Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams

Hong Su, Elke A. Rundensteiner and Murali Mani

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

1 Introduction

Our *Raindrop* framework [6, 9] aims at tackling challenges of stream processing that are particular to XML. In contrast to the tuple-based or object-based data streams, XML streams are usually modeled as a sequence of primitive tokens, such as a start tag, an end tag or a PCDATA item. Unlike a *self-contained* tuple or object whose semantics are completely determined by its own values, a token lacks semantics without the *context* provided by other tokens in the stream. This poses specific challenges for query processing over such XML streams.

State-of-the-Art. Since the *automata model* was originally designed for matching patterns over strings, it is a natural paradigm for structural pattern retrieval on XML token streams [7, 8, 4]. However the automata model suffers from not being able to strike a balance between the expressive power of the query it can handle and the manageability of its constructs. It either provides limited "recognizer-like" query capabilities, e.g., [4] gives only boolean answers to XPath expressions rather than constructing the results. Or, it may require a huge number of states, actions and transitions, resulting from the low level description of the patches for providing more query capabilities [8, 7].

In contrast, the *algebraic query processing paradigm* has been proven to be practical for query optimization, because of (1) its modularity of composing a query from individual operators, and (2) its support for iterative and thus manageable optimization decisions at several abstraction levels (e.g., logical and physical plans). However, the data model underlying this paradigm assumes sets of self-contained tuples. XML token streams however do not meet this requirement.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

Features of Raindrop. Either paradigm has its own deficiencies. Yet, they complement each other. We therefore propose a novel paradigm for XQuery stream processing, called *Raindrop*, that is the first system to strike a balance between the two paradigms. The novel features of *Raindrop* include:

1. **Uniform Modeling.** *Raindrop* is a layered algebraic framework that uniformly models both the tuple-based and token-based paradigms. This leads to the optimization opportunities not studied in the previous literature [2, 8, 7, 4], namely, the tradeoff between pushing computation into and out of the automata.
2. **Layered Optimization.** The framework supports several abstraction levels of plan refinement, which would not be as easily feasible as . We have developed XML stream-specific optimization techniques for each level.

In this demo, we highlight the schema-based optimization (SQO) on one abstraction level. Schema knowledge is used to rewrite a query into a more efficient one. Most current literature on SQOs in XML focuses on techniques that are either (1) general regardless of persistent or streaming XML sources [1] or (2) specific to persistent XML sources [3]. For example, *query tree minimization* [1] is a general technique. It eliminates a pattern from the query if the pattern is known to always exist. Since the pruned query involves less computation than the original one, it is more efficient to evaluate regardless of the nature of data sources. For another example, the *query rewriting using state extents* [3] technique requires indices on the data. Applications on persistent XML can usually afford the preprocessing of building indices while this is often not the case for XML stream applications due to the on-the-fly arriving nature of their data. Therefore this technique is more suitable for persistent XML. We instead focus on SQOs specific to XML streams. To the best of our knowledge, no previous work has proposed a comprehensive solution for XML stream specific SQO techniques. [4] is the closest to our work. It handles only limited query (i.e., boolean XPath match) with one type of constraint. In contrast, first, we handle more complex query type, i.e., a subset of XQuery; Second, we support most commonly used constraints in XML Schema.

2 Uniform Modeling

To exploit both algebraic and automata paradigms, we choose to extend the algebraic model to accommodate automata for two reasons. First, this allows us to reuse the well-studied algebra-based techniques. Second, algebra provides multiple description levels which allows us to present the details of automata computation at the lower level while presenting the semantics of automata computation at the higher level.

Our *Raindrop* algebraic framework is composed of four levels of abstraction [6, 9]. The highest level, a *semantics-focused plan*, describes the semantics of a query regardless of persistent or stream data sources. We reuse Rainbow [10], an XQuery engine for persistent inputs, for the initial plan construction and general XQuery optimization. Next, the *stream logical plan* extends the first level with new constructs for tokenized stream inputs. The next lower level is the *stream physical plan* describing implementation strategies for each logical operator. The final level, the *stream execution plan*, captures the synchronization and data transfer mechanisms among physical operators (i.e., scheduling). Each level refines the plan at the adjacent higher level with more details.

We here highlight the stream logical plan level since new operators and plan structures are first introduced into this level to model the token-based automata computation as algebraic plans. Also this level is where SQO techniques are applied. Such plans are seamlessly integrated with the tuple-based algebraic plans, providing a uniform view of all computation.

