Massively Multi-Query Join Processing in Publish/Subscribe Systems[*]

Mingsheng Hong
Cornell University
Ithaca, NY 14853, USA
mshong@cs.cornell.edu

Alan Demers
Cornell University
Ithaca, NY 14853, USA
ademers@cs.cornell.edu

Johannes Gehrke
Cornell University
Ithaca, NY 14853, USA
johannes@cs.cornell.edu

Christoph Koch[†]
Saarland University
Saarbrücken, Germany
koch@infosys.uni-sb.de

Mirek Riedewald
Cornell University
Ithaca, NY 14853, USA
mirek@cs.cornell.edu

Walker White
Cornell University
Ithaca, NY 14853, USA
wmwhite@cs.cornell.edu

ABSTRACT

There has been much recent interest in XML publish/subscribe systems. Some systems scale to thousands of concurrent queries, but support a limited query language (usually a fragment of XPath 1.0). Other systems support more expressive languages, but do not scale well with the number of concurrent queries. In this paper, we propose a set of novel query processing techniques, referred to as *Massively Multi-Query Join Processing* techniques, for processing a large number of XML stream queries involving value joins over multiple XML streams and documents. These techniques enable the sharing of representations of inputs to multiple joins, and the sharing of join computation. Our techniques are also applicable to relational event processing systems and publish/subscribe systems that support join queries. We present experimental results to demonstrate the effectiveness of our techniques. We are able to process thousands of XML messages with hundreds of thousands of join queries on real RSS feed streams. Our techniques gain more than two orders of magnitude speedup compared to the naive approach of evaluating such join queries.

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Query processing

General Terms

Algorithms, Experimentation, Design, Performance

[†]Work done while visiting Cornell University.

[*]This material is based upon work supported by the National Science Foundation under Grant 0621438 and by the Air Force under Grant AFOSR FA9550-06-1-0111. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

Keywords

Publish/Subscribe, Stream Query Processing, Multi-Query Optimization, XML Join

1. INTRODUCTION

XML has become the primary standard for data exchange on the Internet and for enterprise applications. The rapid emergence of Web Services in particular has underlined the need to support efficient XML processing in distributed environments. A crucial component of Web Service based architectures are message brokers. They manage large numbers of subscriptions, or queries that express the interest of subscribers — both users and applications. The subscriptions are matched in real-time with *event streams* (or for short, streams) of incoming XML documents, created by publishers like applications behind a Web Service interface, news services, or blog writers. Because of its close relationship to traditional publish/subscribe (pub/sub) systems, we will use the term *XML publish/subscribe system* to refer to this class of message brokers. In the setting of processing XML streams, events and documents are interchangeable terms.

It is crucial for XML pub/sub systems to be both expressive and scalable. Expressiveness refers to the ability of the query language to support a wide variety of queries. The downside of greater expressiveness is that complex queries are more difficult to implement efficiently. For applications like message brokering, an XML pub/sub system has to scale in terms of the number of subscriptions and the stream rate of incoming messages, while providing sufficient functionality to express all relevant subscriptions.

There has been much recent work on XML pub/sub systems that can efficiently process a large number of XML subscriptions over streaming XML documents [5, 10, 13, 16, 17, 25]. These systems support a proper subset of XPath 1.0, typically limited to forward axes (child and descendant), predicate evaluation and wild card operator $\star$. However, they are unable to express a large class of important queries: queries that correlate *multiple input events* to detect complex patterns in real-time. This class has been recognized as being highly important for event processing [27, 12]. We refer to these queries as *inter-document queries*.

Inter-document queries *join* different XML documents based on values in their nodes, either attributes or text. An inter-document query is capable of joining multiple documents in either the same XML stream, or across multiple streams. For example, for monitoring blogs and news articles, users might be interested in blog postings by the same author or about the same topic that appear within

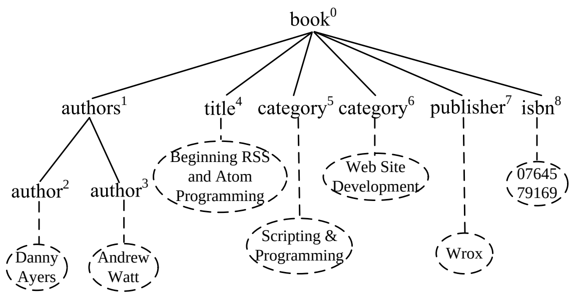

Figure 1: A book announcement document $d1$

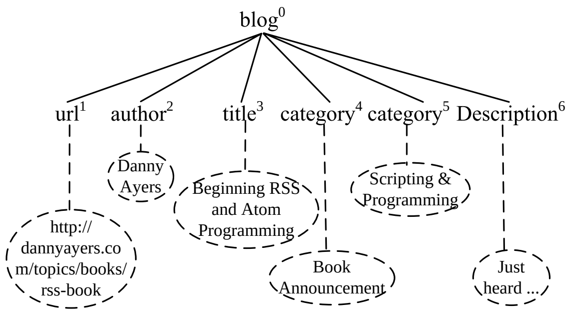

Figure 2: A blog article document $d2$

a short time of each other and are above some reputation threshold. Inter-document queries are also building-blocks for more powerful queries like finding all electronics product announcements that "create above-average attention in the blogosphere." In enterprises, related events containing information about the quality of service that customers receive need to be processed to monitor compliance with service level agreements. There has been some emerging work on XQuery stream processing [21, 15]. XQuery can express join queries, but none of the existing systems scales to a large number of concurrently running queries.

Example. We illustrate our approach with a running example. For ease of exposition, we consider processing of a single XML stream S that includes book announcements and RSS feed items for blog articles. Our techniques can be easily extended to handle multiple XML streams. Two example documents are shown in Figures 1 and 2. The superscript of each element node denotes its node id as defined by pre-order traversal of the XML tree. The dashed ovals connected to leaf nodes with dashed lines represent the text values of the leaf nodes in this document.

Table 1 shows three example queries. Query Q1 looks for a book announcement followed by a blog article from one of its authors that promotes this book. Q2 tries to find a book announcement followed by a blog article from one of its authors following up on material in the book. Q3 checks for blog cross-postings.

Q1	Return a book announcement, followed by a blog article from one of its authors with the same title as the book.
Q2	Return a book announcement, followed by a blog article from one of its authors on the same category as the book.
Q3	Return a pair of blog postings by the same author and with the same title.

Table 1: Examples of Inter-Document Queries

Q1	S//book->x1[.//author->x2][.//title->x3] FOLLOWED BY{x2=x5 AND x3=x6, T1} S//blog->x4[.//author->x5][.//title->x6]
Q2	S//book->x1[.//author->x2][.//category->x7] FOLLOWED BY{x2=x5 AND x7=x8, T2} S//blog->x4[.//author->x5][.//category->x8]
Q3	S//blog->x4[.//author->x5][.//title->x6] FOLLOWED BY{x5=x5' AND x6=x6', T3} S//blog->x4'[.//author->x5'][.//title->x6']

Table 2: XSCL Formulations of queries in Table 1

XML message brokers are used for applications ranging from tens of publishers and subscribers, in small enterprises, to hundreds of thousands of users in Internet scale RSS feed monitoring for blogs and news. Hence an XML pub/sub system has to process anywhere from a few hundred to millions of concurrently active subscriptions for streams that can have high arrival rates. The only way to achieve this kind of scalability is by effective multi-query optimization (MQO).

Unfortunately, MQO for inter-document queries is a very challenging problem. As even the simple queries in Table 1 illustrate, the join condition consists both of tree patterns (e.g., to identify the author nodes and title nodes) and node value comparisons (e.g., equality of author name text for book announcement and blog article). This can create a wide variety of conditions with little apparent commonality. To address this issue, we propose to dissect each query into *tree pattern components* and *value comparison components*. The tree pattern components are expressible in the simpler XPath fragments supported by existing XML pub/sub systems like YFilter [13]. This enables us to leverage existing XML pub/sub technology for efficient discovery of tree pattern components. Unfortunately this does not suffice, because the main performance bottleneck in practice is the evaluation of the value comparison components, as is confirmed by our experimental section.

We show that value comparison components, which have only very limited structure information, almost always can be described by a small number of *query templates*. This is guaranteed for XML documents that have a fairly regular schema, which is common in practice [11], and for documents with a small number of nodes, which is often the case for individual RSS feed items. Even for other XML streams, in practice the number of value comparison components is small, because only a few of the possible comparisons are semantically meaningful. (E.g., it is unlikely that a query would ever compare the author name with the ISBN of a book.) This observation gives us a powerful handle on MQO. Without dissecting join conditions, each different condition would have to be implemented and executed individually, similar to a nested loops join whose outer loop iterates over all queries and whose inner loop evaluates the join predicates. Our dissection approach induces a partitioning of the query set into a small number of equivalence classes, one for each query template. Now we only need a per-template implementation and can take advantage of set-oriented processing of all queries that belong to the same template. By mapping this into a relational join problem, we can take advantage of a wealth of expertise in relational query processing.

