Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams

Hong Su*
Worcester Polytechnic Institute
Worcester, MA 01609
suhong@cs.wpi.edu

Jinhui Jian
Worcester Polytechnic Institute
Worcester, MA 01609
jian@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
Worcester, MA 01609
rundenst@cs.wpi.edu

ABSTRACT

XML stream applications bring the challenge of efficiently processing queries on sequentially accessible token-based data. While the automata model is naturally suited for pattern matching on tokenized XML streams, the algebraic model in contrast is a well-established technique for set-oriented processing of self-contained tuples. However, neither automata nor algebraic models are well-equipped to handle both computation paradigms. The goal of the *Raindrop* project is to accommodate these two paradigms within one algebraic framework to take advantage of both. In our query model, both tokenized data and self-contained tuples are supported in a uniform manner. Query plans can be flexibly rewritten using equivalence rules to change what computation is done using tokenized data versus tuples. This paper highlights the four abstraction levels in *Raindrop*, namely, *semantics-focused plan*, *stream logical plan*, *stream physical plan* and *execution plan*. Various optimization techniques are provided at each level. The necessity of such a uniform and layered plan is shown by experimental study.

Categories and Subject Descriptors

H.2.4 [**Database Manager**]: Query Processing

General Terms

Management

Keywords

XML Stream, XQuery Algebra, Query Processing

1. MOTIVATION

There is a growing interest in data stream applications such as monitoring systems for stock, traffic and network [2]. These applications process continuously arriving data streams rather than previously stored data. XML stream applications such as personalized web page delivery and online shopping order handling, because of the nested complex XML data format, pose additional challenges beyond those in the relational stream applications [2].

Since an XML stream can be potentially infinite or may not be complete within a reasonable time due to the network delay, a strategy that incrementally processes the available data is favored over a strategy that only handles the data after it has been completely received. SAX parsers [12] are therefore frequently used since they support incremental XML processing in a token-by-token manner. A token in XML can be a start tag, an end tag or a PCDATA item. Note that such a token-by-token processing is not directly analogous to the tuple-by-tuple pipelining processing typical for relational query engines. Tuples in relational databases are discrete and have a fixed schema. Most importantly, a tuple, coupled with the schema knowledge, has its semantics completely determined by its own values. A token, on the other hand, is not self-contained since it lacks semantics without the *context* provided by other tokens in the stream.

State-of-the-Art: Automata versus Algebra. Current proposals for processing tokenized XML streams can be divided into two categories. The first category, such as XSM [9], offers a new query paradigm different from algebraic-based tuple processing. XSM uses a transducer model for query processing in which there is no explicit concept of tuples. All operations are modeled as Turing machine behaviors, i.e., reading tokens from input buffers and writing tokens on output buffers. However, query optimization techniques for such a model have not been much studied by the database community. On the other hand, the tuple-based algebraic query model has been long studied and widely adopted by the database community. Various XML algebras have also been proposed using a tuple [15, 6] or a tuple-like [11] data model. Approaches in the second category, including YFilter [4] and Tukwila [7], works in a paradigm that could more naturally take advantage of existing set-oriented tuple processing techniques.

The query processing in the second category can be decomposed into two phases. In the first phase, tokens are processed by an extended automata engine for pattern retrieval. Automata are well-suited for this role because they were originally designed for recognizing languages, i.e., matching alphabet sequences against the patterns specified in a grammar. The automaton is extended with auxiliary functionalities so that when the tokens are scanned and recognized, objects are created from the tokens and organized into tu-

*Supported by IBM PhD Corporative Fellowship.

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.

ples. In the second phase, these tuples are input to a more conventional query engine capable of processing tuples.

For example, suppose we have an XML stream (a data set used by an XML benchmark XMark [14]) shown in Figure 1 (a). Each token is annotated with an identifier number in italics font. Figure 1 (b) shows an XQuery on this stream. Figure 2 shows the Tukwila [7] query plan for Figure 1 (b). While the processing in the second-phase engine is expressed as a query tree of operators, the processing in the first-phase engine (i.e., automaton) is modelled as a single operator called *xscan*. This *xscan* operator exposes a fixed interface, namely, the bindings to *all* the XPath expressions in the query, to its downstream operators.

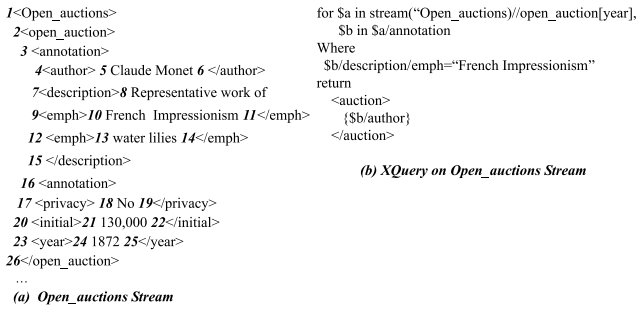

Figure 1: XML Document and XQuery

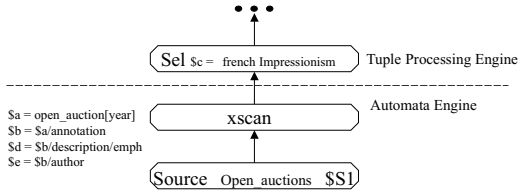

Figure 2: Tukwila Query Plan

Such a modelling has two major drawbacks. First, the fixed interface excludes the possibility of optimizing by pushing computation into or pulling computation out of the *xscan* operator. For example, Tukwila would not allow pushing the selection $Sel_{\$d=\text{"}French\ Impressionism\text{"}}$ into *xscan* and thus this potential optimization is missed. Second, such a bloated *xscan* operator models the complete pattern retrieval phase as one black box. We note that *xscan* itself is composed of a sequence of computations such as finding linear patterns separately and composing linear patterns into tree patterns. However no operators of a finer granularity are provided below this rather complex *xscan*. In short, such a rigid and heavyweight operator cannot be effectively optimized via standard query rewriting techniques. It can neither be rewritten with the other operators in the query plan, nor can be internally rewritten. The automata processing, though accommodated in an algebraic framework as a special operator, actually does not benefit from the opportunities that an algebraic framework is supposed to provide.