Example 1 FOR $a in Stream("onlinenews")/news
LET $b = $a/source, $c = $a/keyword
WHERE $b = "CNN"
RETURN <CNN>$c</CNN>

Example 1 gives an XQuery which asks for the keywords of online news reported by CNN. Suppose we perform all pattern retrieval in the automata, the previous literature [5, 2] models the whole automata computation as a single operator. This operator (called *x-scan* in [5]) exposes a fixed interface, namely, the bindings to *all* the XPath expressions in the XQuery, to its downstream operators. Such an operator cannot be rewritten in combination with the other non-automata operators due to its coarse granularity and fixed interface. Instead, we propose to model the automata computation as an algebraic plan consisting of operators at finer granularity compared to *x-scan*. The benefit is that this overall algebraic plan can be uniformly understood and optimized even it contains more traditional tuple-based operators and the novel token-based automata operators.

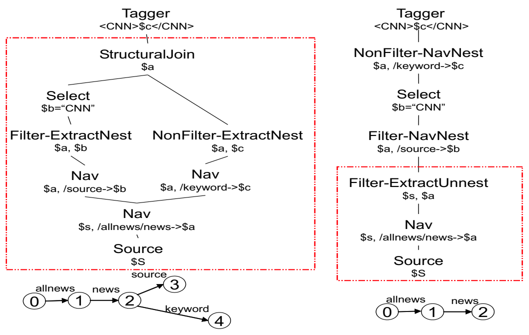

Figure 1: Alternative Stream Logical Plans

The subplan in the dashed box in Figure 1 (a) shows our modeling of the pattern retrieval in Example 1. The intuition behind this modeling is that the retrieval of a tree pattern in the automata can be decomposed into (1) the retrieval of its "linear patterns", and (2) the combination of bindings of individual linear patterns into bindings of tree patterns. For example, see Figure 1 (a). First, $Nav_{\$a,/source \rightarrow \$b}$ locates all the tokens that form the elements accessible via */source* from $a and binds these tokens to variable $*b*. Second, $Filter\text{-}ExtractNest_{\$a,\$b}$ composes the tokens identified by $Nav_{\$a,/source \rightarrow \$b}$ into self-contained elements. The linear pattern $\$a/keyword$ is similarly resolved except that a different composition operator $NonFilter\text{-}ExtractNest_{\$a,\$c}$ is used. The difference between $Filter\text{-}ExtractNest_{\$a,\$b}$ and $NonFilter\text{-}ExtractNest_{\$a,\$c}$ is that: in the former, the absence of $*b* (i.e., the *source* elements) within a *news* element filters out this $*a* (i.e., the *news* element); while in the latter, the absence of $*b* does not filter out $*a*, e.g., $<CNN></CNN>$ will be returned for a *news* element whose source is CNN but does not contain any keywords. Finally, $StructuralJoin_{\$a}$ joins together the composed *source* and *keyword* elements that are within the same *news* element. The automaton in Figure 1 (a), similar to those used in [5, 2], is one implementation for *Nav* operators in the plan.

We now show that different amount of computation can be done in the automata for the same semantics-focused plan. Figure 1 (b) shows another stream logical plan for Example 1. This plan only retrieves *news* elements from the token streams. Compared to Figure 1 (a), it has less computation done in the automata[1]. Suppose only a small proportion of *news* are from CNN, then plan (b) may benefit from the early applying of $Sel_{\$b="CNN"}$, because this saves the time of finding those *keyword* elements whose source is not CNN. Figure 2 shows our experimental results of the execution time of two plans for a query similar to Example 1 but with more path expressions and

[1] *Filter-NavNest* and *NonFilter-NavNest* [10] are non-automata operators navigating into self-contained tree-like elements (e.g., *news* elements composed by $Filter\text{-}ExtractUnnest_{\$s,\$a}$) to find the targets.

selections [9]. For data with different selectivities for the select conditions, different plans are optimal. Note that the previous literature [2, 7, 8, 4] simply embraces a *maximal pushdown* strategy (corresponds to the "5 Navs Pushed Down" strategy in Figure 2) which does not ensure the optimality. This illustrates that our framework opens up more optimization opportunities beyond the existing solutions in the literature.

