Automaton Meets Query Algebra: Towards A Unified Model for XQuery Evaluation over XML Data Streams

Jinhui Jian, Hong Su, and Elke A. Rundensteiner

Department of Computer Science, Worcester Polytechnic Institute
Worcester, MA 01609, USA
{jian, suhong, rundenst}@cs.wpi.edu

Abstract. In this work, we address the efficient evaluation of XQuery expressions over continuous XML data streams, which is essential for a broad range of applications including monitoring systems and information dissemination systems. While previous work has shown that automata theory is suited for on-the-fly pattern retrieval over XML data streams, we find that automata-based approaches suffer from being not as flexibly optimizable as algebraic query systems. In fact, they enforce a rigid data-driven paradigm of execution. We thus now propose a unified query model to augment automata-style processing with algebra-based query optimization techniques. The proposed model has been successfully applied in the Raindrop stream processing system. Our experimental study confirms considerable performance gains with both established optimization techniques and our novel query rewrite rules.

1 Introduction

XML has been widely accepted as the standard data representation for information exchange on the Web. Two camps of thoughts have emerged on how to deal with ubiquitous data. The "load-and-process" approach preloads the data (e.g., a complete XML document) into a persistent storage and only thereafter starts processing. The "on-the-fly" approach instead processes the data while it is being received. In this work, we adopt the second approach and address the efficient evaluation of XQuery expressions over continuous XML data streams. Such capability is essential for a broad range of applications, including monitoring systems (e.g., stock, news, sensor, and patient information) and information dissemination systems.

While previous work [2, 8, 14, 11] has shown that automata theory is suited for XPath-like pattern retrieval over token-based XML data streams, we find these automata-based approaches suffer from being not as flexibly optimizable as, for example, traditional database systems that are based on query algebras [3, 6]. We thus now propose a unified model to augment automata-style processing with algebra-based query optimization techniques. It is our goal to exploit the respective strength inherent in these two paradigms and to bring them together into one unified query model.

We shall focus on flexible integration of automata into query algebras. To the best of our knowledge, we are the first to tackle this flexible integration problem. Previous work applies automata techniques such as NFAs or DFAs in a rather fixed fashion that impairs the potential of query optimization. For example, [11] treats the whole automaton, which is used to scan all XPath-like patterns, as one single query operator, that is, as a "black box" with multiplexed yet fixed functionality. This has some disadvantages. First, it disallows or at least hinders pulling out some of the pattern scans from the automaton. In fact, the trade-off between moving query functionality into and out off the automata is one theme of this paper. Second, because multiple pattern scans are encoded in one single operator, the relationship between these scans is hidden from the topology of a logical plan. Hence traditional query optimization techniques such as equivalent rewriting [3, 6] are not directly applicable. In contrast, we attempt to open the "black box" and create a logical "view" within our algebraic framework. The key advantage of our approach is that it allows us to refine automata in the same manner as refining an algebraic expression (i.e., with equivalent rewriting).

We have implemented a prototype system based on the proposed query model to verify the applicability of established optimization techniques [3, 6] in the context of query plans with automata constructs. We have devised a set of query rewriting rules that can flexibly move query functionality into and out off the automata. All these optimization techniques allow us to reason about query logic at the algebraic level and only thereafter play with the implementation details. Our experimental study confirms considerable performance gains achievable from these optimization techniques.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 presents the overall approach. Sections 4, 5, and 6 describe the three layers in our model, respectively. Section 7 presents the experimental results and our analysis. Section 8 summarizes our conclusion and possible future work.

2 Related Work

The emergence of new applications that deal with streaming XML data over the Internet has changed the computing paradigm of information systems. For example, dissemination-based applications [2] now require a system to handle user queries on-the-fly. [2, 8] have adopted automata-based data-driven approaches to deal with this new requirement. Following this data-driven paradigm, [11] employs a set of Finite Machines, [5] employs a *Trie* data structure, and [14] employs Deterministic Finite Automata (DFA) to perform pattern retrieval over XML streams. These recent works all target some subset of XPath [16] as the query language. With the XQuery language [17] emerging as the *de facto* standard for querying XML data, however, more and more XML applications adopt XQuery to express user requests. This more complex language raises new challenges that must be met by new techniques.