Our Approach. We instead propose to model the automata processing as a query subplan composed of operators at a more reasonable granularity. Such a model now offers benefits in several aspects. First, the subplan can be reasoned over in a modular fashion. Optimization techniques can be studied for each operator separately rather than only for a plan as a whole. Second, equivalence rules can be applied for rewriting the query plan. The automata processing therefore is accommodated in the algebraic framework uniformly with the other operators, treating all query semantics as first class citizens.

Our algebraic framework is composed of plans at four levels of abstraction. The highest level is a *semantics-focused plan* which expresses the query semantics regardless of persistent or stream data sources. Next, the *stream logical plan* extends the first level with special constructs for tokenized stream data sources. The next lower level is the *stream physical plan* describing implementation strategies for each operator defined at the stream logical plan level. The final level, the *stream execution plan*, describes the synchronization and data transfer mechanism among physical operators. Each level adds more details to the plan at the adjacent higher level. Mapping alternatives are offered to convert a plan from each level to the next lower level. Such a layered framework enables us to reason at different levels of detail, thus rendering optimization tractable and practical.

Roadmap. We briefly describe the semantics-focused plan in Section 2. Then we focus on the stream logic plan and stream physical plan in Sections 3 and 4 respectively. Due to space limitations, we do not present the lowest level here.

2. SEMANTICS-FOCUSED PLAN

The semantics-focused plan is based on an algebra called the XML Algebra Tree (XAT) [15]. It captures the core features of XQuery. The operators in XAT include (1) XML specific operators, e.g., *Tagger*, *NavUnnest* and *NavNest*; (2) SQL like operators, e.g., *Select* and *Join*. The data model is a collection of tuples called *XAT tuples*. A XAT tuple is composed of cells each of which is bound to a variable.

Figure 3 shows the semantics-focused plan for the query in Figure 1 (b). The intermediate XAT tuples are also shown for some operators. The semantics of the XAT operators used in the example are described in Table 1. For each operator, except the *Source* operator, its semantics are defined in terms of the outputs expected when given an input collection S.

Operator	*Description*
$\text{Source}^{\$col}_{desc}$	Bind data source *desc* to $\$col$
$\text{Tagger}^{\$col}_{p}$	For each tuple $s \in S$, tagger s in pattern p. A new tuple generated by concatenating s with taggered data bound to $\$col$.
$\text{NavUnnest}^{\$col2}_{\$col1,path}$	For $s \in S$, navigate into column $\$col1$. For each node reachable via *path*, a new tuple generated by concatenating s with target bound to $\$col2$.
$\text{NavNest}^{\$col2}_{\$col1,path}$	For $s \in S$, navigate into $\$col1$, aggregate all nodes reachable via *path*. A new tuple generated by concatenating s with aggregated nodes bound to $\$col2$.
Select_{c}	For $s \in S$, filter s by condition c.

Table 1: Semantics of XAT Operators

A cell in an XAT tuple can be one of the following types, called *regular XAT cell formats*: (1) an atomic value, (2) an XML element node modeled as a labeled tree (e.g., binding to $\$a$ in the output of $NavUnnest_{S1,//open\_auction}\$a$) or (3) an unordered or ordered collection[1] of XML element nodes or atomic values (e.g., binding to $\$d$ in the output of $NavNest_{\$b,/description/emph}\$d$).

[1]XQuery supports both unordered and ordered expressions.

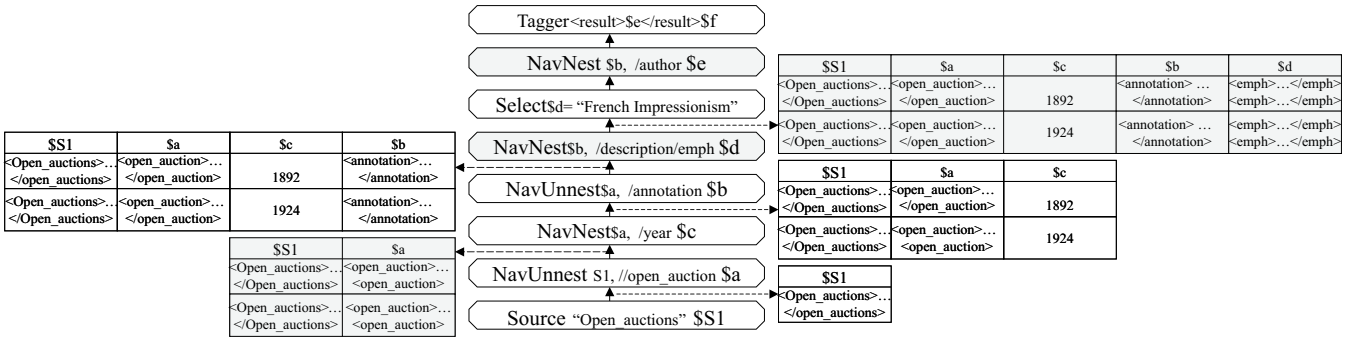

Figure 3: Semantics-Focused Plan (annotated with intermediate results for querying over Figure 1 (a))

Particularly, a variable bound to a path expression in the *For* clause (e.g., $a from *For* $a *in Stream("Open_auctions") //open_auction*) is expressed as an output variable of *NavUnnest* (e.g., $NavUnnest_{S1,//open\_auction}\$a$). *For* clause evaluates the path expression, iterate over the items in the resulting sequence, and bind the variable to each item in turn. In contrast, the *Let*, *Where* or *Return* clause binds a variable (e.g., $d from *Where* $b/description/emph = "French Impressionism"$) to the expression result without iteration. These variables are expressed as output variables of *NavNest* (e.g., $NavNest_{\$b,/description/emph}\$d$). The two highlighted intermediate results in Figure 3 show the difference.