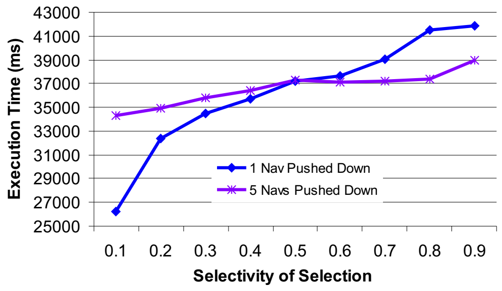

Figure 2: Execution Time of Two Alternative Plans (on a 85M XML Stream)

3 Schema-Based Optimization

At the stream logical plan level, schema knowledge is used to rewrite a query into a more efficient one. We focus on SQO techniques specific to XML streams. The distinguishing feature of pattern retrieval on XML token streams is that it solely relies on the document-order traversal of tokens due to the lack of indices. We observe that the order or occurrence constraints of XML elements can be used to expedite the traversal by avoiding unnecessary pattern retrieval.

Let's consider a query similar to Example 1 but asking for both the *news* and its *keyword* elements. For ease of illustration, we use DTD as the schema description language here (even though we use XML schema in our system). We suppose the element type *news* is described as <*!ELEMENT news* (*source?*, *body*, *related*, *keyword*$^+$)>.

Our approach consists of three phases. In the first phase, i.e., the *initiation phase*, we construct query trees describing the structural pattern to be resolved in the automata (there can be multiple query trees when there are multiple data sources). Figure 3 shows the query tree for Figure 1 (a). The relationship between *news* and *source* (represented as solid line) is distinguished from that between *news* and *keyword* (represented as dashed line). As mentioned in Section 2, the absence of *source* subelement in a *news* element would filter out this *news* while the absence of *keyword* would not. Furthermore, we apply type inference [3] on the query trees so that nondeterministic navigation steps such as "//" or "*" are resolved as much as possible.

The second phase is the *reasoning phase*, namely, to apply a set of schema-based rewriting rules on the

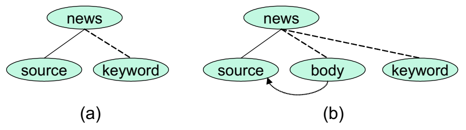

Figure 3: Query Tree for Figure 1 (a)

initialized query trees. An example rewriting rule is *pattern introduction*. For example, a structural pattern /*body* can be introduced to the query tree in Figure 3 (a). Figure 3 (b) shows the rewritten query tree. The arc from *body* to *source* indicates that if *body* has been encountered, the recognition of *source* will then be dropped within the current *news* element. This is because a *source* element, if any, must have appeared before the *body* element. If the recognition of *source* element is dropped without finding any *source* elements, all other pattern retrievals within the current *news*, i.e., composing the *news* element and retrieving *keyword* elements, will also be dropped. This can be a major saving when the *body*, *related* and *keyword* elements are long.

We analyze the relationship among the rewriting rules and derive an application order to ensure the quality of modified query tree. This order ensures two properties, *completeness* and *minimality*. *Completeness* indicates that no beneficial rule application is missed, e.g., the introduction of pattern /*body* will not be missed. *Minimality* means no redundant rule application is introduced. For example, pattern /*related* will not be introduced since the pattern /*body* already serves the purpose for marking the completion of pattern /*source* and bindings of /*body* occur before bindings of /*related* within a binding of *news*.

In the third phase, *plan generation*, we rewrite the original plan to a new plan according to the new query tree. Figure 4 shows the new plan corresponding to the modified query tree in Figure 3 (b). Compared to Figure 1 (b), a new state, i.e., state 5, is added to the automaton. State 5 is associated with the $ExtractNest_{\$a,\$b}$ operator, indicating that if state 5 is activated, this operator is checked. If no outputs within current binding of $\$a$ exist or no outputs satisfy the predicate $\$b = \text{"}CNN''$, the transitions from state 2 will be suspended which results in the automaton in the left bottom part in Figure 4.

4 Demonstration

We will use two applications in the demonstration:

1. *Sports news dissemination.* We use the real data set conforming to *SportsML* DTD (www.sportsml.com), the standard for sharing sports data developed by International Press Telecommunications Council.

2. *On-line auction monitoring.* We generate synthetic data conforming to on-line auction DTDs proposed by XMark (www.xml-benchmark.org), an XML benchmark.