[13] applies the concept of an *extended transducer* to handle XQuery. In short, an XQuery expression is mapped into an XSM (XML Stream Machine). The XSM is consequently translated into a C program and compiled into binary code. The incoming data is then streamed through this system at run time. Note that the data-driven nature of execution is retained. We speculate that this rigid-mapping approach has scalability and optimization problems. For example, it is not clear how such a model can be flexibly extended to support multiple queries. It is also unstudied what query optimization techniques can be applied to this rigid automata-based model.

On the other hand, the traditional database literature [7, 3, 6, 9] has advocated algebraic systems as the suitable foundation for query optimization and evaluation. While a number of recent papers have proposed algebras for XML query evaluation [4, 19, 18, 12, 11], none address flexible integration of automata theory into query algebras. [4, 19, 18] focus on querying XML views over relational databases. [12] queries native XML data. Neither can process streaming data. [11] applies automata theory but integrates the automata in a rather fixed fashion, as discussed in Section 1.

3 The Raindrop Approach

Our approach faces an intricate trade-off. On the one hand, we want to exploit automata theory to process streaming XML data on-the-fly. On the other hand, we need to overcome the limitations imposed by the automata model and instead exploit query algebra for optimization.

While a number of recent papers [2, 8, 11, 14, 13] have shown that automata theory is suitable for XML stream processing, we now analyze the limitations of automata in terms of query optimization. Automata such as NFA, DFA, and transducer models enforce data-driven execution, which implies an underling token-based data model. In the XML context, a token is typically represented as a piece of XML data, such as an open tag, a close tag, or a piece of character data (PCDATA). This token-based data model is different from the one adopted in the XQuery language, which instead is a sequence of node-labeled trees [17] or a collection of tree fragments [12]. This mismatch is two-fold. First, tokens are sequential and discrete, while trees are connected and have an internal structure. Second, tokens and trees have different granularities in terms of abstraction.

From the query optimization point of view, mixing these heterogeneous models complicates the optimization. First, the design of operators is more complex because the operators now must process more complex mixed-typed objects. Second, the search for optimal plans may be more expensive because this new mixed data model introduces a new dimension to the search space. Third, different operators may require data types in different data models, which impairs the uniformity of the query model.

Considering the need for both data models and the limitations of arbitrarily mixing them, we now propose a three-layer hierarchical model to resolve this dilemma. The top layer represents the semantics of query plans and thus is

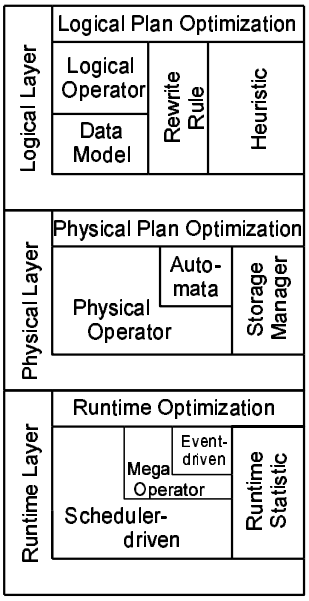

Fig. 1. Three layers in the unified query model

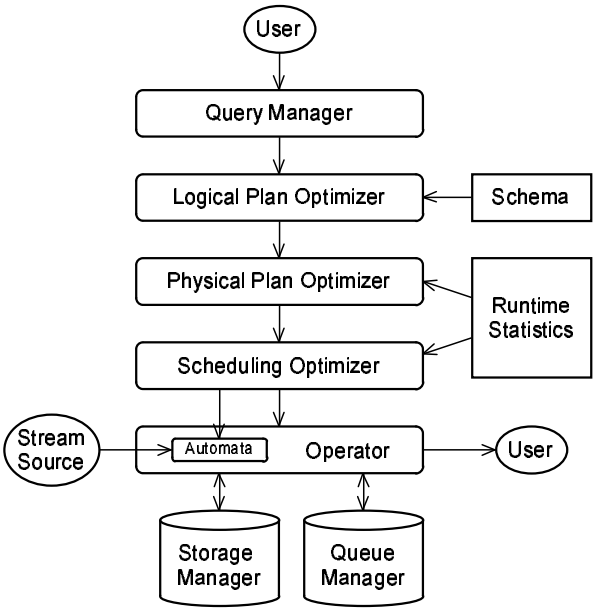

Fig. 2. System components

called logical layer (see Section 4. It serves specifically as the basis for plan optimization, i.e., to find efficient query plans. We adopt the tuple model in this layer to simplify the design of operators and particularly to leverage various established query optimization techniques developed specifically for the tuple model [3, 6]. Note that while the token model is hidden from this layer, the automata flavor of the system is still embodied in, for example, the Extract operators in Figure 4. Hence, we can refine the automata in the same manner as we refine algebraic query plans, thus overcoming the limitations imposed by automata techniques.