At this top level of the framework, we apply general rewriting optimization [15] such as decorrelation.

3. STREAM LOGICAL PLAN

At the stream logical plan level, besides the *regular XAT cell formats*, we additionally support a token-based data format to accommodate the tokenized input. Meanwhile, new operators and plan structures are introduced to manipulate this new data format.

3.1 Token-Based Data Format

The new data format, *contextualized tokens*, is composed of two parts: value and context. The value models the local characteristics of the token itself while the context models the relationship between this token and other tokens.

Token Value. The token value is represented as a triplet ($ttype$, $tname$, $attrs$). $ttype$ is the token's type which can be *START*, *END*, or *PCDATA*. $tname$ is the tag name for the start and end tag, or the data content for *PCDATA* item. For the start tag token, $attrs$ is a set of attribute name and attribute value pairs. For the other types, $attrs$ is an empty set $\emptyset$. The values of the first 6 tokens in Figure 1 are:

1 (START, "Open_auctions", $\emptyset$) 2 (START, "open_auction", $\emptyset$)
3 (START, "annotation", $\emptyset$) 4 (START, "author", $\emptyset$)
5 (PCDATA, "Claude Monet", $\emptyset$) 6 (END, "author", $\emptyset$)

Definition 1. A token's *associated element* is the element of which this token is a start tag, an end tag or a direct PCDATA content.

Definition 2. A token t is a *component token* of an element e if t's associated element is e's subelement or e itself.

Token Context. We focus on the context regarding the ancestor-descendant relationship between tokens. This relationship is most commonly queried in XPath expressions using such axis specifications as *child*, *descendant*, *parent* and *ancestor-or-self axis* etc. Similar to other work in the literature, we currently do not address the relative position relationships among siblings, such as *preceding-sibling* axis, nor position predicates like *description/emph*[2].

Each token context also carries the concept of an identifier which identifies the token's associated element. An element's start tag, end tag or direct PCDATA token thus have the same context identifiers.

Functionalities Supported by Contextualized Tokens. The contextualized tokens should support three boolean functions as defined below. In the notations below, t and t' represent two tokens, and p represents an XPath expression.

1. $Reachable(t, t', p)$: If t and t' are both start tags, return whether from the element associated with t, the element associated with t' is reachable via p. Otherwise, return false.
2. $Within(t, t')$: If t is a start tag, return whether for the element e associated with t, token t' is a component token of e. Otherwise return false.
3. $t \cong t'$: Indicate whether the context identities are the same. In other words, it indicates whether elements associated with t and t' are the same.

3.2 Operators for Contextualized Tokens

We now introduce new operators that take contextualized token inputs or generate contextualized token outputs, namely, *Source*, *Navigate*, *ExtractUnnest*, *ExtractNest* and *StructuralJoin*. An operator is defined by $Op_{paras}outvar(U_n)$, where Op is the operator's name, $paras$ is the parameters and $outvar$ is the variable bound to newly generated cell in the output tuples. It gives the outputs of Op on U_n, the first n input tuples.

We use the monoid comprehension calculus [5] to describe the semantics. Informally, a monoid comprehension is in the form of $mergeOp\{f(a, b, ...) \mid a \leftarrow A,\ b \leftarrow B,\ ...,\ pred_1,\ pred_2,\ ...\}$ where $mergeOp$ is for merging several collections into one collection. The following computation would be performed:

```
result := an empty collection;
for each a in A, b in B, ...,
    if pred1 ∧ pred2 ∧ ...
      result := result mergeOp f(a, b, ...)
```

For example, $\cup\{(a, b) | a \leftarrow \{1, 2\}, b \leftarrow [4]\}$ joins the set {1, 2} with the list [4]. Since $\cup$, the $mergeOp$ for a set, is used for result collection construction, a set is returned, i.e.,

Notation	Explanation
$u.\$c$	get value of cell $\$c$ from tuple u
$< c_1 = v_1, c_2 = v_2, ... >$	construct a tuple with cell c_1 of value v_1, cell c_2 of value v_2...
$u_1 \circ u_2$	construct a tuple concatenated from tuples u_1 and u_2
$\Pi_{\neg colList}\, u$	construct a tuple by projecting all columns except those in $colList$ from tuple u
$+\!\!+$	merge operator for list (represented as [ ])

Table 2: Notations

{(1, 4), (2, 4)}. Several other notations used for defining the semantics are listed in Table 2.

3.2.1 Source

$$Source_{strName}\$s(T_n) = +\!\!+ \{< \$s = t_0, \$\widetilde{s} = t >| t \leftarrow T_n\}$$

The *Source* operator's inputs are a list of tokens represented as T_n. *Source* binds a stream specified by *strName* to an output variable $\$s$. $\$s$ is bound to the first start tag (represented as t_0) in the stream, which identifies the root element and thus the stream. Moreover, the content of the root element needs to be outputted. Therefore each output tuple also contains an implicit variable $\$\widetilde{s}$ which is bound to the component tokens of $\$s$. In general, we use $\$\widetilde{v}$ to represent an implicit variable accompanying explicitly specified variable $\$v$.