The query dissection into tree pattern and value comparison components naturally leads to a *two-stage* approach to query processing. Our system has two major components—the *XPath Evaluator* for processing all tree pattern components and the *Join Processor* for evaluating the value comparison components (see Figure 3). For an incoming XML document, first the XPath Evaluator is in-

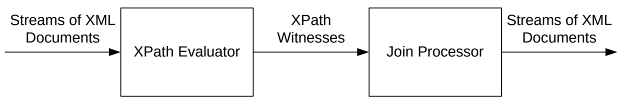

Figure 3: Two-Stage Query Processing

voked to evaluate the tree patterns. It produces a set of bindings of variables defined in these patterns. These bindings are referred to as *XPath witnesses*, or *witnesses* for short. Second, the Join Processor uses the witnesses to perform value joins on a per-template basis. In this scheme, the XPath Evaluator can be viewed as an access method or accelerator for efficiently "retrieving" the witnesses for the join processing stage. As mentioned above, we can leverage existing XML pub/sub technology for the XPath Evaluator and hence focus on the Join Processor in this paper.

Our contributions. The problem we address in this paper is to efficiently process a large number of continuous inter-document queries against incoming XML streams. Our main contributions are as follows.

- We propose novel *Massively Multi-Query Join Processing* techniques[1] for efficiently evaluating a large number of inter-document queries over streams of XML documents. The key to achieving scalability is to dissect join conditions into tree pattern and value comparison components. This leads to a two-stage processing approach in which both storage and computation can be shared effectively among queries.
- We develop a compact representation for the results of the first processing stage, the tree pattern witnesses produced by the XPath Evaluator, for efficient access during the second processing stage. (Section 3)
- We propose a scalable Join Processor for the second stage. The main idea is to map the problem into a relational framework which facilitates sharing of join processing cost across different queries. (Section 4)
- We present query optimization techniques for the Join Processor to further improve performance. Here we take advantage of the relational formulation, e.g., for view materialization. (Section 5)
- We evaluate the performance of our join processing techniques through an extensive set of experiments in Section 6.

We discuss related work in Section 7 and conclude in Section 8.

2. XSCL QUERY LANGUAGE

The XPath fragments that form the query language for existing XML pub/sub systems like YFilter are not expressive enough for inter-document queries. It is possible to express these queries in XQuery, but that is a much more general language with many additional features (and complications), which are not relevant for this discussion. Some of the inter-document queries would look unnecessarily complex in XQuery, obscuring the query structure and optimization opportunities.

To be able to express inter-document queries in a natural and compact manner, we define the XML Stream Conjunctive Language, or XSCL for short. XSCL adds join operators to the XPath operators used by previous XML pub/sub systems. It can be viewed as a fragment of XQuery, i.e., all XSCL queries can be converted into equivalent XQuery expressions. Due to space constraints we omit the formal language definition, which is not necessary for grasping the features relevant to this discussion.

[1]This term is grammatically correct since "Massively" refers to "Multi-Query", rather than to "Join."

Each query in XSCL consists of three clauses: SELECT, FROM and PUBLISH. The SELECT clause specifies how to construct the output XML stream of the query, and is similar to the XQuery RETURN clause. The PUBLISH clause assigns a name to the query's output stream, so that other queries can refer to it as their input. For example, the query "SELECT * FROM blog" outputs every event from input stream blog. This query can be alternatively written as "blog", since in XSCL the SELECT clause can be omitted, defaulting to SELECT *. From a query optimization point of view, the most relevant construct is the FROM clause. It specifies the join condition for the query's input streams, using a variety of operators from two groups—traditional XPath operators and join operators.

XPath operators. Tree patterns in XML documents can be expressed with the same XPath operators that are used by existing XML pub/sub systems. In particular, the following axis operators can be used: / (child), // (descendant), @ (attribute) and [] (predicate). These operators have the usual XPath semantics. We can apply these operators to a particular XML stream S by placing the stream name before them. For example, S//blog//title outputs the titles of blog articles from stream S.

Join operators. In addition to the operators drawn from XPath, XSCL has two join operators, which make it significantly more expressive than the previously used XPath fragments. The join operators are used for inter-document queries. The first, JOIN, is equivalent to the time-based window join operator in the relational data stream processing literature [19]. It has two parameters, *pred* and T, the *join predicate* and *time constraint*, respectively. The expression A JOIN$\{pred, T\}$ B produces an output event when there is an event produced by expression A and an event produced by expression B occurring within T time units of each other, and they together satisfy predicate *pred*. Subexpressions A and B are composed from XPath operators only. We refer to them as *XPath query blocks*, or *query blocks* for short. We will usually use π to denote a query block. In this paper we assume *pred* contains only equality predicates. Efficiently processing a large number of inequality predicates is left as future work.

The second join operator, FOLLOWED BY, corresponds to the sequencing operator in event processing systems [8, 12, 27]. It has the same two parameters as JOIN and can be used in the same context. The only difference is that FOLLOWED BY is "forward-looking." Expression A FOLLOWED BY$\{pred, T\}$ B only produces an output result when there is an event produced by expression A followed by (i.e., with timestamp value greater than) an event produced by expression B within T time units, and they together satisfy predicate *pred*.

Notice that the time constraint parameter T requires XML documents to have timestamps. They can be assigned either by the publishers (event sources) or by the XML pub/sub system itself. This choice is application dependent. A detailed discussion on how to manage timestamps is beyond the scope of this paper and has been examined in related work [26].

Variable binding construct. In the FROM clause, we can also bind XML element nodes obtained through XPath operators in query blocks to variables through the use of the AS clause. These variables can be referred to in join predicates in the FROM clause, and in the SELECT clause for output construction. (This is similar to SQL's AS clause.)

Examples. Table 2 shows the XSCL formulation of the example queries from Table 1, using Ti as the window constraint for query

Qi. Three points should be noted for the XSCL formulations. First, the semantics of the equality operator in XSCL is defined as equality of the string values of the nodes, where the string value of a node is defined by XPath semantics.

Second, in the FOLLOWED BY predicate $pred$ of an XSCL query, it is possible to apply the standard XPath operators like /, // and [] to variables bound in the query blocks to FOLLOWED BY. However, we can show that any XSCL query can be rewritten into a form where predicates inside the FOLLOWED BY part of the query do not contain any XPath operators and only contain value joins that involve pairs of variables bound in the two input query blocks of FOLLOWED BY. We say that an XSCL query q is in *value-join normal form* if q has this property. In the remainder of this paper we assume queries are in this normal form. Also, when two variables (in two different queries or in the same query) have exactly the same definition, we assume the two variables are of the same name. Our assumptions are without loss of generality, since these effects can be achieved through rewrite techniques during query insertion. The three queries presented in Table 2 fulfill our assumptions.

Third, when the SELECT clause is omitted for a join query, we construct the output XML tree in a default way as follows. We create a new root node and make the root element nodes from the two query blocks its children. For example, for query Q1 each output XML tree has two subtrees under the root, where the first subtree corresponds to the output of XPath expression //book[.//author][.//title] given by the first query block, and the second subtree corresponds to the output of XPath expression //blog[.//author][.//title] given by the second query block.

Expressiveness of XSCL. It is easy to show that XSCL is more expressive than conjunctive queries [3]. When the join graph of an XSCL query is cyclic, it is therefore NP-hard to find an optimal query evaluation plan (join ordering) in general. Since we would like to process a large number of continuous XSCL queries, this makes our problem even harder. Hence instead of attacking the general conjunctive query processing problem, we propose an efficient solution that is applicable to a very large and practically important subset of the problem instances.

3. STAGE 1: FROM XSCL QUERIES TO VALUE JOINS

Recall that the two-stage query processing scheme separates XSCL query processing into XPath tree pattern processing and value join processing. Given a set of input XSCL queries, we take all the (single-document) tree patterns corresponding to query blocks in these queries, and insert them into the XPath Evaluator with the goal of returning *witnesses* that represent single-document variable bindings. For each event e, we first invoke the XPath Evaluator to produce all its witnesses, and then value-join the witnesses from e with witnesses from events earlier in the stream. Due to space constraints, we omit the proof that this two-stage query processing scheme yields correct query results.

In this section we describe the first of the two stages of our multiple XSCL query processing, XPath Processing, and focus on how to efficiently represent the witnesses produced by the XPath Evaluator (Section 3.1).

