our approach increases in time as the number of queries increases is orders of magnitude lower than that of Sequential Join approach. Due to unavailability of the implementation of a previous work by Hong et al. (2007), we were unable to compare our approach directly with Hong et al. Instead, we compare our approach indirectly with them via YFilter, as Hong et al. state that they use YFilter as an XPath processor in their implementation, and the XPath evaluation costs by YFilter are much smaller than their join processing costs.

To compare our Join Processing Engine with YFilter, we prepared XPath expressions from XQuery queries as follows. For each XQuery used in the experiment, we randomly selected a `for` clause, and then extracted the XPath path expression from it. We repeated until we collected n distinct XPath expressions, and they were processed by YFilter against the same dataset. Note that the experiment is in favor to YFilter, as it only needs to process up to $100,000$ expressions, although queries with j join operations typically need to process $100,000 \times (j+1)$ path expressions. Figure 10 shows that our join processing costs are lower than YFilter's XPath processing costs. For the following experiments, we only present the experiment results for our Join Tree approach.

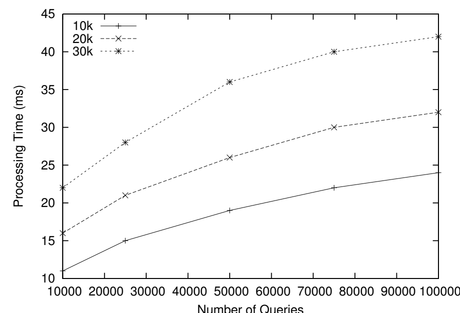


Figure 11: Processing time vs. Number of queries with various sizes of documents

Figure 11 shows the join processing times when various numbers of queries are evaluated against various sizes of documents. As the number of queries increases, the rate of which the join processing time increases drops slowly. This is because, as the number of queries increases, the total number of unique join operations that must be evaluated increases at a much slower rate than the rate of which the total number of join operations increases. In addition, as the number of queries increases, the number of join operations that are shared amongst queries also increases, which also lowers the rate of growth. Lastly, similar patterns were observed for queries with 1, 3, 7, etc... join operations, and when documents were streamed randomly.

Figure 12 shows the join processing times when various numbers of queries with various numbers of join operations were evaluated. The rate of which the processing time increases slowly drops as the number of queries becomes larger. This is because, as the number of queries increases, the number of intermediate join results that are shared amongst similar queries also increases. Therefore, the total number of join operations that must be evaluated does not increase at the same rate as the number of queries increases. Furthermore, the processing time increases as the number of join operations per query increases.

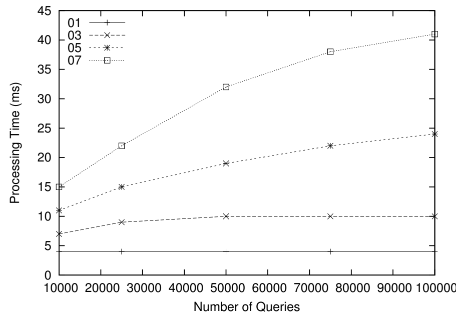


Figure 12: Processing time vs. Number of queries with various numbers of join operations

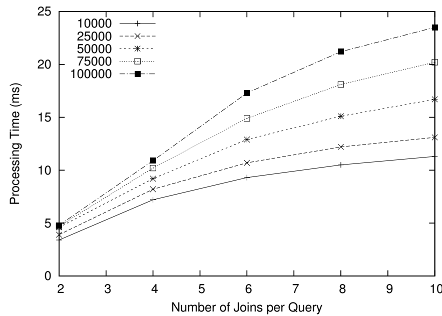


Figure 13: Processing time vs. Number of join operations with various queries

This is because, as the depth of Join Tree increases, the number of Join Tree nodes increases, which result in a larger number of join operations to evaluate.

Figure 13 shows join processing times when queries with various numbers of join operations were evaluated. In this experiment, documents were streamed in a random order. As the number of join operations increases, the growth of processing time slowly decreases. This is because the number of intermediate join results being shared are increased, and while Join Tree is being traversed, short cut evaluations are extensively performed to skip branches that do not have any matching elements. In addition, the number of matching queries has also dropped as the number of join operations increases.