The physical layer refines the logical query plan by specific algorithms to implement the functionalities of logical operators (see Section 5. In particular, the physical layer describes how automata techniques are used to implement the *automata operators* such as Extract and Structural Join. The token-based data model, which is hidden from the logical layer, is made explicit inside each *automata operator*. Note that this is a hierarchical design where the logical layer describes the overall query plan and the functionality of each operator, while the physical plan describes the internal implementation of each individual operator. Although the token model is made explicit in the physical layer, it is restricted to being exposed only inside each individual operator. It does not impair the homogeneity of the data model at the logical layer.

The runtime layer describes the overall execution strategy for a query plan. It specifies the control and coordination for each physical operator. In particular, we

devise a two-level control mechanism that integrates the data-driven execution strategy common for automata with more flexible execution strategies such as the the traditional iterator-based evaluation strategy or the more recently proposed scheduler-driven strategies. We employ the concept of *mega operator* to bridge these two execution levels, as explained in Sectionruntime.

4 The Logical Layer

4.1 The Data Model

Because [17] has defined the *XQuery data model* for query evaluation, we now first explain why we in addition need yet another data model. According to [17], a data model defines the logical view of (1) the source data and (2) the intermediate data of query expressions (i.e., the inputs and outputs of query operators). In the *XQuery data model*, source data is defined as node-labeled trees augmented with node identity, while intermediate data is defined as a sequence of zero or more items, each either a node or an atomic value.

We cannot directly adopt the *XQuery data model* to query streaming XML data. First, streaming XML data can be more naturally viewed as a sequence of discrete tokens, where a token can be an open tag, a close tag, or a PCDATA. In fact, the node-labeled tree view of an XML data stream is incomplete until after the stream is wholly received and parsed.

Second, the *XQuery Data Model* is not suitable for pipelining the execution. To make it clear, we shall draw a comparison with the *relational data model* [7], which is based on sets of tuples. A relational algebraic query plan usually ignores whether it will be executed in a pipelining fashion or in an iterative fashion. In other words, the execution strategy is left out off the logical query model. This design is feasible, however, only because the *relational data model* has a natural atomic execution unit, i.e., a tuple. A query plan executor can choose whatever execution strategy without breaking this atomic unit. In the XML realm, however, such an atomic unit is not directly available because of its arbitrarily nested structure. In fact, many approaches such as the one suggested by the *XQuery data model* consider the complete XML document as a unit, and thus exclude the possibility of pipelined execution. Going to another extreme, [13] considers each XML data token as an atomic unit. As discussed in previous sections, this purely token-based approach is rigid and at times too low-level.

Based on the above observations, we now define our logical data model. We shall adopt <tag> to denote XML tags, [...] for a list (or a sequence), $\langle ... \rangle$ for a *tuple*, with each *field* in the *tuple* bound to an *attribute name* (i.e., the *named perspective* in [1]). We shall use $\circ$ for *tuple concatenation* and π for *tuple projection*.

1) We define the source XML streams to be sequences of *tokens*, where a *token* can be an open tag, a close tag, or a piece of *character data* (PCDATA). Formally, we define $\mathcal{T}$, the domain of *tokens*, as:

$$\mathcal{T} = \{< x > | x \in \mathcal{E}\} \cup \{< /x > | x \in \mathcal{E}\} \cup \{d | d \in \mathcal{D}\}$$

where $\mathcal{E}$ is the domain of *XML element names* and $\mathcal{D}$ is the domain of *character data* (strings).

2) We define intermediate data of query expressions (i.e., the inputs and outputs of query operators) to be a sequence of zero or more *tuples*, with each *field* in a *tuple* being a sequence of zero or more *items*. Each *item* is either an *XML node* or an *atomic value*. Formally, we define $\mathcal{P}$, the domain of *tuples* as:

$$\mathcal{F} = \{\ [v_1, ..., v_n] \mid v_i \in \mathcal{A} \cup \mathcal{X}, n\ is\ the\ size\ of\ a\ field\}$$
$$\mathcal{P} = \{\langle f_1, ..., f_n\rangle | f_i \in \mathcal{F}, n\ is\ the\ arity\ of\ a\ tuple\}$$

where $\mathcal{F}$ is the domain of *fields*, $\mathcal{A}$ is the domain of *atomic values*, and $\mathcal{X}$ is the domain of *XML nodes*.