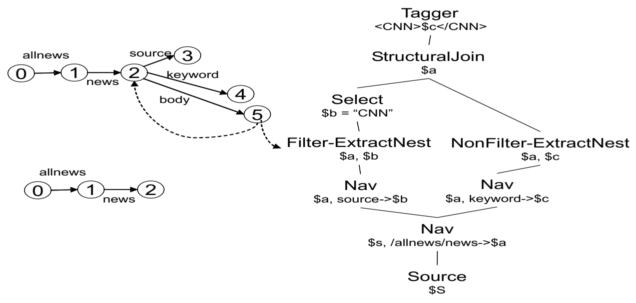

Figure 4: Modified New Plan

Our demonstration is composed of two parts.

Stream Logical Plan.

1. *Generating Alternative Plans.* Given an XQuery, we first show its semantics-focused plan. We generate a set of alternative stream logical plans from the semantics-focused plan, using different computation pushdown strategy. Users can choose a plan from the left panel in Figure 5 and view the plan as well as its automata in the right panel.

2. *Comparing Alternative Plans.* We deploy a stream generator on one machine and send the stream continuously to another machine on which the stream processing engine runs. For comparison of alternative plans, alternative plans run in turn. We provide a "plot all performances" functionality (in the bottom panel) to visually compare their performances. We will change the data characteristics of the stream source, illustrating that under different circumstances, the optimal plan can be different.

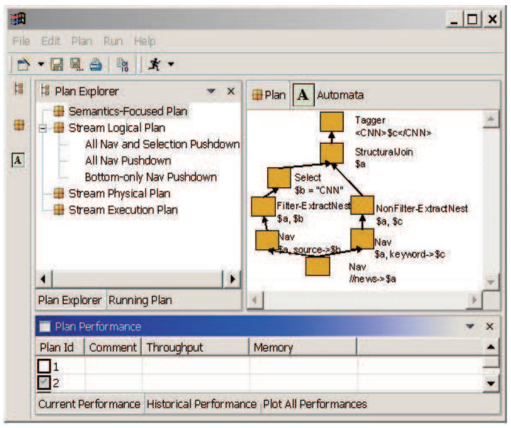

Figure 5: GUI Showing Stream Logical Plans

Schema-Based Optimization.

1. *Showing Static Structures.* We show the required static structures, including (1) query tree; (2) schema (modeled in graphs), and (3) a library of rewriting rules in the upper panel in Figure 6.

2. *Tracing Reasoning Process.* When a rule is applied on a query tree node, the rule in the library, the node and any changes made to the query tree are highlighted in the bottom panel in Figure 6. The history of rule applications can be traced by scrolling the bottom panel.

3. *Comparing Optimized Plans with Original Plans.* We plot the performances comparing an original plan with those optimized ones under various data characteristics of the stream.

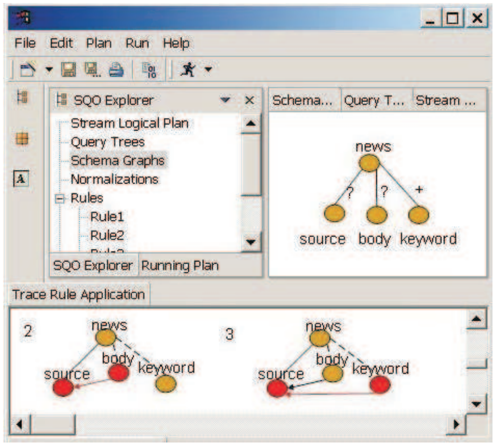

Figure 6: GUI Showing Schema-Based Optimization

References

[1] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD, Santa Barbara, California*, pages 497–508, June 2001.

[2] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proceedings of VLDB*, pages 261–272, 2003.

[3] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE, Orlando, Florida*, pages 14–23, February 1998.

[4] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, pages 419–430, 2003.

[5] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*, 11 (4): 380–402, 2002.

[6] J. Jian, H. Su, and E. Rundensteiner. Automaton Meets Query Algebra: Towards A Unified Model for XQuery Evaluation over XML Data Streams. In *Proceedings of ER*, 2003.

[7] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, pages 227–238, 2002.

[8] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of SIGMOD*, pages 431–442, 2003.

[9] H. Su, J. Jian, and E. A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *Proceedings of CIKM*, 2003.

[10] X. Zhang, K. Dimitrova, L. Wang, M. El-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In *SIGMOD Demo*, page 671, 2003.