Figure 14 shows processing times when the number of documents stored in a buffer was varied. In this experiment, the number of join operations per query was set to 1, and the number of queries were set to 100,000. The join processing time linearly increases as the number of buffers increases. This is because, as the number of documents in the buffer increases, the number of join operations that need to

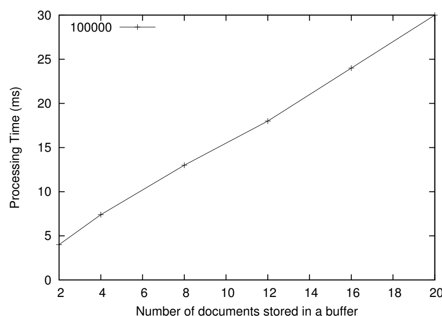


Figure 14: Processing time vs. Number of documents stored in a buffer

Table 3: Analyzing the number of join operations in both normal and optimized modes

	Name	Normal	Optimization
1	# of skipped Join Tree nodes	361	233
2	# of join operations	643	478
3	# of non-empty join operations	590	408
4	# of empty join operations	102	70
5	# of successfully traversed leaves	189	189

be performed every time when a new document arrives increases. We are currently working on sharing buffers to decrease the linear growth in time.

Table 3 shows how much improvement we get with our Join Tree optimization. The table displays the average number of join operations performed for each document. In this experiment, the total number of queries were set to 50,000, but similar patterns were observed when the total number of queries were 10,000, 25,000, etc... The first row shows the number of Join Tree nodes that were examined but not traversed further because there were no matching instances for those queries. These decisions were made without performing join operations. The second, third and fourth rows show the total number of join operations performed, and out of these join operations, how many of them produced non-empty and empty sets. The last row shows the number of Join Tree leaves that were successfully reached and therefore could find matching instances for queries. The analysis shows that the total number of Join Tree nodes that must be traversed was reduced, and hence the total number of join operations that must be performed was reduced substantially. As a result, it significantly increased the overall performance of the system, since join operations were the most expensive operations in terms of runtime costs.

5 Conclusion

We have presented an efficient approach for evaluating a large number of XQuery queries that contain inter-document value-based join operations against streaming documents. We use Join Tree to group and process similar queries simultaneously. While queries are being processed, intermediate join results are shared in order to reduce the overall number of join operations. In addition, unlike many previous works, we return all matching nodes for each matching query. Experiments have shown that, our approach can efficiently evaluate a large number of queries with join operations.

There are several possibilities for future work. First, we are currently working on to support more expressive queries, and join optimization based on the size of Join Tree when frequency information is not available. Second, we are looking at integrating some previous works to improve buffer management. Third, we are looking at integrating some of the traditional join optimizations to further improve both compile and runtime performance.

References

Barta, A., Consens, M. P. & Mendelzon, A. O. (2005), Benefits of path summaries in an xml query optimizer supporting multiple access methods., *in* 'Proceedings of the 31st International Conference on Very Large Data Bases', ACM, Trondheim, Norway, pp. 133–144.

Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M. & Josifovski, V. (2003), Streaming

xpath processing with forward and backward axes, *in* 'Proceedings of the 19th International Conference on Data Engineering', IEEE Computer Society, Bangalore, India, pp. 455–466.

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. & Teubner, J. (2006), Monetdb/xquery: a fast xquery processor powered by a relational engine., *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, Chicago, IL, pp. 479–490.

Bruno, N., Koudas, N. & Srivastava, D. (2002), Holistic twig joins: optimal xml pattern matching, *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, Madison, WI, pp. 310–321.

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

Chen, Y., Davidson, S. & Zheng, Y. (2006), An efficient xpath query processor for xml streams, *in* 'Proceedings of the 22nd International Conference on Data Engineering', IEEE Computer Society, Atlanta, GA, p. 79.

Choi, R. H. & Wong, R. K. (2009), 'Efficient filtering of branch queries for high-performance xml data services', *To appear: Journal of Database Management* .

Clark, J. & DeRose, S. (1999), 'Xml path language (xpath)'. `http://www.w3.org/TR/xpath`.