4.2 The Logical Operators

Following the data model defined above, every logical operator accepts a sequence of tuples as input and produces a sequence of tuples as output. We distinguish between two classes of operators. The *pattern retrieval operators*, such as Navigate, Extract, and Structural Join, are designed to locate XML elements specified by XPath expressions. The *filtering and construction operators*, such as Select, Join, and Tagger, are designed to filter data and construct nested data fragments. Considering the limitation of space, here we only show details of the *pattern retrieval operators*. For the *filtering and construction operators*, readers are referred to [19]. For illustrative purposes, we shall follow a simple running example as shown in Figures 3 and 4.

```
<articles>
    <article id="1">
        <name> item1 </name>
        <review>  <rate>5</rate>  </review>
        <review>  <rate>3</rate>  </review>
    </article>

    <article id="2">
        <name> item2 </name>
        <review>  <rate>4</rate>  </review>
    </article>

    <article>
        ...
</articles>
```

```
FOR $a in document
      ("articles.xml")//article
WHERE  $a/review/rate = 5
RETURN
  <best>
    $a/name
  </best>
```

Fig. 3. Example document and query

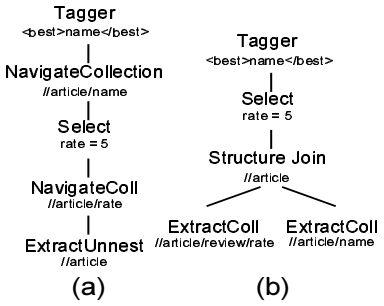

Fig. 4. Two equivalent query plans

NavigateUnnest $\Phi_{ep,path}(\ T\ ) = [\ t \circ\ \langle f\rangle \mid t \leftarrow T, f \leftarrow follow(\pi_{ep}(t), path)\ ]$

The *follow* operation denotes the "descendant" relation in [16] and returns a sequence of descendent elements of the first argument. The NavigateUnnest operator finds all descendants of the *entry point* ep in *tuple* t that conform

to the XPath expression *path*. An output tuple is constructed for each such descendant by inserting the descendant as a field into the input tuple. For example, let tuple list $T = [\ \langle article \rangle\ ]$, where "article" is the first article element in Figure 3, and let $a denote the attribute name of "article", then $\Phi_{\$a,//article/rate}(\ T\ ) = [\ \langle\ article, r_1\ \rangle, \langle\ article, r_2\ \rangle\ ]$, where "r1" and "r2" are the two "rate" descendants of "article".

NavigateCollection $\phi_{ep,path}(\ T\ ) = [t \circ\ \langle f \rangle \mid t \leftarrow T, f = [\ f' \mid f' \leftarrow follow(\pi_{ep}(t), path)\ ]\ ]$

This operator again finds all descendants of the *entry point* ep in *tuple* t that conform to the XPath expression *path*. One collection, constructed from all descendent elements, is then concatenated to the input tuple as a field. Following the above example, $\phi_{\$a,//article/rate}(\ T\ ) = [\ \langle\ article, [\ r_1, r_2\ ]\ \rangle\ ]$.

ExtractUnnest $\Psi_{str,path}(\ ) = [\ \langle n \rangle \mid n \leftarrow follow(str, path)]$

This operator identifies from the input data stream *str* (i.e., the root element) all XML elements that conform to the given XPath expression *path*. A new tuple is generated for each matched element. For example, let the stream name of the document in Figure 3 be "articles.xml", $\Psi_{\text{“}articles.xml'',//article/review/rate}(\ ) = [\ \langle r_1 \rangle, \langle r_2 \rangle, ..., \langle r_n \rangle\ ]$, where $r_1, r_2, ...r_n$ are "rate" elements conforming to "//article/review/rate".

An *ExtractUnnest* operator does not take in any input tuples. Instead, as will be further explained in the physical layer, the operator's internal mechanism, namely its associated automaton, is responsible for analyzing the input stream and extracting the desired data. This association with the automaton does not affect the semantics of the operator; it is simply an implementation issue. Hence optimization techniques such as query rewriting are not restricted by the data model and the execution strategy that are imposed by automata theory.