For ease of exposition, we make simplifications to the query structures in the following discussion. First, we consider only XSCL queries with a single FOLLOWED BY operator, where the two corresponding query blocks will match two different XML documents in order to produce a query output. Second, we assume that the predicate of a FOLLOWED BY operator is a conjunction of simple equality predicates on string values. In the following, each such simple equality predicate is referred to as a *value join predicate* or *value join* for short. We also assume that value joins occur only between leaf nodes of tree patterns. Last, we assume all queries read a single input stream. Our techniques can be extended to handle queries involving multiple FOLLOWED BY or JOIN operators with more complex predicates than conjuncts, and more than one input stream.

3.1 XPath Processing and Output Representation

Given an input XML document, the XPath Evaluator can benefit from existing XML pub/sub technology for efficient discovery of tree patterns. How do we represent these witnesses for the second stage value-join processing, while preserving tree structure information in them? One extreme design point for representing XPath witnesses is a relational schema storing each valid combination of all the variable bindings involved in an XPath query block. The other extreme design point would be to completely shred the witnesses into a binary relation of individual bindings of variables, as described below.

For a given XPath query block π, we derive a *variable tree pattern*, which extends the standard notion of an XPath tree pattern [1] by associating each tree node with a variable name. We then create a binary relation for each pair of a parent and a child node in the tree pattern.

This binary relation factors out redundant information. It is analogous to normalization of relational schemas based on functional, multi-value and join dependencies. In addition, the representation for witnesses of one query block will be easy to share among other query blocks that bind to the same XML element nodes. Thus in this paper we decided to examine in-depth this way of representing witnesses; a full exploration of this design space is future work.

To reduce the number of relations, instead of using a binary relation for each edge in the variable tree patterns, we use a single relation of four attributes `(var1, var2, node1, node2)` to store the pairs of variable bindings for *all* edges in the variable tree patterns. Each tuple in this relation stores in `node1` and `node2` a binding consisting of a pair of node ids, and this binding corresponds to a pair of variables whose names are stored in `var1` and `var2`. We denote this relation as R_{binW}, which stands for "binary representation of witnesses".

There are other pieces of information that need to be stored for value join processing in the second stage. We encode them in relations as follows. Note that R_{binW} stores bindings of pairs of variables from the currently processed stream document. The id and timestamp of this document are stored in the singleton-relation R_{docTSW} with schema `(docid, timestamp)`. For example, suppose event e_1 in Figure 1 has document id $d1$ and timestamp $t1$. When it is the current document being processed, R_{docTSW} contains one tuple `(d1, t1)`. Similarly, binary relation R_{docTS} stores the docid, timestamp pairs of previous documents.

The representation of bindings from previous stream documents is very similar to R_{binW}, and they are all stored in a relation R_{bin}. However, since the bindings could come from different documents, the schema of R_{bin} extends that of R_{binW} with an additional `docid` attribute. Its schema is therefore `(docid, var1, var2, node1, node2)`.

In addition to storing the bindings of variable pairs in the tree pattern, we also need to store the string values of nodes that are bound to variables, so that we can evaluate the value joins on the string values of these variable bindings in the Join Processor. To store the string values of nodes from the current stream document while avoiding redundancy, we use a binary relation R_{docW} with schema `(node, strVal)` for this purpose. Nodes that are not bound to any variable will not be stored in this relation. Similarly, we store the string values of nodes bound in previous stream documents in a relation R_{doc}. Its schema is `(docid, node, strVal)`, extending that of R_{docW} with a `docid` attribute.

Example continued. Consider again our running example with Queries Q1, Q2, and Q3 shown in Table 2. Assume that the document $d1$ shown in Figure 1 has been processed. Then Tables 4(b) and 4(c) show the contents of relations R_{bin} and R_{doc}.

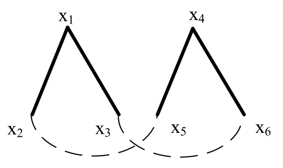

Figure 4: Join Graph of Query Q1 in Table 2

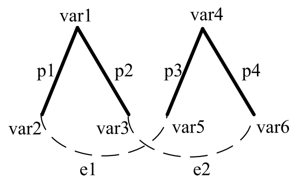

Figure 5: Query Template $\mathcal{Q}$ for Q1, Q2 and Q3 in Table 2

4. STAGE 2: PROCESSING VALUE JOINS

In this section we propose novel techniques for processing a huge number of value joins. A straightforward way would be to evaluate the FOLLOWED BY operator for each XSCL query separately. This strategy is not scalable for two reasons. First, there is no opportunity for sharing of computation among multiple queries. Second, this one-query-at-a-time processing imposes a specific nested-loop style join strategy, where the "outer loop" iterates over each query, and the "inner loop" completes the join processing for that query. With set-oriented query processing strategies, we can significantly improve performance.

Thus, we would like to group the join processing of multiple queries so that computation can be shared among them, and a more efficient join strategy compared to one-query-at-a-time can be used. However, since join operators in different queries could access different variables and have different join conditions, it seems that set-oriented processing of multiple queries is extremely hard to achieve.

The key insight here is that with the right query plan, two different queries can still share processing. In this section, we define the notion of query templates, and present the query plans for value-join processing based on query templates. Intuitively, the XSCL queries are partitioned into equivalence classes based on which query templates they belong to. The join processing of all the XSCL queries belonging to the same query template can now be shared. Therefore, instead of performing joins individually for each XSCL query, we now perform a join for each set of XSCL queries belonging to the same query template.[2]

4.1 Query Template Based Join Processing

Due to space constraints, we give only an informal presentation of the ideas illustrated by examples, emphasizing intuition rather than rigor. The formal definitions of query template based join processing, as well as its proof of correctness, can be found in our online technical report [18].

Given an XSCL query Q with two query blocks connected by a FOLLOWED BY operator, such as query Q1 in Table 2, we can visualize it as a graph, referred to as a *join graph*, illustrated by Figure 4. Each query block is represented by a tree pattern formed by solid, bold edges, referred to as *structural edges*. Each node in the tree pattern is labeled by the name of a variable bound in the corresponding query block in Q. For example, the root node of the left-hand-side tree pattern in Figure 4 is labeled by x_1, the name of the variable bound to //book in Q1. There are two types of structural edges, representing child axis and descendant axis. For ease of exposition, we assume only descendant axes are present in the XSCL queries we deal with. For each equality predicate $x = y$ in the FOLLOWED BY predicate of Q, we draw a dashed edge between the two (leaf) nodes corresponding to x and y. We call such an edge a *value join edge*. For example, the value join edge between x_2 and x_5 in Figure 4 corresponds to the join predicate $x_2 = x_5$ in Q1.

A *query template* (or a *template* for short) $\mathcal{Q}$ of Q, is a graph isomorphic to its join graph with different node labels described as follows. Each node u in $\mathcal{Q}$ is labeled by a uniquely named *meta-variable*, whose value is the label of u's corresponding node v in the join graph of Q; i.e., the name of v's corresponding variable in the query Q. Each edge is $\mathcal{Q}$ is also uniquely labeled.

For example, Q1 in Table 2 belongs to the query template denoted as $\mathcal{Q}$, which is shown in Figure 5. Q2 and Q3 in Table 2 also belong to the same query template. The six nodes in this query template are labeled from `var1` to `var6`. The value of the meta-variable `vari` is `xi` for $1 \leq i \leq 6$. The correspondence between nodes and edges in the query template and each query is obvious. For example, edge `p1` connecting `var1` and `var2` in the template corresponds to the structural constraint `x1//x2` in Query Q1.

4.2 Sharing Templates With Graph Minor

In Section 4.1, we require that the query template $\mathcal{Q}$ of a query Q be isomorphic to its join graph. However, we can show that if we derive a simplified query template $\mathcal{Q}'$ of Q from the graph minor [24] of the join graph of Q through a set of reduction rules below, the join processing result on $\mathcal{Q}'$ will be the same as $\mathcal{Q}$. This enables more queries to share the same query template for join processing.

Given the join graph of Q, we compute its minor via the following reduction rules. First, we recursively remove the leaf nodes that do not participate in any value joins from the join graph. Next, we remove the nodes that are not the descendants of the least common ancestors of the remaining leaf nodes. Finally, we remove all those intermediate nodes that have only one child in the modified join graph. The resulting join graph contains only leaf nodes that participate in value joins, as well as the intermediate nodes that are the least common ancestors of some of the leaf nodes. We derive the query template of Q from the resulting join graph.

The intuition is that since the structural constraints for each individual query block in Q have been evaluated by the XPath Evaluator in Stage 1, the value join processing stage need only check the value constraints, as well as a subset of the structural constraints involving those leaf nodes that satisfy the value constraints. The correctness of this approach is proved in [18].