We illustrate the semantics of each operator using the example in Figure 1. Each token in the tuple cells is represented by its identifier as annotated in Figure 1 (a).

Example 1. For $Source_{\text{"}Open\_auctions\text{"}}\$s1$, the first 3 output tuples are,

$\$s1$	$\$\widetilde{s1}$
1	1
1	2
1	3

3.2.2 Token Navigate Operator Nav

We provide a token navigate operator *Nav* for pattern recognition over the tokens. *NavUnnest* and *NavNest* at the stream logical level now specifically refer to the navigation over XML element nodes. Unlike *NavUnnest* and *NavNest* whose outputs contain the target element nodes, *Nav* only pinpoints the target node by its start tag while composition of the target nodes are modeled separately in *ExtractUnnest* and *ExtractNest*.

$$Nav_{\$e,p}\$d(U_n) =$$
$$+\!\!+\{< \$d = u_1.\$\widetilde{e}, \$\widetilde{d} = u_2.\$\widetilde{e} > \circ \Pi_{\neg\{\$\widetilde{e}\}} u_1 \mid u_1 \leftarrow U_n,\ u_2 \leftarrow U_n,\ Reachable(u_1.\$e, u_1.\$\widetilde{e}, p),\ Within(u_1.\$\widetilde{e}, u_2.\$\widetilde{e})\}$$

Each input tuple u to $Nav_{\$e,p}\$d$ contains a cell $\$\widetilde{e}$ which is a component token of the entry variable $\$e$ (i.e., the element to be navigated into). If $Reachable(u.\$e, u.\$\widetilde{e}, p)$ is true, $u.\widetilde{e}$ is then the target element and bound to $\$d$. Each output tuple also contains a component token (bound to $\$\widetilde{d}$) of $\$d$.

Example 2. If $Nav_{\$s1,//open\_auction}\$a$ takes the first 5 output tuples from $Source_{\text{"}Open\_auctions\text{"}}\$s1$ as inputs, its outputs are:

$\$s1$	$\$a$	$\$\widetilde{a}$
1	2	2
1	2	3
1	2	4
1	2	5

3.2.3 Composition Operator ExtractUnnest

Sections *3.2.3* and *3.2.4* introduce two extract operators. They convert the data in a contextualized token format into a regular XAT cell format. In other words, they materialize the on-the-fly tokens. We use the term *Extract* to in general refer to both the *ExtractUnnest* and *ExtractNest* operators.

$ExtractUnnest_{\$e,p}\$d$ only takes outputs of *Nav* as inputs. For each target element found, *ExtractUnnest* composes (represented as $\oplus$) its component tokens into an XML element node. For each such composed node, a tuple is created. In the notation below, $Group_{(colList),aggregateOp(col)}C$ represents (1) grouping the collection C by the columns in *colList* and (2) for each group, producing one tuple consisting of: *i.* values of columns in *colList*, and *ii.* the aggregation (which is still bound to *col*), resulting from applying operator *aggregateOp* on the column *col* of that group.

$$ExtractUnnest_{\$e,p}\$d(U_n) = Group_{(\$e,\$d),\oplus(\widetilde{\$d})} U_n$$

Example 3. If $ExtractUnnest_{\$s1,//open\_auction}\$a$ consumes the first 4 output tuples of $Nav_{\$s1,//open\_auction}\$a$ (shown in Example 2), an output tuple is generated in which the cell $\$\widetilde{a}$ contains a yet-to-complete XML element node. This tuple is only a partial output for the inputs seen so far and would be updated as more inputs are processed. Eventually, $\$\widetilde{a}$ in the tuple would contain a complete XML element node composed from $2 \oplus 3 \oplus ... \oplus 26$ where tokens 2 and 26 are the start and end tags of an *open_auction* element.

$\$s1$	$\$a$	$\$\widetilde{a}$
1	2	$2 \oplus 3 \oplus 4 \oplus 5$

3.2.4 ExtractNest

$ExtractNest_{\$e,p}\$d$ aggregates (represented as $+\!\!+$) all target nodes within the same $\$e$ into a collection and creates one tuple for the whole collection.

$$ExtractNest_{\$e,p}\$d(U_n) = Group_{(\$e),+\!\!+(\$\bar{d})}(Group_{(\$e,\$d),\oplus(\$\bar{d})} U_n)$$

Example 4. Assume $ExtractNest_{\$b,/description/emph}\$d$ consumes the first 6 output tuples from $Nav_{\$b,/description/emph}\$d$, the output is:

$\$s1$	$\$a$	$\$b$	$\$\widetilde{d}$
1	2	3	[$9 \oplus 10 \oplus 11, 12 \oplus 13 \oplus 14$]

3.2.5 Structural Join

A *Nav* operator can only resolve a linear XPath expression, i.e., an expression without node test filters. To recognize a tree expression like $a/b[c]$, a structural join operator is needed to glue the bindings to separate linear expressions. For example, the expressions a/b and a/c are merged on their common bindings to their navigation step a. In the notations below, we use UR_{n1} and UL_{n2} to denote the first $n1$ and $n2$ input tuples from the left and right side respectively. Tuples are joined on the context identities of $\$e$.

$$StructuralJoin_{\$e}(UL_{n1}, UR_{n2}) =$$

$$++ \{< ul \circ ur > | ul \leftarrow UL_{n1}, ur \leftarrow UR_{n2}, ul.\$e \cong ur.\$e\}$$