ExtractCollection $\psi_{str,path}(\ ) = [\ \langle n \rangle \mid n =\ [\ n' \mid n' \leftarrow follow(str, path)]\ ]$

This operator identifies from the input data stream *str* all XML elements that conform to the XPath expression *path*. A collection, composed of all such elements, is then used as the only field for the output tuple. For example, $\Psi_{\text{“}articles.xml'',//article/review/rate}(\ ) = [\ \langle\ [\ r_1, r_2, ..., r_n]\ \rangle\ ]$. Similar to an *ExtractUnnest* operator, an *ExtractCollection* operator does not take in any input tuples, but instead generates the output using its internal mechanism.

Structural Join $X \bowtie_{path} Y = [t_x \circ t_y \mid t_x \leftarrow X; t_y \leftarrow Y; precede(t_x, path) = precede(t_y, path)]$

The *precede* operation denotes the "ancester" relation in [16]. The Structural Join operator concatenates input tuples based on their structural relationship such as common ancestor. For example, $[\ \langle\ [\ r_1, r_2, r_3]\ \rangle\ ]\ \bowtie_{//article}\ [\ \langle\ [\ n_1, n_2]\ \rangle\ ]\ = [\ \langle\ [\ r_1, r_2\ ], n_1\ \rangle, \langle\ r_3, n_2\ \rangle\ ]$

Although a *Structural Join* operator can be implemented like a traditional join, i.e., by value comparisons, it can be more efficiently implemented using an associated automaton. Again this association does not affect the operator's formal semantics. Hence we can take advantage of automata theory without being restricted by its implied limitations in terms of optimization.

4.3 The Potential Query Rewriting

Now we discuss the optimization opportunity provided by the proposed algebra. Due to the space limitation here we only discuss rewrites on *pattern retrieval operators*. Readers are are referred to [19] for other rules. The two plans in Figure 4 are equivalent. Plan 4(a) uses an Extract operator to grab "article" elements and two Navigate operators to identify proper descendants of the "article". This is a top-down dissection process in which higher level XML nodes are first grabbed and then lower level XML nodes are identified by navigating into the higher lever node. In contrast, two Extract operators in plan 4(b) first grab lower level nodes and a Structural Join operator then joins the lower level nodes by matching their common parent. This can be viewed as a bottom-up construction process.

Formally, Plan 4(a) can be transfromed into Plan 4(b) using the following rules:

$$\phi_{\$a,p_1}(\phi_{\$a,p_2}(\psi_{str,p_3})) = \psi_{str,p_1} \bowtie_{p_3} \psi_{str,p_2}$$
$$\sigma_{cond}(\phi_{\$a,p}) = \phi_{\$a,p}(\sigma_{cond})$$

Where the attribute name \$a denotes the output of ψ_{str,p_3}, the XPath expression p_3 is the common prefix of p_1 and p_2, str is an XML data stream, σ denotes a Select operator, and the condition *cond* is independent of the attribute \$a.

5 The Physical Layer

5.1 Applying Automata

We resort to NFA to implement the *automata operators* such as Extract and Structural Join. Each *automata operator* is associated with an NFA. Each NFA is constructed to recognize one given XPath expression. All related NFAs, i.e., whose XPath expressions are based on the same input stream, are merged into one single NFA for sharing computation. Such NFA construction and merging is a well-studied topic in the literature [8]. We differ from previous work, however, in the operations undertaking by the sequence of accepting events.

Figure 5 shows a partial plan with its associated NFA (after merging). State 3 together with all its preceding states is constructed to recognize the expression "//article/name". For each "name" element, State 3 will be activated twice, one for the open tag "<name>" and the other for the close tag "</name>". The associated operator (i.e., the one connected to State 3 as shown in Figure 5) is invoked by the close tag. This seemingly simple scheme is actually very powerful when multiple operators are invoked in a specific order, as will be shown below.

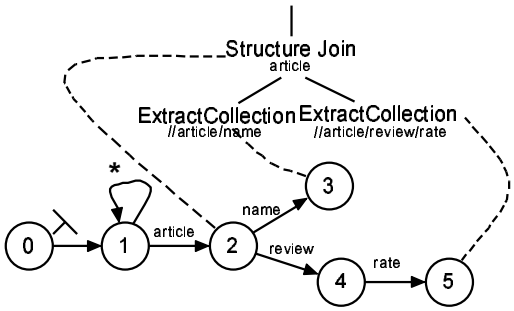

Fig. 5. Physical operators and their associated automaton

5.2 Implementing Physical Operators

Extract An Extract operator has two steps: first assembling input tokens that conform to a given XPath expression into XML elements and second encapsulating the elements into tuples. Although each Extract operator may look for elements conforming to different XPaths, these elements may often overlap. Take the document in Figure 3 for example. There may be an Extract operator looking for "//article" and another for "//article/review/rate", i.e., the former element contains the latter. It is naive to store them separately. Hence we adopt a centralized approach to perform the first step.