The number of different query templates depends on the maximum number of value join predicates in the query workload, but not on the number of queries registered with the system, even if these queries have very different tree patterns or seem to equate different nodes. For example, for queries with three value joins in the join predicate of one FOLLOWED BY operator, we show all 16 possible query templates in Figure 6. The first 6 templates in

[2]Mathematically speaking, instead of performing join on the original XSCL query space, we now perform join on the quotient space of the XSCL queries defined by the equivalence relation induced by query templates.

#VJ	#QT(flat schema)	#QT(complex schema)
1	1	1
2	3	3
3	6	16
4	16	<230

Table 3: Number of Query Templates with respectd to Number of Value Joins

```
Algorithm 1 Join Processing Algorithm
Require: Current stream document $d$
1: Invoke the XPath Evaluator in $d$ to produce $R_{binW}$, $R_{docW}$ and $R_{docTSW}$
2: for all query templates $\mathcal{Q}$ in the system do
3:     Evaluate the corresponding conjunctive query to produce results of XSCL queries belonging to template $\mathcal{Q}$
4: Maintain join state with Algorithm 2
```

the dashed box correspond to the query templates for queries defined on a “flat” XML document schema with two tree levels, such as the schema of the blog articles illustrated in Figure 2. Table 3 shows the relationship between the number of value joins involved in the queries and the number of different query templates for these queries. We leave it as future work to derive a closed-form formula for the exact relationship.

In the remainder of this section, we will explain our join processing techniques based on query templates. Our techniques can be decomposed into two parts. First, we encode all the information needed in join processing as relations, so that we can leverage techniques from relational join processing (Section 4.3). Second, for each query template, we create a relational conjunctive query with which we evaluate all XSCL queries belonging to that query template at once (Section 4.4). Our query template based join processing algorithm for each document d is given as Algorithm 1.

4.3 Representing Join Graphs As Relations

The information needed in join processing includes the join graphs of the XSCL queries, and the XPath witnesses from the current stream document as well as from previous stream documents that participate in the join. We have shown in Section 3.1 how to encode XPath witnesses from the current and previous stream documents in relations R_{binW}, R_{docW}, R_{docTSW}, R_{bin}, R_{doc} and R_{docTS}. We now describe how to encode the join graphs of XSCL queries based on query templates as relations.

For each query template $\mathcal{Q}$, we use a relation R_T to encode the join graphs of XSCL queries belonging to this template. The schema contains one attribute `qid` for storing the query id. Also, it contains one attribute `vari` for each node in the query template labeled by `vari`, the name of a meta-variable. Finally, it contains one attribute `wl` for storing the window length of the join operator. Each query belonging to the template $\mathcal{Q}$ will be encoded as a tuple in relation R_T. For example, the schema and content of relation R_T for the three queries in Table 2 belonging to join template $\mathcal{Q}$ in Figure 5 is shown in Table 4(a).

4.4 Conjunctive Query For Each Template

For each XSCL query template $\mathcal{Q}$, we create a relational conjunctive query, denoted as CQ_T, so that the XSCL queries belonging to $\mathcal{Q}$ can be evaluated all at once in CQ_T.

We present the conjunctive queries in Datalog. For a given query template $\mathcal{Q}$, here is how we create CQ_T. For each

```
Algorithm 2 Maintain Join State $R_{doc}$, $R_{bin}$ and $R_{docTS}$
Require: $R_{docW}$, $R_{binW}$ and $R_{docTSW}$ produced by the XPath Evaluator when processing the current stream document
1: Set $R_{doc}$ to $R_{doc} \cup (R_{docW} \times \pi_{timestamp}(R_{docTSW}))$
2: Set $R_{bin}$ to $R_{bin} \cup (R_{binW} \times \pi_{timestamp}(R_{docTSW}))$
3: Set $R_{docTS}$ to $R_{docTS} \cup R_{docTSW}$
```

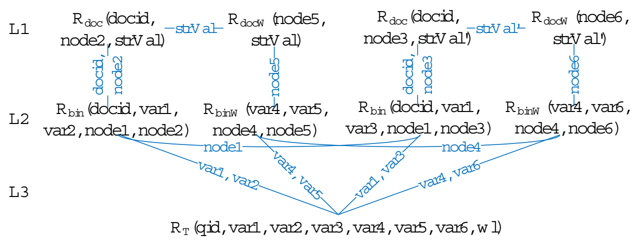
L1
L2
L3
Rdoc(docid, node2, strVal)
strVal
RdocW (node5, strVal)
Rdoc(docid, node3, strVal')
strVal'
RdocW (node6, strVal')
docid, node2
node5
docid, node3
node6
Rbin(docid, var1, var2, node1, node2)
RbinW (var4, var5, node4, node5)
Rbin(docid, var1, var3, node1, node3)
RbinW (var4, var6, node4, node6)
node1
node4
var1, var2
var4, var5
var1, var3
var4, var6
RT(qid, var1, var2, var3, var4, var5, var6, wl)

Figure 7: Relational Conjunctive Query CQ_T For XSCL Query Template $\mathcal{Q}$ in Figure 5

value join edge in template $\mathcal{Q}$, there is a copy of R_{doc} and R_{docW} joined on their string value attributes in the body of CQ_T. For example, for edge e_1 of query template $\mathcal{Q}$ in Figure 5, we put a copy of R_{doc}`(docid, node2, strVal)` and R_{docW}`(node5, strVal)` joined on string value `strVal` in the body of CQ_T. For each structural edge in the query template, we put a copy of R_{bin} or R_{binW} in CQ_T body, depending on whether this edge appears on the LHS or RHS tree pattern in the query template. We do not need to evaluate in the body of CQ_T the tree pattern parts of the XSCL queries, since these structural constraints have been evaluated in the XPath Evaluator, and their results have been stored in R_{binW} and R_{bin}. For example, for edge $p1$ of query template $\mathcal{Q}$ in Figure 5, we put a copy of R_{binW}`(docid, var1, var2, node1, node2)` in the body of CQ_T. This completes the construction of CQ_T body.

The head of the conjunctive query CQ_T is a relation denoted as R_{outT}, whose schema contains `qid`, `docid1`, `wl`, as well as one attribute for each node involved in the conjunctive query. For example, the schema of R_{outT} for query template $\mathcal{Q}$ in Figure 5 is `(qid, docid1, node1, node2, node3, node4, node5, node6, wl)`, where `nodei` stores the binding node id of an XSCL query variable whose named is stored as value in `vari` in the template. For each tuple in this relation, `node1` through `node3` values come from document `docid1`. `node4` through `node6` values come from the current document.

Below we give the Datalog representation of the conjunctive query for query template $\mathcal{Q}$ in Figure 5.

$$
\begin{aligned}
&R_{outT}(qid, docid, node1, node2, node3, node4, node5, node6, wl) \text{ :- }\\
&R_{doc}(docid, node2, strVal), R_{bin}(docid, var1, var2, node1, node2),\\
&R_{docW}(node5, strVal), R_{binW}(var4, var5, node4, node5),\\
&R_{doc}(docid, node3, strVal'), R_{bin}(docid, var1, var3, node1, node3),\\
&R_{docW}(node6, strVal'), R_{binW}(var4, var6, node4, node6),\\
&R_T(qid, var1, var2, var3, var4, var5, var6, wl)
\end{aligned}
$$

This conjunctive query CQ_T is visualized in Figure 7. In this figure, each node is a relation in the body of CQ_T. There is an edge between two relations, if there is a join between them. The edge is labeled by the set of attributes on which the two relations are joined. In the visualization of the conjunctive query, we place the relations in three levels, denoted as L1, L2 and L3. The relations in level L1 are copies of R_{doc} and R_{docW}. The relations in L2 are copies of R_{bin} and R_{binW}. In level L3, there is always only one

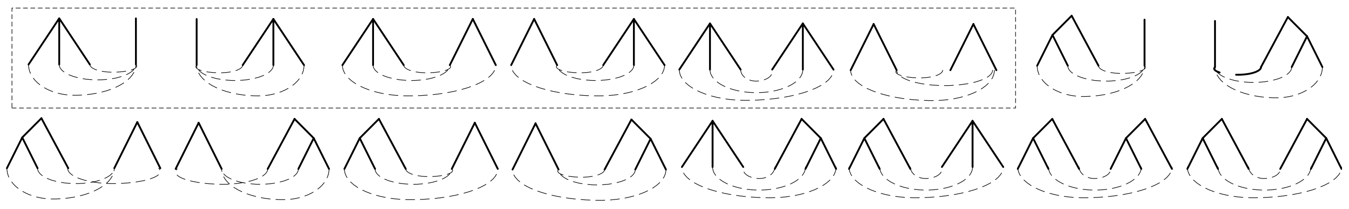

Figure 6: 16 Query Templates With 3 Value Joins

Table 4: Relations involved in Section 4.4.1

(a) R_T for Query Template $\mathcal{Q}$ in Figure 5

qid	var1	var2	var3	var4	var5	var6	wl
Q1	x1	x2	x3	x4	x5	x6	T1
Q2	x1	x2	x7	x4	x5	x8	T2
Q3	x4	x5	x6	x4	x5	x6	T3