Example 5. Suppose $ExtractNest_{\$b,/description/emph}\$d$ is structurally joined with $ExtractNest^{\$e}_{\$b,/author}$ on $\$b$ (a start tag of an *annotation* element). Assume the left input is:

$\$s1$	$\$a$	$\$b$	$\$\widetilde{d}$
1	2	3	[9 ⊕ 10 ⊕ 11, 12 ⊕ 13 ⊕ 14]

and the right input is:

$\$s1$	$\$a$	$\$b$	$\$\widetilde{e}$
1	2	3	[4 ⊕ 5 ⊕ 6]

Then the output is:

$\$s1$	$\$a$	$\$b$	$\$\widetilde{d}$	$\$\widetilde{e}$
1	2	3	[9 ⊕ 10 ⊕ 11, 12 ⊕ 13 ⊕ 14]	[4 ⊕ 5 ⊕ 6]

3.3 Stream-Specific Plan Structures

XML streams arrive on the fly. Without materialization, each token can be accessed only once. The token consuming operators must be arranged in a meaningful way to avoid any conflicts of "repetitive token access". We therefore propose a special plan structure, called *order-free-navigation*, to meaningfully organize those token-consuming operators.

In essence, the *order-free-navigation* plan structure models the automata behavior. An automaton, when required to find the pattern A or B, would not impose any order on which pattern to find first. Rather it defines each pattern as a sequence of states and lets the input alphabets drive the transition between these states. Either pattern can be found before the other depending on the order in which they occur in the data. Multiple patterns are resolved during a single scan without the materialization. *Order-free-navigation* is such an automata on-the-fly processing style. It decouples the data dependency between navigate operators in the semantics-focused plan and thus lifts the execution order restrictions.

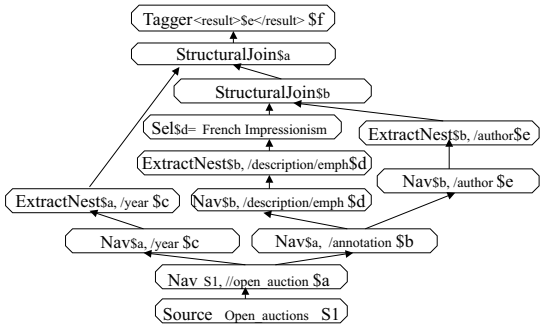

Figure 4: Stream Logical Plan

For example, for the two shaded operators in Figure 3, $NavNest_{\$b,/emph/description}\$d$ must be executed before $NavNest_{\$b,/author}\$e$ can be executed on the same $\$b$. Since $\$b$ needs to be navigated twice, $\$b$ has to be materialized in between. In Figure 4 which depicts the corresponding stream logical plan, we instead model these two operators as being independent. They share the same upstream operator $Nav_{\$a,/annotation}\$b$, which indicates the shared read from $Nav_{\$a,/annotation}\$b$. The downstream $StructuralJoin_{\$b}$ then joins the outputs originating from the two navigate operators, imposing that the two patterns must both be satisfied.

3.4 Stream Logical Plan Rewrite Rules

We now have plan structures that model both automata execution style and non-automata execution style. Rewrite rules are offered to switch operators between these two styles. For example, the *break-linear-navigation* rule rewrites a set of "linear" node navigation operators into independent token navigation operators.

$$NavigateNest^{col2}_{col0,path2}(NavigateNest^{col1}_{col0,path1}) \longleftrightarrow$$
$$ExtractNest^{col2}_{col0,path2}(Nav^{col2}_{col0,path2}) \bowtie_{col0 \cong col0}$$
$$ExtractNest^{col1}_{col0,path1}(Nav^{col1}_{col0,path1})$$

Similar rules are provided for the different combinations of $NavNest$ and $NavUnnest$. The query plan in Figure 4 is rewritten into the plan in Figure 3 by repeatedly applying these rules until no more can be applied.

4. STREAM PHYSICAL PLAN

4.1 Implementation of Contextualized Tokens

A contextualized token is a "raw" token in the stream enhanced with explicit context information. We use a variation of a *pushdown automaton* to implement such tokens.

Data Structures. The automaton is composed of two parts. Figures 5 (a) and (b) shows these two parts for the query plan in Figure 4 respectively. One part is based on a non-deterministic finite machine (NFA). It encodes the path expressions present in the query. Each final state in the automaton marks the end of a path expression. The second part is a stack-like structure representing the contexts of tokens. For each start token, instances of those states that the token would induce the NFA to transit to would be pushed into the stack. If there is no state to transit to, an empty set is pushed instead. For each end token, the instances at the stack top are popped. The stack is restored to the status before the matching start tag had been encountered. For a PCDATA token, since it lacks a paired token to restore the stack, the instances would first be pushed into and then popped out of the stack. For a start or PCDATA token, the stack after the instance push-in is the token's context. For an end tag, the stack before the instance pop-out is its context. Each context is identified by the token that triggers the push-in of its stack top. Therefore the tokens associated with the same element have the same context.

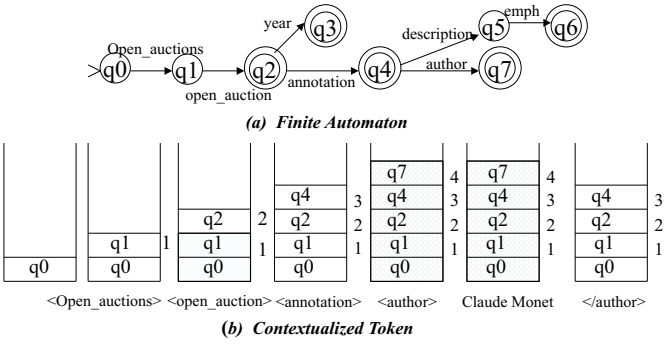

Figure 5: Automaton Implementation of Context

Natural Lifetime of Context. Since the instances of the stack top states can be removed, a start token context's

"natural lifetime" (meaning without any persistence of the removed state instances) is only valid until the context of its paired end tag is computed. For example, the context of token 4 is only accessible in the two stacks highlighted with rippled lines in Figure 5. A contextualized token can be represented in two ways, with a persistent context or a non-persistent context. If the presentation of a non-persistent context is used, the operator consuming such tokens must be executed with discretion to be consistent with the natural lifetime of its inputs (discussed in Section 4.3).