An XML element can be viewed as a token connected to other tokens. Hence the assembly process is a matter of connecting proper tokens. Take the plan in Figure 5 for example. Suppose the open tag "<article>" comes from the stream, and we are now at state 2. It is easy to conclude that whatever next open tag is encountered, it must be a child of "article". In general, a *context node* is maintained during runtime and is set to the document root at the initialization. Every incoming open tag is connected to the *context node* and becomes the new *context node*; every incoming close tag resets the *context node* to the previous *context node* by referring to a *context stack*.

The second step, i.e., encapsulating proper elements into tuples, is performed by the Extract operators. Recall that *automata operators* are invoked by close tags. Take Figure 5 as example. The Extract operator at the left hand side is associated with state 3, which will invoke the operator when a "</name>" arrives. Note that all descendent tokens of a particular "name" element, e.g., its *Text*, must come between "<name>" and the corresponding "</name>" tags. Hence before invoking the Extract operator, we would have assembled a complete "name" element by the first step. This element will be passed to the operator and will be put into a tuple.

Structural Join [12] points out that "an efficient implementation of structural join is critical in determining the overall performance of an XML query processing system". While naive structural join algorithms would take a high-order

polynomial time, we now propose the JIT (Just-In-Time) Structural Join algorithm. This join algorithm exploits the sequentiality of stream tokens and takes linear time.

The algorithm is simple. When a Structural Join operator is invoked by its associated NFA, it makes a cross product out of all its inputs. The cross product is guaranteed to be the correct output. The trick is the timing of invocation. Take the document in Figure 3 and the plan in Figure 5 as example. The Structural Join is first invoked on the first "</article>". At this time, the output of the left Extract is $\langle n_1 \rangle$, and the output of the right Extract is $\langle [\ r_1, r_2\ ] \rangle$, where n_1, r_1, and r_2 are defined in Section 4.2. It is obvious that n_1, r_1, and r_2 are descendants of the first "article" element. Hence the cross product $\langle n_1, [\ r_1, r_2\ ] \rangle$ is the correct output. Similarly, every consequent invocation of the Structural Join must have descendants of the current "article" element as input, because descendants of the previous "article" element would have been consumed by the previous invocation of the Structural Join and descendants of future "article" elements have not yet come. Because no value comparison is involved, the complexity of the JIT join is equal to the complexity of output tuple construction, hence linear in the output size.

Other Operators Other operators such as Join, Select, and Navigate are rather generic and can be found in other XQuery engines [19, 18]. We skip further description here.

6 The Runtime Layer

A general assumption in stream systems is that data arrival is unpredictable [15]. This characteristic makes the demand-driven execution strategies [10] not directly applicable in the stream context. One solution is to let the incoming data "drive" the execution, which leads to a purely data-driven (or event-driven) execution strategy. This approach is adopted by for example [13]. The disadvantage is its rigidity: every incoming data token will trigger a fixed sequence of operations immediately. This rigidity excludes the possibility of deferring certain operations and then batching their processing, which may be more efficient because the cost of context switching between operators can be reduced.

Another solution [15] uses a *global scheduler* who calls the *run* methods of query operators based on a variety of scheduling strategies. The disadvantage is its over generality. In principle the scheduler-driven strategy subsumes the data-driven strategy, i.e., the general strategy can simulate the rigid one. It is easy to conceive, however, that this simulation may be less efficient than directly applying the data-driven strategy because of the cost in making such scheduling decisions.

In this paper, we adopt a hierarchical approach to integrate both the data-driven and the scheduler-driven strategies by employing a *mega operator*. We organize a physical operator plan into two levels as shown in Figure 6. The top level plan is composed of all *non-automata operators*, while the bottom level is