(b) R_{doc} After Processing $d1$

docid	node	strVal
d1	0	–
d1	2	Danny Ayers
d1	3	Andrew Watt
d1	4	Beginning RSS and Atom Programming
d1	5	Scripting & Programming
d1	6	Web Site Development

(c) R_{bin} After Processing $d1$

docid	var1	var2	node1	node2
d1	x1	x2	0	2
d1	x1	x2	0	3
d1	x1	x3	0	4
d1	x1	x7	0	5
d1	x1	x7	0	6

(d) R_{docW} of $d2$

node	strVal
0	–
2	Danny Ayers
3	Beginning RSS and Atom Programming
4	Book Announcement
5	Scripting & Programming

(e) R_{binW} of $d2$

var1	var2	node1	node2
x4	x5	0	2
x4	x6	0	3
x4	x8	0	4
x4	x8	0	5

(f) Content of R_{outT} After Processing $d2$

qid	docid1	node1	node2	node3	node4	node5	node6	wl
Q1	d1	0	2	4	0	2	3	T1
Q2	d1	0	2	5	0	2	5	T2

Algorithm 3 Producing Query Results From R_{outT}

```
Require: Input relations $R_{outT}, R_{docTSW}$ and $R_{docTS}$
1: Let the single tuple in $R_{docTSW}$ be $d2$
2: for all tuples $a$ in $R_{outT}$ do
3:     Find a tuple $d1$ in $R_{docTS}$ with $d1.docid = a.docid1$
4:     if $0 < d2.timestamp - d1.timestamp \leq a.wl$ then
5:         Construct an output XML document for the query with id
           $a.qid$ based on the specification of its SELECT clause
```

relation R_T for the query template $\mathcal{Q}$. The relations in level L1, L2 and L3 are joined together to produce R_{outT}.

To produce final query outputs from R_{outT}, we invoke Algorithm 3, which iterates over tuples in R_{outT}. For each tuple, we first make sure that the temporal constraint of its corresponding query is satisfied (Line 4). Note that the temporal constraint we check in Algorithm 3 corresponds to that for FOLLOWED BY operator. If the temporal constraint is satisfied, we then produce an output XML document according to the specification of the SELECT clause in that query. This process of producing query results from R_{outT} is straightforward. We therefore do not discuss it further and focus only on the conjunctive query CQ_T that produces relation R_{outT} for each query template $\mathcal{Q}$.

After query results have been generated for the current document, in Line 4 of Algorithm 1, we maintain the join state consisting of relations R_{doc}, R_{bin} and R_{docTS} with Algorithm 2. Afterwards, we can discard the relations R_{docW}, R_{binW} and R_{docTSW}, and start processing the next stream document.

4.4.1 *Query Processing Example*

Let us now walk through the query processing steps for queries Q1, Q2, Q3 in Table 2 against the sequence of two documents $d1$ and $d2$ shown in Figure 1 and 2, which have timestamps $t1$ and $t2$ ($t1 < t2$) respectively.

When document $d1$ comes into the system, since R_{doc} and R_{bin} are initially empty, $d1$ does not produce any query result. R_{docW}, R_{binW} and R_{docTSW} are then merged into R_{doc}, R_{bin} and R_{docTS} respectively, with the docid value of each new tuple in R_{doc} and R_{bin} set to $d1$. The content of R_{doc} and R_{bin} at the end of processing this document is shown respectively in Table 4(b) and 4(c). R_{docTS} contains only one tuple, $\{(d1, t1)\}$.

When document $d2$ arrives, we show the content of R_{docW} and R_{binW} produced by the XPath Evaluator in Table 4(d) and 4(e). R_{docTSW} contains one tuple $\{(d2, t2)\}$.

Now we want to join R_{doc}, $R_{bin}, R_{docW}, R_{binW}$, and R_T to produce R_{outT}. The content of R_{outT} is shown in Table 4(f).

Finally, we invoke Algorithm 3 to produce one output XML document each for query Q1 and Q2. According to XSCL semantics, the two output XML documents produced by Q1 and Q2 have exactly the same content. The root of the output document has two subtrees, where the first subtree corresponds to the subtree rooted at the `book` element in $d1$, and the second subtree corresponds to the subtree rooted at the `blog` element in $d2$.

5. QUERY OPTIMIZATION

We have presented the basic ideas of query template based join processing in Section 4. The result of these techniques, Algorithm 1, evaluates the conjunctive queries for different templates independently as is shown in Line 2. It therefore leaves much room for sharing computation among these query templates. Also, join processing for the current XML event on the stream might benefit from remembering the results of processing previous XML events. In this section, we propose view materialization as the solution to both these issues.

So far we have assumed that we keep as join state only R_{doc}, R_{bin} and R_{docTS}. We have not considered materializing any intermediate join results for the conjunctive query CQ_T of a query template $\mathcal{Q}$. We now would like to explore the view materialization spectrum with respect to join processing cost.

Let $R_{\hat{L}}$ denote the result of joining R_{doc} and R_{bin}. In one extreme of the spectrum, adopted by the Algorithm 1, we do not materialize $R_{\hat{L}}$, and instead compute it from R_{doc} and R_{bin} for each incoming document. This is likely to result in redundant computation in the join processing. In the other extreme of the spectrum, we can try to materialize the entire $R_{\hat{L}}$, and keep it up to date after processing each incoming document. The materialization of $R_{\hat{L}}$ makes the join processing for each input document less expensive. However, the view maintenance cost of $R_{\hat{L}}$ is likely to be high, since in order to maintain $R_{\hat{L}}$ for each incoming document, we need to first join R_{binW} and R_{docW} together, whose result is denoted as $R_{\hat{R}}$, and then merge $R_{\hat{R}}$ into the existing $R_{\hat{L}}$. Although R_{docW} will be small for each incoming document, the size of R_{binW} could be proportional to the number of XSCL queries in the system, and therefore the join result could be very large. Also, it may not be worth maintaining the entire $R_{\hat{L}}$, if we do not use such a materialized result in its entirety in processing future documents. We would therefore like to find a sweet spot in the materialization spectrum to minimize the sum of join processing and view maintenance costs.

Determining how much of $R_{\hat{L}}$ to materialize requires a careful study of how $R_{\hat{L}}$ is used in query processing. The schema of $R_{\hat{L}}$ is `(docid1, var1, var2, node1, node2, strVal)`, where variables `var1` and `var2` bind respectively to nodes `node1` and `node2` in document `docid1`, and `node1` is an ancestor of `node2`. Also, `strVal` is a string value corresponding to node `node2`. Recall this is because we assumed that value joins only happen at tree pattern leaf nodes; that is, R_{doc} and R_{bin} are joined on R_{doc}.`node` = R_{bin}.`node2`, and therefore `strVal` in the result corresponds to the string value of node `node2`.

Note that for each incoming document, we usually do not have to access *all* the tuples in $R_{\hat{L}}$. Instead, we only need to access those tuples whose string values appear in the nodes from the current stream document that are bound to variables. In other words, we will only access those tuples in $R_{\hat{L}}$ whose string values are in the result of $R_{docW} \ltimes_{\texttt{strVal}} R_{doc}$. Formally, we denote this subset of $R_{\hat{L}}$ as R_L, defined by $R_{docW} \ltimes_{\texttt{strVal}} (R_{doc} \bowtie_{\texttt{node=node2}} R_{bin})$. If we could materialize this part of $R_{\hat{L}}$, then we could save the costs of the joins that produce them in the join processing for conjunctive query CQ_T's. Also, this observation is symmetric between $R_{\hat{L}}$ and $R_{\hat{R}}$. That is, those tuples in $R_{\hat{R}}$ whose string values correspond to some nodes in $R_{\hat{L}}$ will be accessed and participate in other joins. This means we will have to compute those parts of $R_{\hat{R}}$. Formally, the subset of $R_{\hat{R}}$ that needs to be computed is $R_R \equiv R_{doc} \ltimes_{\texttt{strVal}} (R_{docW} \bowtie_{\texttt{node2=node}} R_{binW})$. In sum, only the tuples in R_L and R_R will participate in conjunctive query processing.