4.2 Alternative Physical Operators

A logical operator may have several alternative physical operators it can map to. Due to space limitation, we here give one example on *Structural Join*.

A structural join, depending on its position in the query plan, may require a certain representation of its contextualized token inputs. Figure 6 shows two alternative subplans. Since $\$b$ is a subelement of $\$a$ ($\$b$ = $\$a/annotation$), the natural lifetime of $\$b$ is shorter than that of $\$a$. Suppose $StructuralJoin_{\$a}$ on inputs from $ExtractNest_{\$a,/year}\$c$ and $Select_{\$d=\text{“}French\ Impressionism\text{”}}$ is very selective and we want to perform it before $StructuralJoin_{\$b}$ as shown in Figure 6 (a). Suppose within each $\$a$, $\$b$ occurs before $\$c$ ($\$c$ = $\$a/year$), then $\$b$'s context expires before $\$a$'s. When a tuple containing $\$c$ is generated by $ExtractNest_{\$a,/year}\$c$, the contexts of $\$b$ in the outputs of $ExtractNest_{\$b,/author}\$e$ would have been lost if a non-persistent context representation were used. Therefore a persistent context representation of $\$b$ must be used here for the later structural join on $\$b$. In contrast, in Figure 6 (b), $StructuralJoin_{\$b}$ is executed first. When the context of $\$b$ expires, the context of $\$a$ is still alive. A non-persistent context representation of both $\$a$ and $\$b$ can be used here. Therefore, we design two implementations of structural join. *InTimeStructuralJoin* joins on a non-persistent context representation while *AnyTimeStructuralJoin* joins on a persistent context representation of the contextualized tokens.

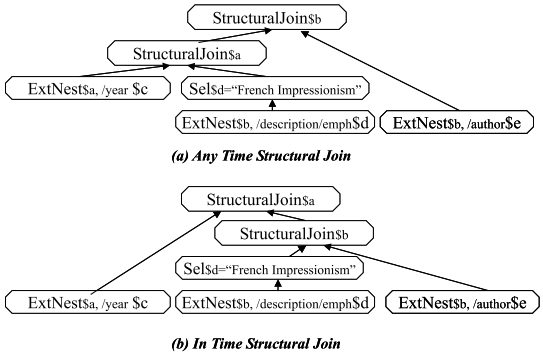

Figure 6: Alternative Structural Joins

4.3 Executing Automata-Inside Operators

Definition 3. An operator, if either (1) it consumes the values of the contextualized tokens, or (2) any cell of the input tuples contains contextualized tokens with non-persistent contexts is said to be **executed inside the automata**.

The automata-inside operators consume data with limited lifetime. In case 1 above, the value of a token expires when the following token is scanned by token navigate operators. In case 2, the non-persistent context of a start tag is alive until the paired end tag is scanned. These operators have to be coupled with the automata in two modes in order to synchronize their computation with the lifetime of their inputs. The *regular-invoke* mode associates a state with token value consuming operators while the *group-by-invoke* mode associates a state with token context consuming operators.

Regular-Invoke. $ExtractUnnest_{\$e,p}\$d$ or $ExtractNest_{\$e,p}\$d$ consumes token values. It will be coupled in a regular-invoke mode with a final state S marking the end of p. When an instance s_0 of S is pushed into the stack by a start tag of a target element (i.e., a binding to $\$d$), the extract operator is invoked and the subsequent component tokens of this $\$d$ would be composed. When an end tag triggers the popping of s_0, the extract operator is revoked indicating the completion of token composition for the current $\$d$.

For example, in Figure 7 (b), when token 9 leads to the push-in of an instance of q_6 (shown as the left stack), $ExtractNest_{\$b,/description/emph}\$d$ is invoked and "French Impressionism" is extracted into the cell bound to $\$d$ (the left operator). Token 12 again would invoke this operator (the right stack) and extract another string value "Water lilies" into the same cell (the right operator) as required by $ExtractNest$'s semantics of aggregating all *emph* subelements within the same *annotation* element. We represent the cell in Figure 7 (b) without underlying line, indicating that the cell is "open", namely, it can accept more string values reachable via "*/description/emph*" from token 3.