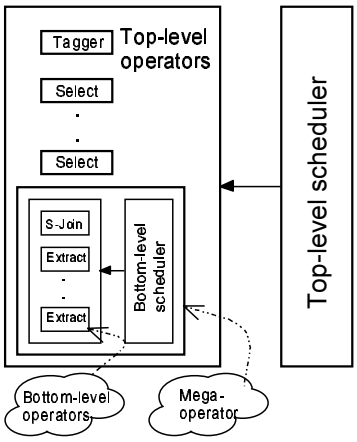

Fig. 6. Two-level scheduling

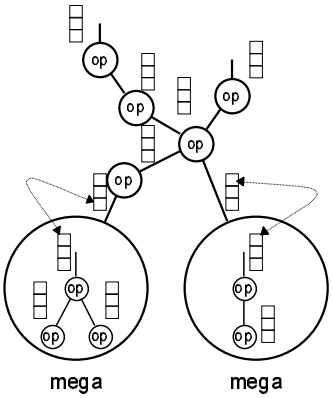

Fig. 7. Two-level data passing

composed of *automata operators*. A *mega operator* encapsulates a set of automata operators at the bottom level and represents this subplan as one operator at the top level, as shown in Figure 7. Each bottom level plan is controlled by an NFA, while the top level plan is flexibly governed by the global scheduler. When a *mega operator* is invoked by the global scheduler, it takes the outputs from the bottom level plan and transfers them to the top level plan.

7 Experimental Evaluation

Our evaluation studies the trade-offs between moving query functionality into and out off the automata. This is to be compared with previous work [8, 11] which offers no such flexibility and instead assumes (or even advocates) maximal pattern retrieval push-down.

We have implemented our prototype system with Java 1.4. All experiments are conducted on a Pentium-III 800Mhz processor with 384 MB memory running Windows 2000. The JVM is initialized with 256 MB heap space. Garbage collection is explicitly invoked to avoid interference. Every experiment is repeated at least 10 times.

The test data is synthetically generated similar to the document in Figure 3. The data size is 9.0 MB. The user queries are similar to the one in Figure 3, with slight modifications on the *RETURN* clause to include more path bindings. Accordingly, the query plans are similar to those in Figure 4. We call plan (a) "bottom-most navigation only"(BMNO), and plan (b) "all navigation"(AN). We make another plan using equivalent rewrite rules to push the selection into the automata, which is called "all navigation and selection"(ANS).

Our first experiment compares the three push-down strategies. Figures 8 and 9 illustrate their performances in different settings, i.e., different workloads and selectivities. BMNO wins in Figure 8 (i.e., the output rate is higher and

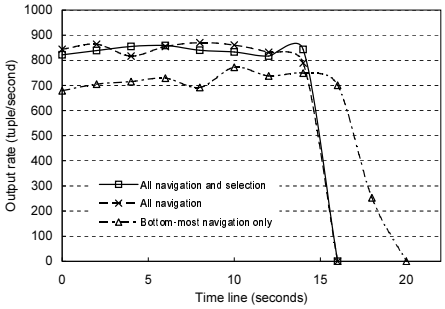

Fig. 8. Output rate with low workload and high selectivity

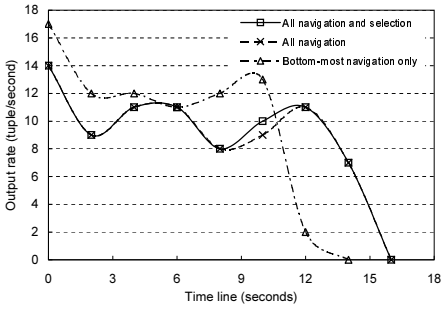

Fig. 9. Output rate with high workload and low selectivity

the finishing time is shorter) while AN and ANS win in Figure 9. Clearly both workload and selectivity affect the result. This justifies our effort to allow for flexible query rewriting even in this new automata context.