For each incoming XML event, we cannot avoid the cost of computing R_R. However, it is possible to reduce the cost of computing R_L through materialization of join results for previous events. To do so, we break up $R_{\hat{L}}$ into slices, where each slice is a set of tuples produced by the join of tuples in R_{doc} with a certain string value and R_{bin}. Specifically, we keep a "view cache" of slices in $R_{\hat{L}}$, denoted as VC, where each cache entry is keyed on a string value s, and stores in the value component a relation $R_{L,s}$, computed by $\mathcal{E}_{L,s} \equiv \sigma_{\texttt{strVal}=s}(R_{doc}) \bowtie_{\texttt{node=node2}} R_{bin}$. Similarly, we define $\mathcal{E}_{R,s}$ to be $\sigma_{\texttt{strVal}=s}(R_{docW}) \bowtie_{\texttt{node=node2}} R_{binW}$.

Whenever we perform a join between the set of tuples in R_{doc} with a certain string value s and R_{bin}, we first look up the view cache with search key s, to see whether it has been materialized. The size of the view cache can be set according to the memory constraint of the system. Cached entries can be replaced by a cache replacement policy appropriate for the workload, such as LRU.

We incorporate the materialization based optimization above into Algorithm 1 to produce an improved algorithm, Algorithm 4. Essentially, Line 2 through Line 8 are newly added to compute the slices of R_L and R_R, in order to reduce the query processing cost of Line 10. The computation of slices of R_L benefits from remembering the partial result of processing previous XML events, in particular, slices of $R_{\hat{L}}$. The union of these computed slices of R_L (resp. R_R) gives the result of the entire R_L (resp. R_R).

We then evaluate the conjunctive query for each query template in Line 9 – 10. Note that we no longer need to access $R_{bin}, R_{binW}, R_{doc}$ and R_{docW}. Instead, we access only R_L and R_R computed above. This enables sharing of join processing among different query templates. For example, to process query template $\mathcal{Q}$ in Figure 5, we modify the conjunctive query CQ_T presented in Section 4.4 into the following query which accesses only R_L, R_R and R_T.

$$
\begin{aligned}
&R_{outT}(qid, docid, node1, node2, node3, node4, node5, node6, wl) \text{ :– }\\
&R_L(docid, var1, var2, node1, node2, s),\\
&R_R(var4, var5, node4, node5, s),\\
&R_L(docid, var1, var3, node1, node3, s'),\\
&R_R(var4, var6, node4, node6, s'),\\
&R_T(qid, var1, var2, var3, var4, var5, var6, wl)
\end{aligned}
$$

Finally, we maintain the join state and view cache in Line 11 – 12 of Algorithm 4.

6. PERFORMANCE EVALUATION

We measure the performance of join processing and our optimization techniques at two levels. We generate a technical benchmark through synthetically generated data of different document schema complexity, and we also measure the performance of our techniques on real RSS data. We have written an XSCL translator, which translates XSCL queries into SQL queries that correspond to the relational conjunctive queries described in Section 4. These SQL queries are then evaluated on an SQL engine. We choose Microsoft SQL Server 2005 Standard Edition in the experiments as the Join Processor. All experiments were run on a Dual Core 3.6 GHz Pentium D PC with 3.5 GB of RAM. The operating system is Windows XP Professional. We repeat each experiment 10 times. The standard deviation in all runs was well below 1%; we therefore report only averages, omitting error bars from the graphs.

6.1 Technical Benchmark

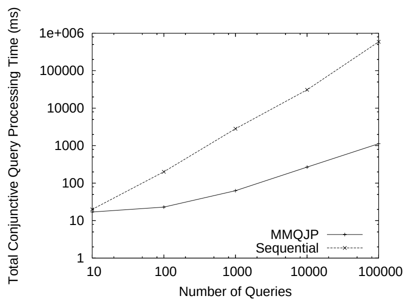

Figure 8: Performance on Simple Document Schema

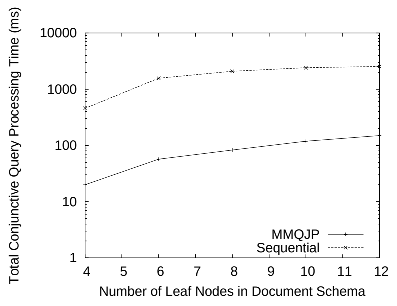

Figure 9: Performance on Simple Document Schema

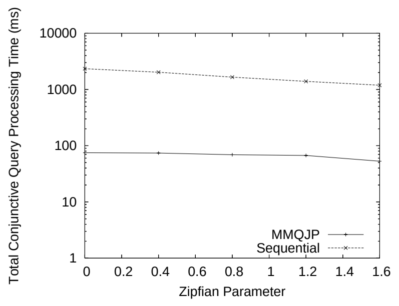

Figure 10: Performance on Simple Document Schema

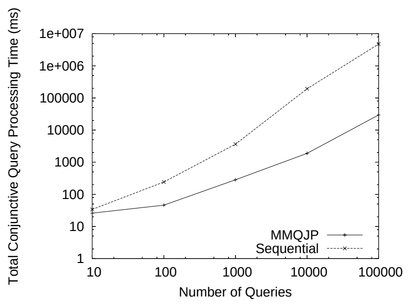

Figure 11: Performance on Complex Document Schema

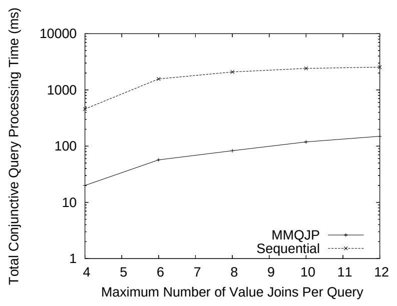

Figure 12: Performance on Complex Document Schema

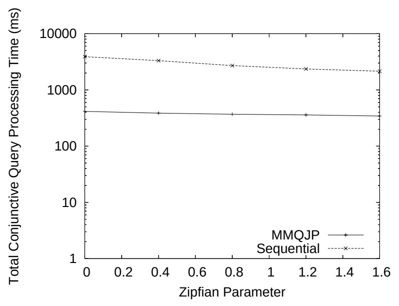

Figure 13: Performance on Complex Document Schema

Algorithm 4 Improved Join Processing Algorithm With View Cache

```
Require: Current stream document $d$
 1: Invoke the XPath Evaluator in $d$ to produce $R_{binW}$, $R_{docW}$
    and $R_{docTSW}$
 2: Semi-join $R_{docW}$ with $R_{doc}$ on strVal to obtain a set $STR$
    of common string values
 3: for all distinct string values $s$ in $STR$ do
 4:   if there is an entry with key $s$ in view cache $VC$ then
 5:     Set relation $R_{L,s}$ to the value component of the entry
 6:   else
 7:     Compute relation $R_{L,s}$ by $\mathcal{E}_{L,s}$, and insert an entry into
        $VC$ with key $s$, and value $R_{L,s}$
 8:   Compute relation $R_{R,s}$ by $\mathcal{E}_{R,s}$
 9: for all query templates $\mathcal{Q}$ in the system do
10:   Evaluate the corresponding conjunctive query $CQ_T$, with
      $R_{L,s}$'s and $R_{R,s}$'s computed above
11: Maintain join state with Algorithm 2
12: Maintain $VC$ with Algorithm 5
```

In this first set of experiments, we evaluate a set of XSCL queries that join two fixed input documents. We compare the performance of our join processing algorithm from Section 4, which we denote as MMQJP in the figures, with a naive approach which sequentially evaluates the FOLLOWED BY operator in each XSCL query, denoted as Sequential. We run this experiment on XML documents with different complexity in their schema.

Two-Level Document Schema. We first choose a document schema that models the schema of an RSS feed item, shown by the example in Figure 2. The schema has only two levels, where all leaves are children of the root. Let N be the number of leaves in the schema. parameters in this experiment and their default values are shown in Table 5.

Algorithm 5 Maintain View Cache VC

```
Require: Set $STR$ of common string values in $R_{doc}$ and $R_{docW}$
Require: The $R_{L,s}$'s and $R_{R,s}$'s computed when processing the
    current document
1: for all string values $s$ in $STR$ do
2:   Set $R_{L,s}$ to $R_{L,s} \cup R_{R,s}$
3:   Insert/Update the cache entry keyed on $s$ with value $R_{L,s}$
```

We then manually compose two documents conforming to this schema, referred to as d_1 and d_2. The root node in d_1 is denoted as n_0, and the N leaf nodes in d_1 are denoted as n_1 through n_N. Similarly, the root node in d_2 is denoted as n'_0, and the N leaf nodes in d_2 are denoted as n'_1 through n'_N. These two documents have the property that all leaf nodes in each document have different string values, but each leaf node n_i in $d1$ has the same string value as the leaf node n'_i in the corresponding position in $d2$, for $1 \leq i \leq N$.