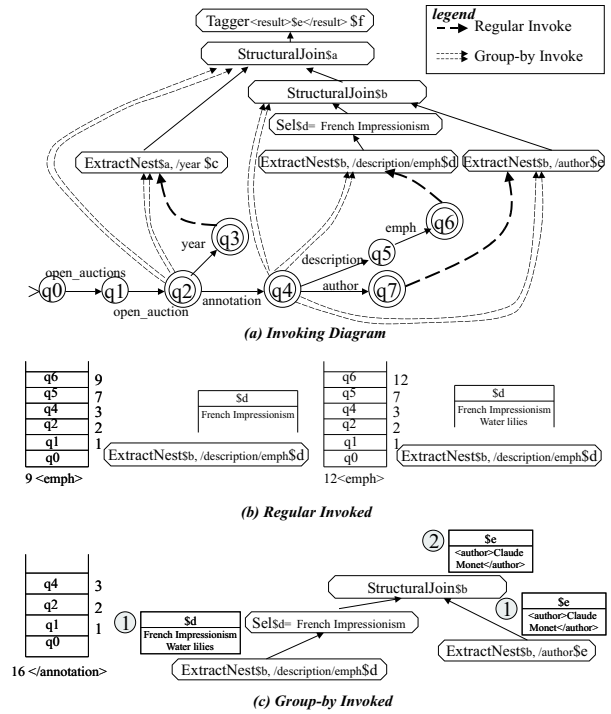

Figure 7: Invoking Automata-Inside Operators

Group-by-Invoke. $ExtractNest_{\$e,p}\$d$ must acquire the knowledge of when a cell is "close", i.e., a group of subelements within a certain $\$e$ has been formed. Because of the sequential traversal of the XML stream, the extracted

target subelements of bindings to $e naturally cluster on $e. An end tag of a binding to $e signals that a complete group of subelements within this $e have been generated. $InTimeStructuralJoin_{\$o}$ also makes use of such *group end signals*. Since all input tuples are clustered on $o, *InTimeStructuralJoin* can simply perform a cartesian product on the inputs when invoked in a group-by mode by end tags of bindings to $o. Correspondingly, all clustered inputs on current $o are purged after the cartesian product, since they should not participate in the next cartesian product on a different binding to $o.

For example, in Figure 7 (c), when an instance of q_4 is removed as token 16 is scanned, $ExtractNest_{\$b,/description/emph}\$d$ is invoked in a group-by mode. The cell collecting the *emph* elements within the *annotation* element with a start tag 3 is now close (represented as a solid line at the bottom of the cell bound to $d), indicating a complete output tuple has been formed. Similarly, $ExtractNest_{\$b,/author}\$e$ closes the cell bound to $e. $InTimeStructuralJoin$ is invoked subsequently (the number "2" beside it indicates it must be fired after the two operators with number "1" have been fired), performing a cartesian product on the input tuples.

5. RELATED WORK

Typical stream applications include networking traffic monitoring, sensor network management and web tracking and personalization. Projects like *Fjord* [10], *Aurora* [1] and *STREAM* [2] address general issues of querying data streams, assuming a tuple-like data model.

Studies also surge in the field of querying XML streams. XSM [9] and XSQ [13] both tackle this problem based on transducer models. XSM decompose an XQuery expression into subexpressions each of which is mapped to a transducer machine. These machines are connected into a network, in which the output buffer of a machine can be the input buffer of another machine. XSQ supports XPath, a subset of XQuery. XSQ supports a more efficient memory management than XSM by promptly cleaning up intermediate buffers when they are no longer needed. Both models present all execution details on the same low level. Such models suffer in both understandability and optimization. First, the large number of constructs in the model, such as states, transitions, actions on buffers, makes it less intuitive to express query semantics. Second, such a large network is difficult to reason about for optimization. In contrast, the success of the relational algebraic model has justified a general approach of layered query plans, i.e., to hide implementation details at different levels of abstraction.

YFilter [4] and Tukwila [7] are closest to our work. As mentioned in Section 1, their approaches suffer from the rigidity and coarse granularity of the modelling of token-based automata computation. Our work instead uniformly integrate the token-based and tuple-based computations and thus offers more query rewrite optimization opportunities. Meanwhile, our physical operators are efficiently implemented by taking advantage of the automata behaviors.

This paper differs from a preceding work [8] in two aspects. First, this paper presents precise semantics for operators handling tokenized stream inputs. Particularly, it proposes a novel concept, *contextualized tokens* for modeling the inputs. Second, multiple mappings from logical to physical operators are provided in this approach while there is no separation of logical and physical levels in the preceding work since a logical operator is tied with only one implementation.

6. EXPERIMENTS

We have implemented a prototype called *Raindrop* with Java 1.4.1. We use ToXGene [3], an XML data generator, to generate the XML documents conforming to the DTD used in the XML benchmark XMark [14]. We run experiments on two Pentium III 800 Mhz machines with 512MB memory. One machine sends XML streams via sockets to another machine which would then process the received data.

Figure 8 shows a semantics-focused plan and various stream physical plans resulted from different amount of computation pushed down into the automata. The query in Figure 8 asks for every element $b whose sibling $c satisfies a given condition. The upper part of Figure 8 (b) shows the subplan for automata-inside operators when only the $NavUnnest_{S1,//open\_auction}\$a$ is pushed into the automata. The bottom part of Figure 8 (b) shows another subplan when all three navigate operators and selections are pushed down into the automata. Our experiments explore the impact of different computation pushdown strategies on the performance. We do not include the time spent on reading the stream from the sockets and parsing the stream in the plan execution time.

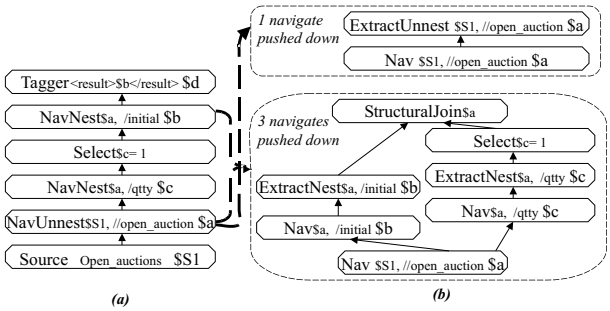

Figure 8: Plans from Different Computation Pushdowns to Automata

Our test queries are similar to Figure 8 (a), but vary in the number of navigate operators. Each query has multiple physical plans resulted from different amount of computation pushed down. For each such physical plan, we test it using various XML documents on which the selection operators have different selectivities. These XML documents are generated in a way so that the selectivities of all selections are the same and independent of each other.