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

Diao, Y. & Franklin, M. J. (2003), Query processing for high-volume xml message brokering., *in* 'Proceedings of 29th International Conference on Very Large Data Bases', Morgan Kaufmann, Berlin, Germany, pp. 261–272.

Diao, Y., Rizvi, S. & Franklin, M. J. (2004), Towards an internet-scale xml dissemination service., *in* 'Proceedings of the 30th International Conference on Very Large Data Bases', Morgan Kaufmann, Toronto, Canada, pp. 612–623.

Goldman, R. & Widom, J. (1997), Dataguides: Enabling query formulation and optimization in semistructured databases., *in* 'Proceedings of 23rd International Conference on Very Large Data Bases', Morgan Kaufmann, Athens, Greece, pp. 436–445.

Gottlob, G., Koch, C. & Pichler, R. (2005), 'Efficient algorithms for processing xpath queries', *ACM Trans. Database Syst.* **30**(2), 444–491.

Gou, G. & Chirkova, R. (2007), Efficient algorithms for evaluating xpath over streams., *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, Beijing, China, pp. 269–280.

Grust, T., Rittinger, J. & Teubner, J. (2007), Why off-the-shelf rdbmss are better at xpath than you might expect., *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, Beijing, China, pp. 949–958.

Gupta, A. K. & Suciu, D. (2003), Stream processing of xpath queries with predicates, *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, San Diego, CA, pp. 419–430.

Hong, M., Demers, A. J., Gehrke, J., Koch, C., Riedewald, M. & White, W. M. (2007), Massively multi-query join processing in publish/subscribe systems., *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, Beijing, China, pp. 761–772.

Huo, H., Wang, G., Hui, X., Zhou, R., Ning, B. & Xiao, C. (2006), Efficient query processing for streamed xml fragments, *in* 'Proceedings of the 11th International Conference on Database Systems for Advanced Applications', Springer, Singapore, pp. 468–482.

Koch, C., Scherzinger, S., Schweikardt, N. & Stegmaier, B. (2004), Schema-based scheduling of event processors and buffer minimization for queries on structured data streams, *in* 'Proceedings of the Thirtieth International Conference on Very Large Data Bases', Morgan Kaufmann, Toronto, Canada, pp. 228–239.

Kwon, J., Rao, P., Moon, B. & Lee, S. (2005), Fist: Scalable xml document filtering by sequencing twig patterns., *in* 'Proceedings of the 31st International Conference on Very Large Data Bases', ACM, Trondheim, Norway, pp. 217–228.

Li, X. & Agrawal, G. (2005), Efficient evaluation of xquery over streaming data., *in* 'Proceedings of the 31st International Conference on Very Large Data Bases', ACM, Trondheim, Norway, pp. 265–276.

Olteanu, D., Furche, T. & Bry, F. (2004), An efficient single-pass query evaluator for xml data streams, *in* 'Proceedings of the 2004 ACM symposium on Applied computing', ACM, New York, NY, pp. 627–631.

Onizuka, M. (2003), Light-weight xpath processing of xml stream with deterministic automata, *in* 'Proceedings of the 12th International Conference on Information and Knowledge Management', ACM, New Orleans, LA, pp. 342–349.

Peng, F. & Chawathe, S. S. (2005), 'Xsq: A streaming xpath engine', *ACM Trans. Database Syst.* **30**(2), 577–623.

Rao, P. & Moon, B. (2004), Prix: Indexing and querying xml using prüfer sequences., *in* 'Proceedings of the 20th International Conference on Data Engineering', IEEE Computer Society, Boston, MA, pp. 288–300.

The Apache XML Project (2007), 'Xerces2 java parser'. `http://xerces.apache.org/xerces2-j/`.

Uchiyama, H., Onizuka, M. & Honishi, T. (2005), Distributed xml stream filtering system with high scalability, *in* 'Proceedings of the 21st International Conference on Data Engineering', IEEE Computer Society, Tokyo, Japan, pp. 968–977.

UW Database Group (2002), 'Xml data repository'. `http://www.cs.washington.edu/research/xmldatasets/`.

Wang, H., Park, S., Fan, W. & Yu, P. S. (2003), Vist: A dynamic index method for querying xml data by tree structures., *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', ACM, San Diego, CA, pp. 110–121.