Since our focus is measuring the performance of the join processor, we need to compute R_{doc}, R_{docW}, R_{bin} and R_{binW} as the inputs to join processing. Given the properties of the two documents, we compute these tables as follows. We insert N tuples into R_{doc} corresponding to the N leaves in d_1, where each tuple stores the information of node ID and the string value of a particular leaf in d_1. R_{bin} also contains N tuples, where each tuple corresponds to a particular parent, child pair in d_1. Similarly, we load information of d_2 into R_{docW} and R_{binW}. Note that the tables generated above are guaranteed to be supersets of the results returned by the XPath Evaluator on any number of XPath query blocks. We therefore do not need to invoke the XPath Evaluator in this experiment.

We generate each XSCL query by first selecting a set of variables bound in the LHS and RHS tree patterns of that query in the

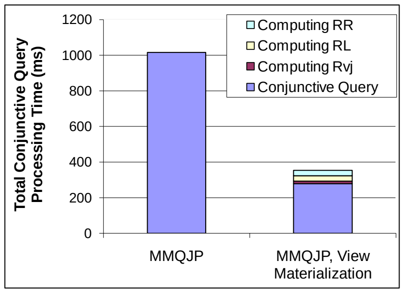

Figure 14: View Materialization on Simple Document Schema

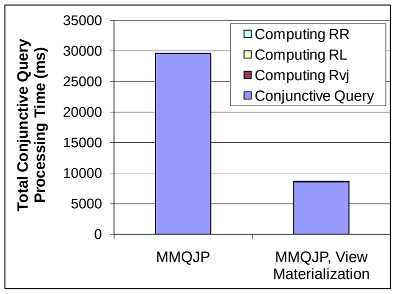

Figure 15: View Materialization on Complex Document Schema

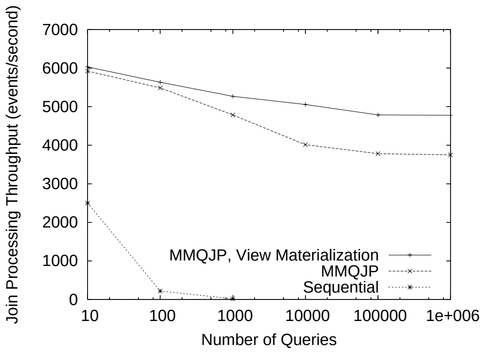

Figure 16: Performance on RSS Stream Processing

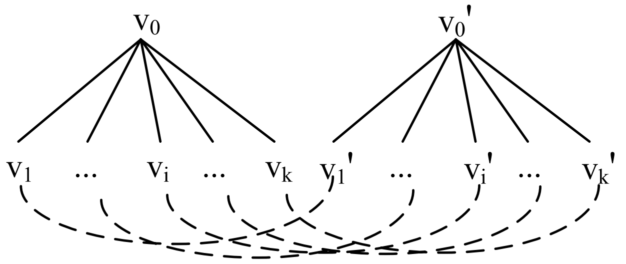

Figure 17: Random Generation of XSCL Queries

Variable	Default Value
Number of XSCL queries	1000
Number of leaves in document schema	6
Zipfian parameter	0.8

Table 5: Parameters (default values)

following way. We randomly pick an integer value k from 1 to N with a Zipfian distribution. For the LHS tree pattern, there are k variables bound to the leaf nodes in the document schema, denoted as v_1 through v_k, as well as a variable v_0 bound to the root node. v_0 is bound only to root node n_0 in document d_1. The k variables $v_i(1 \leq i \leq k)$, are mapped to k different leaf nodes $n_j(1 \leq j \leq N)$ in document d_1 chosen uniformly at random. Similarly, there are k variables v'_1 through v'_k bound to leaf nodes for RHS tree pattern, as well as a variable v'_0 bound to the root. v'_0 is bound only to n'_0 in document d_2. The k variables $v'_i(1 \leq i \leq k)$ are randomly bound to k different leaf nodes $n'_j(1 \leq j \leq N)$ in d_2. We now generate k value joins for this query, where the i^{th} join has a string value equality predicate $v_i = v'_i$. This finishes the construction of query Q. The query construction is shown in Figure 17. Observe that based on this query generation approach, the maximum number of query templates in our join techniques is exactly N, regardless of the actual number of XSCL queries generated.

First, we vary the number of XSCL queries, and show the result in Figure 8. When the number of queries is small, the performance of MMQJP and sequential evaluation does not differ much. However, MMQJP gains more than two orders of magnitude improvement when the number of queries is large.

We then vary N, the number of leaf nodes in the schema. The result is shown in Figure 9. In the way we generate XSCL queries, increasing N will result in more query templates in MMQJP. The time cost of both approaches is about 6 times larger at $N = 12$ compared to $N = 4$; recall from Section 4 that the complexity of the query template does not increase linearly with N.

We also vary the Zipfian parameter for generating k for each query (queries with smaller k values are more likely to be generated), and show the results in Figure 10. Parameter k has little impact on the performance of MMQJP, since the number of query templates remain the same under these Zipfian values[3]. On the other hand, the performance of sequential evaluation improves by a factor of 2 when the Zipfian value increases from 0.0 to 1.6, because the queries are much simpler at a higher value of the parameter of the Zipf distribution.

Three-Level Document Schema. We repeat the same set of experiments on a more complex document schema. This schema has three levels of tree nodes, where the root and the intermediate nodes all have a branching factor of 4, resulting in 16 leaf nodes in this schema. As in the previous setting, we manually compose two documents d_1 and d_2 conforming to this schema, with the property that the string values of the leaf nodes in the corresponding positions of the two documents are identical.

In this setting, we have a new parameter K, denoting the maximum number of value joins per query. Its default value is 4. To generate each query, we first randomly pick a value k from 1 to K with Zipfian distribution. As in the previous setting, for the LHS tree pattern, there are k variables v_1 through v_k bound to leaf nodes in the document schema. We pick uniformly at random k different leaf nodes from d_1 to be bound to these k variables. variable v_0 in LHS pattern is bound to the root node of document d_1. Now, to form a more complex tree pattern compared to the previous setting, the nodes in the intermediate level of the document schema that are along the paths between the root node and the leaf nodes bounded by v_1 through v_k will be bounded by additional variables in the LHS tree pattern. This adds additional structural joins in the conjunctive query for each query template. The construction for RHS tree pattern is similar. Finally, we generate k value join predicates for the XSCL query, where the i^{th} predicate is $v_i = v'_i$.

In this setup, we vary the number of queries, the maximum number of value joins per query, and the Zipfian parameter for generating k. The results are shown Figures 11, 12 and 13, respectively.

When we vary the number of queries, the time cost of both approaches grows more than linearly. This is because as the number of queries grows, more query templates are involved. In MMQJP, the number of query templates increases from 6 to 22 when the number of queries grow from 10 to 100000. Still, MMQJP outper-

[3]Only when the Zipfian distribution is extremely skewed, some query templates involving many value joins will not occur.

forms sequential evaluation by two orders of magnitude when there are 100000 queries.

When we vary K, the maximum number of value joins per XSCL query, we see that the time cost of MMQJP grows faster than sequential evaluation. This is because MMQJP is affected more significantly by the increasing number of query templates. The numbers of query templates are 2, 6, 20 and 39 for $K = 2, 3, 4$ and 5, respectively. Varying the Zipfian parameter in this setting has a larger impact on the performance of sequential evaluation compared to MMQJP, because similar as in the previous scenario the numbers of query templates stay constant (around 20), whereas many actual queries have a simpler structure.

6.2 Query Optimization

We presented query optimization techniques based on view materialization in Section 5. We now evaluate its effectiveness based on the synthetic workload described in the previous section. Since we are interacting with the database engine on the level of SQL, it is difficult to cache slices of $R_{\hat{L}}$ as was described in Section 5. Therefore, given the input $R_{bin}, R_{binW}, R_{doc}$ and R_{docW} to the Join Processor, we materialize the following relations:

$$R_{vj}(n1, n1', s) \coloneq R_{doc}(d1, n1, s), R_{docW}(n1', s)$$
$$R_L(d1, v1, v2, n1, n2, s) \coloneq R_{vj}(n1, n1', s), R_{bin}(d1, v1, v2, n1, n2)$$
$$R_R(v1, v2, n1', n2', s) \coloneq R_{vj}(n1, n1', s), R_{binW}(v1, v2, n1', n2')$$

We then evaluate the conjunctive query CQ_T for each query template $\mathcal{Q}$ based only on R_L and R_R, and we compare the join processing cost of MMQJP without view materialization and the cost of MMQJP with view materialization. For the latter, we also measure the time cost of computing R_{vj}, R_L and R_R, respectively.