Figures 9 and 10 show the execution time of plans, with 5 and 9 navigate operators respectively, on XML streams of size 85Mb. They reflect the general trend of the execution time in the sequence of queries tested as the number of navigate in the query increases. The x-axis represents the selectivity of a single selection, where the overall selectivity (i.e., the cube of a single selectivity) is shown in brackets.

We can see in both figures, plans in which not all navigate operators are pushed down have consistent rankings. This is because $ExtractUnnest_{\$S1,//open\_auction}\$a$ occurs in all partial pushdown plan. These plans materialize elements in an increasing size since the they have increasing numbers of extract operators which dictates the ranking of the execution time. The amount of materialization is the same

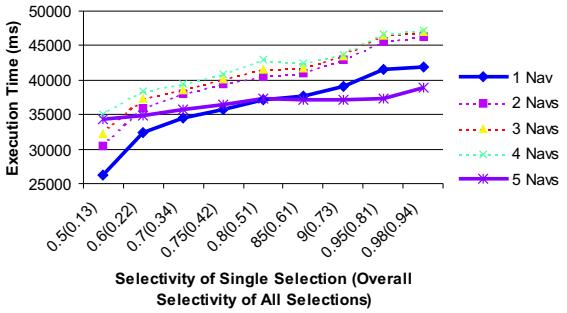

Figure 9: Plan with 5 Navigates and 3 Selections

under various selectivities, since all selections are done after the extraction so that they have no impact on the decisions of what to extract. Therefore the rankings are consistent under various selectivities.

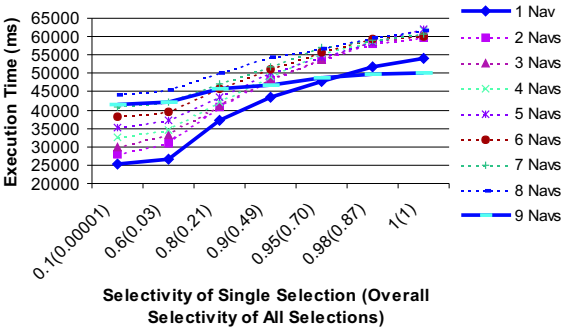

Figure 10: Plan with 9 Navigates and 7 Selections

We also observed that the execution time of the best partial pushdown plan, i.e., the plan with only $ExtractUnnest_{\$S1,//open\_auction}\$a$ in the automata, has a crossover with that of the complete pushdown plan. For the complete pushdown plan, it performs all selections independently and thus spends more time in finding the nodes that would not occur in the final results. However in the complete pushdown plan, $ExtractUnnest_{\$S1,//open\_auction}\$a$ no longer occurs since no $\$a$ is needed outside the automata. Therefore when the overall selectivity is below certain point, the partial pushdown plan benefits from serialized filtering. In contrast, when the selectivity is beyond the point, the benefit of early-stage filtering in the partial pushdown plan is offset by the cost of $ExtractUnnest_{\$S1,//open\_auction}\$a$.

Moreover, the cross over point of the best partial pushdown plan and the complete pushdown plan varies for different queries. It shifts to the higher end as the number of navigate operators in the query increases, e.g., from the overall selectivity of 0.5 to 0.8 in Figures 9 and 10. For our tests ranging from 3 navigates and 12 navigates, the overall selectivity where the cross over occurs range from 0.1 to 0.8.

In summary, we have shown that no single automata push down strategy ensures the best performance for all cases. This further proves the rigid execution strategies the queries are mapped in systems as YFilter and Tukwila may in many cases not be the optimal plan. In contrast, Raindrop's algebraic model provides a framework for finding the optimal plan.

7. CONCLUSION

Raindrop accommodates a token-based automata paradigm and a tuple-based algebraic paradigm within one algebraic framework. This is a novel approach compared to the literature which models the two processing paradigms separately and thus optimizes them separately as well. Our approach instead allows the query optimization over all computations to be performed in an uniform algebraic manner. The proposed four-level algebraic framework enables an iterative process of query plan refinement. We have conducted extensive experiments to explore what computation should be executed in the automata under various circumstances.

8. REFERENCES

[1] D. Abbadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.

[2] B. Babcock, S. Babu, and M. Datar et al. Models and Issues in Data Stream Systems. In *Proceedings of PODS 2002*, pages 1–16, 2002.

[3] D. Barbosa, A. Mendelzon, and J. Keenleyside et al. ToXgene: a Template-Based Data Generator for XML. In *Proceedings of WEBDB*, pages 49–54, 2002.

[4] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proceeding of VLDB*, to appear, 2003.

[5] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of SIGMOD*, pages 47–58, 1995.

[6] H. V.Jagadish, S. Al-Khalifa and A. Chapman et al. TIMBER: A native XML database. In *VLDB Journal*, volume 11(4), 2002.

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

[8] 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*, to appear, 2003.

[9] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, pages 215–226, 2002.

[10] S. Madden and M. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proceedings of ICDE*, pages 555–566, 2002.

[11] P. Mukhopadhyay and Y. Papakonstantinou. Mixing Querying and Navigation in MIX. In *Proceedings of ICDE*, pages 245–254, 2002.

[12] Open Text Corporation. SAX. http://www.saxproject.org, 2002.

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

[14] A. Schmidt, F. Waas, R. Busse, M. Carey, D. Florescu, M. Kersten, and I. Manolescu. Xmark – The XML-Benchmark Project. http://www.xml-benchmark.org, April 2001.

[15] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *WIDM*, pages 15–22, Nov. 2002.