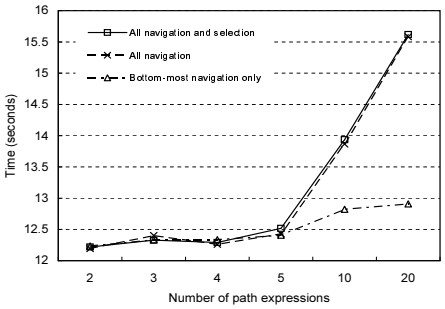

Fig. 10. Comparing different workloads with selectivity = 5%

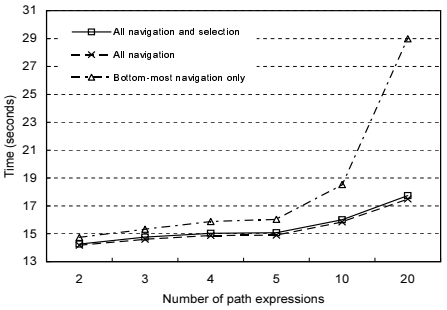

Fig. 11. Comparing different workloads with selectivity = 90%

Our second experiment analyzes the three strategies in terms of different workloads, in particular, different numbers of path expressions in a query. Figures 10 and 11 illustrate the same interesting pattern as the first experiment. But this time we see that with low selectivity (Figure 10), BMNO outperforms AN and ANS. In fact, higher workloads further enlarge the gap between them. In contrast, with high selectivity (Figure 11), AN and ANS outperform BMNO. The gap also increases with a higher workload. Intuitively, this is because automata are efficient in pattern retrieval, especially when the retrieval of several patterns is encoded into one automaton. Hence in Figure 11 the more aggressive push-down strategies (AN and ANS) outperform the less aggressive one (BMNO). However, evaluating all bindings together in the automaton disallows for early selection, a well-established optimization technique in the database literature.

For example, in the query plan shown in Figure 4 (b), the Selection is evaluated before the top-most Navigation. Hence we can reduce the cost of Navigation to all "articles" elements without a "rate-5" review. When selectivity is low, such optimization can save a lot of work, as illustrated in Figure 10.

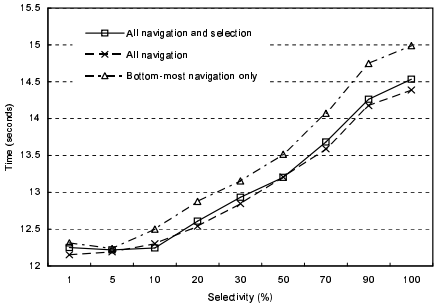

Fig. 12. Comparing different selectivity with numBinding = 2)

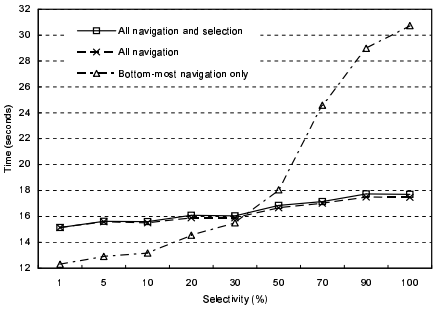

Fig. 13. Comparing different selectivity with numBinding = 20

We now look more closely at the relationship of selectivity and performance. With low workloads, Figure 12 shows that the performance difference is small. But with higher workloads, this difference is enlarged, as in Figure 13. We can see clearly that with a selectivity less than 40%, BMNO outperforms the other two. The smaller the selectivity, the larger the difference. With a selectivity greater than 40%, both AN and ANS outperform BMNO. Their difference also increases when the selectivity becomes larger. This is the trade-off we expect and which our unified framework is empowered to exploit.

8 Conclusion

We have presented a unified model for efficient evaluation of XQuery expressions over streaming XML data. The key feature of this model is its power in flexibly integrating automata theory into an algebraic query framework. Unlike any of the previous work, this power facilitates various established query optimization techniques to be applied in the automata context. It also allows for novel optimization techniques such as rewrite rules that can flexibly change the functionality implemented by the automata-based operators. Our experimental study confirms that these optimization techniques indeed result in a variety of interesting performance trade-offs that can be exploited for efficient query processing. However, this paper serves only as the first step towards a full-fledged framework for XML stream systems. It is a solid foundation for making various optimization techniques possible in the XML stream context.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
2. M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.
3. M. Astrahan et al. System R: a relational approach to database management. *ACM Trans. on Database Systems*, pages 97–137, 1976.
4. M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, pages 105–110, 2000.
5. C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. ICDE*, pages 235–244, 2002.
6. S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pages 34–43, June 1998.
7. E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
8. Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of ICDE*, pages 341–344, 2002.
9. L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed XML data. In *CIKM*, pages 126–133, 2002.
10. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, pages 73–170, June 1993.
11. Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4), 2002.
12. H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.
13. B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. VLDB*, pages 215–226, 2002.
14. G. Miklau, T. J. Green, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *ICDT*, pages 173–189, 2003.
15. R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, pages 245–256, 2003.
16. W3C. XML path language (xpath) version 1.0. http://www.w3.org/TR/xpath, November 2002.
17. W3C. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, November 2002.
18. X. Zhang, K. Dimitrova, L. Wang, M. EI-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. Rainbow II: Multi-XQuery optimization using materialized xml views. In *SIGMOD Demonstration*, page 671, June 2003.
19. X. Zhang, B. Pielech, and E. A. Rundensteier. Honey, I shrunk the XQuery! — an XML algebra optimization approach. In *Proceedings of the fourth international workshop on Web information and data management*, pages 15–22, Nov 2002.