The experiments are performed on both the two-level and the three-level document schema. We use the default values for all parameters above, except that we set the number of queries to 100000. The results on the two-level and the three-level document schema and shown respectively in Figure 14 and 15.

Since according to the experiment setup, R_{bin} and R_{doc} only contain information for a single document, d_1, the materialization costs of R_{vj}, R_L and R_R are small compared to the join processing cost. However, we expect that the materialization cost of R_L could potentially be large in real stream settings, since R_{bin} might contain many tuples produced by the XPath Evaluator from previous events. Therefore the benefit of materializing slices of $R_{\hat{L}}$ for computing R_L, instead of recomputing R_L from scratch when processing each event should be significant. Also, in this experiment, we assume we can afford the space to materialize the entire R_L. In practice we may only be able to materialize some slices of R_L, in which case view cache replacement policies may be involved, as was mentioned in Section 5.

The results show great benefits from evaluating conjunctive queries by first materializing these relations. This is especially true for the case of the three-level document schema, where we have significantly more query templates compared to the two-level schema (22 templates for complex schema versus 6 for the simple schema). Materializing these relations enables sharing of computation among the conjunctive queries for different query templates; therefore, the more query templates we have, the more benefits we receive from view materialization.

6.3 XSCL Queries over RSS Streams

We evaluate the performance of MMQJP and sequential evaluation of XSCL queries over (RSS and Atom) feed streams. The feeds we use in this experiment are collected from 418 channels over a period of time from June to October in 2006. There are a total of 225K items in the feed. Each feed item has a simple document schema similar to the schema in Figure 2. Specifically, it has five leaf nodes tagged `item_url`, `channel_url`, `title`, `timestamp` and `description`.

We randomly generate queries in the same way as in Section 6.1. We assign a time window of ∞ to all the generated queries. This means in processing the 225K feed items, no feed item will be discarded from the join state.

Processing XSCL queries over streams involves both the XPath Evaluator and the Join Processor. We evaluated the XPath expressions corresponding to the XPath query blocks we generated on YFilter, an instance of the XPath Evaluator, and we found the time cost of XPath processing over the entire stream in YFilter is about 15 seconds, which is significantly less than the time cost in join processing (using either MMQJP or sequential evaluation).[4] Therefore, the join processing is the bottleneck of the overall XSCL query processing, and in the following text we focus on measuring the cost of join processing.

To run stream processing experiments on a relational database, we perform the following operations for each feed item. First, we issue bulk load statements to load the data of the current feed item into R_{binW} and R_{docW}. The way we generate R_{binW} and R_{docW} is similar to the way we described in Section 6.1. We do not include the loading cost in our numbers, since that cost will be negligible in a real main memory based implementation. Next, we evaluate the conjunctive queries, and measure their costs. We then move data from R_{binW} to R_{bin}, and R_{docW} to R_{doc} with SQL statements, however for the same reason as before we also do not include this cost in our overall numbers. We run MMQJP with and without view materialization and also compare to Sequential. We report the total time cost of evaluating conjunctive queries over all the items in the web feed stream.

According to the this setup, there are five different query templates in MMQJP. For each feed item, SQL Server needs to evaluate the SQL queries corresponding to CQ_T for each $\mathcal{Q}$ defined in Section 4.4. This means over a stream of S events, the number of queries to evaluate for MMQJP will be $5S$. However, since there is a fixed overhead in the order of tens of milliseconds in submitting an SQL query to a secondary-storage based relational database engine, a measurement of the total cost of evaluating these $5S$ SQL queries will not reflect the real throughput of a publish/subscribe system. Therefore, instead of evaluating the conjunctive queries for query templates once for each feed item, we batch the join processing by loading a set of feed items into R_{binW} and R_{docW} at one time and perform the joins. This significantly reduces the total number of SQL queries to evaluate. Due to space constraints, we omitted the details for this step.

The throughput of MMQJP compared to sequential evaluation while varying the number of queries is reported in Figure 16. MMQJP demonstrates impressive throughput with a large number of queries. View materialization helps further by enabling sharing of computation among different query templates. The throughput of MMQJP with or without view materialization stays flat after the number of queries grow beyond 10000, since there are only thousands of distinct queries according to our query generation scheme — after generating 10000 queries, almost all queries generated later on are duplicates. This is consistent with our assumption about the workload. Note that we recompute R_L from scratch for every batch in this experiment, since we did not materialize slices of $R_{\hat{L}}$. Therefore, we expect the throughput of MMQJP with view materialization to be even higher if that is done. The experimental results

[4]The YFilter implementation we use is based on Java; still its XPath evaluation cost is much smaller compared the join processing cost measured in SQL Server.

where we vary the parameter of the Zipf distribution are similar, and we thus omit them from the paper due to space constraints.

7. RELATED WORK

XML Stream Processing. Our work is the first to address both expressiveness in query language and scalability in system throughput for XML publish/subscribe systems. There has been a large body of work on XML query processing, each addressing parts of these challenges [10, 16, 13, 22]. YFilter [13], XPush [17] and XSQ [25] are based on variants of finite-state automata, and support a significant portion of XPath 1.0 for stream processing. They however do not support queries joining multiple documents or streams. Other XML pub/sub work on more expressive XML query languages has focused on specific optimizations for a small number of queries [21, 20, 6]. Our MMQJP techniques can potentially be combined with these optimization techniques in an XML publish/subscribe system. Examining this is part of our future work.

Other Related Work. Traditional pub/sub systems [4, 28, 14] sacrifice expressiveness to achieve high performance. For example, Le Subscribe [14] is a highly scalable pub/sub system. More recently, Cayuga [12] and SASE [27] propose stateful publish/subscribe systems for complex relational event processing. Data streams have attracted considerable attention in the database community in recent years. Existing DSMSes concentrate on processing of complex relational queries and do not explore multi-query optimization in depth [7, 23, 9, 2].

8. CONCLUSIONS

We have presented Massively Multi-Query Join Processing (MMQJP) techniques, which efficiently process large numbers of continuous inter-document queries over XML streams. Though not elaborated in this paper, it is easy to see that our approach is also applicable to continuous query processing over relational streams.

There are many avenues for future work. First, we would like to build an expressive publish/subscribe system based on MMQJP techniques, capable of processing both relational and XML streams with a large number of continuous queries. Second, in this paper we have explored a sweet spot in the expressiveness/scalability spectrum between XPath and XQuery stream processing; in the future we would like to push this sweet spot towards supporting larger subsets of XQuery in stream settings.

9. ACKNOWLEDGEMENTS

We thank Stefanie Scherzinger and Łucja Kot for reviewing drafts of this paper, Lars Brenna for providing the RSS data for our experiment, and Manuel Calimlim for the tuning suggestions on Microsoft SQL Server 2005.

10. REFERENCES

[1] Xpath leashed. http://www-db-out.bell-labs.com/user/benedikt/papers/leashed.ps.gz.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *Proc. CIDR*, pages 277–289, 2005.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. 1995.

[4] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. PODC*, pages 53–61, 1999.

[5] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. VLDB*, pages 53–64, 2000.

[6] C. Barton, P. Charles, M. Fontoura, V. Josifovski, D. Goyal, and M. Raghavachari. Streaming xpath processing with forward and backward axes. In *Proc. ICDE*, 2003.

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. VLDB*, 2002.

[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. VLDB*, pages 606–617, 1994.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. CIDR*, 2003.

[10] Y. Chen, S. Davidson, and Y. Zheng. An efficient xpath query processor for xml streams. In *Proc. ICDE*, 2006.

[11] Byron Choi. What are real dtds like. 2002.

[12] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.

[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.

[14] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. SIGMOD*, pages 115–126, 2001.

[15] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, and G. Agrawal. The bea/xqrl streaming xquery processor. In *Proc. VLDB*, 2003.

[16] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based xml packets filtering for millions of path queries. In *Proc. ICDE*, 2005.

[17] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM TODS*, 29(4):752–788, 2004.

[18] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. Technical report, Cornell University, 2007. http://techreports.library.cornell.edu.

[19] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. ICDE*, 2003.

[20] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proc. VLDB*, pages 228–239, 2004.

[21] X. Li and G. Agrawal. Efficient evaluation of xquery over stream data. In *Proc. VLDB*, pages 265–276, 2005.

[22] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based xml query processor. In *Proc. VLDB*, 2002.

[23] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, 2003.

[24] Ed Jr. Pegg. Graph minor. http://mathworld.wolfram.com/GraphMinor.html.

[25] F. Peng and S. Chawathe. Xsq: A streaming xpath engine. *ACM TODS*, 30(2):577–623, 2005.

[26] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. PODS*, pages 263–274, 2004.

[27] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. SIGMOD*, 2006.

[28] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *Proc. CIDR*, 2003.