Semantic Query Optimization for XQuery over XML Streams

Hong Su Elke A. Rundensteiner Murali Mani

Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, USA
{suhong|rundenst|mmani@cs.wpi.edu}

Abstract

We study XML stream-specific schema-based optimization. We assume a widely-adopted automata-based execution model for XQuery evaluation. Criteria are established regarding what schema constraints are useful to a particular query. How to apply multiple optimization techniques on an XQuery is then addressed. Finally we present how to correctly and efficiently execute a plan enhanced with our SQO techniques. Our experimentation on both real and synthetic data illustrates that these techniques bring significant performance improvement with little overhead.

1 Introduction

Using schema knowledge to optimize queries, known as semantic query optimization (SQO), has generated promising results in deductive [21], relational [16] and object databases [12]. Naturally, it is also expected to be an optimization direction for XML stream query processing. Among the three major functionalities of an XML query language, namely, pattern retrieval, filtering (e.g., join) and restructuring (e.g., group-by), only pattern retrieval is specific to the XML data model. Therefore, recent work on XML SQO techniques [2, 6, 7, 9, 14] focuses on pattern retrieval optimization. Most of them fall into one of the following two categories:

1. Techniques in the first category are applicable to both persistent and streaming XML. For example, *query tree minimization* [2, 22] would simplify a query asking for "all auctions with an initial price" to one asking for "all auctions", if it is known from the schema that each auction must have an initial price. The pruned query is typically more efficient to evaluate than the original one, regardless of the nature of the data source.

2. Techniques in the second category are only applicable to persistent XML. For example, "query rewriting using state extents" [14] assumes that indices are built on element types. In persistent XML applications, it is practical to preprocess the data to build indices. However, this is not the case for the XML stream scenario since data arrives on the fly and usually no indices are provided in the data.

We instead focus on SQO specific to XML stream processing. The distinguishing feature of pattern retrieval on XML streams is that it solely relies on the token-by-token sequential traversal. There is no way to jump to a certain portion of the stream. We however can use schema constraints to expedite such traversal by skipping computations that do not contribute to the final result, as illustrated in Example 1.

Example 1 *Given a query /news[source] [//keyword contains "ipod"], without schema, whether a news element satisfies the two filters is only known when an end tag of news has been seen. Four computations have to be performed all the time, namely, (1) buffering the news element, (2) retrieving pattern "/source", (3) retrieving pattern "//keyword" and (4) evaluating whether a located keyword contains "ipod". Suppose instead a DTD <!ELEMENT news (title, source?, date, keyword$^+$, ...)> is given. The pattern "/date" can be located even though it is not specified in the query. If a start tag of date is encountered but no source has been located yet, we know the current news will not appear in the final result. We can then skip all remaining computations within the current news element. This can lead to significant performance improvement when the size of the XML fragment from date to the end of news is large (saving the cost of computation (1)) or there are a large number of keyword elements (saving the cost of computations (3) and (4)).*

Only a limited number of XML stream processing

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

engines [3, 4, 7, 9, 13] have looked at the SQO opportunity. Among them, SQO in [7, 13] is not stream-specific (further discussed in Section 2) while SQO in [3, 9] is stream-specific but has the below drawbacks.

Limited Support for Queries. First, [3, 9] address queries with limited expressive power, i.e., boolean XPath matching that only returns boolean values indicating whether an XPath is matched by the XML stream. In other words, it does not differentiate */news[source]* from */news/source*. A more powerful language, like XPath or XQuery, raises new challenges in SQO as listed below.

1. *How to decide whether a schema constraint is useful.* We first use XPath as an example. Given a query *news/source*, knowing that "*source* must occur before *date*" is not helpful. Early detection of the absence of *source* will not lead to any cost savings in buffering, since nothing besides the *source* needs to be buffered (this constraint however would be useful to the query *news*[*source*]). The above constraint will not help the query *news*[*source*]/*title* either, because *title* has already been retrieved when the absence of *source* is detected as <*date*> is encountered. When it comes to XQuery, more subtleties, such as variable bindings and nested queries, have to be considered.

2. *How to execute the optimized query.* XML stream-specific SQO may take place at a lower level than the other SQO. Typically, SQO techniques rewrite a query into a more efficient format at the syntactic level (e.g., with less predicates[16], less patterns [2] or smaller extents [14]). However, no XQuery can capture the optimization in Example 1 at the syntactic level. Specific physical implementations must be devised for these optimization techniques. With more powerful queries supported, the physical implementations become more complex. For example, for an XQuery that buffers data, temporary data must be cleaned carefully when computations are skipped. In Example 1, when *source* is found not to appear, the partially stored *news* must be cleaned. Or for an XQuery that has nested subqueries, a failed pattern in the inner query should not affect the computations in the outer query (discussed more in Section 3.1).

Lack of Strategies for Applying Possibly Overlapping Optimization Techniques. [3, 9] both consider a single optimization technique using one type of schema constraint. Their proposed technique can be independently applied on different parts of the query. If more types of constraints are explored, multiple techniques must be considered. We have observed that when applying these different techniques or even one complex technique on different parts of the query, they may "overlap", i.e., unnecessarily optimizing the same part of the query which causes additional overhead. Strategies are needed to avoid such redundant optimization.

How to support SQO techniques in XQuery and overcome the above drawbacks is the subject of this work. We propose an optimization process consisting of the following steps. First, we use query trees to capture the structural pattern retrieval in the given XQuery. Second, type inference is applied on the query trees. The nondeterministic "*" or "//" navigation steps are replaced with deterministic ones so that more SQO can be applied on the previously schema-less patterns. Third, SQO rules are applied on the query trees. Finally, the query tree is translated back into a query plan executable in our XQuery processing engine. Our contributions include:

1. We utilize type inference to aid with the stream-specific SQO. We handle the complexities caused by type inference in SQO, namely, unions (e.g., $\$a/(b|c)$ resolved from $\$a/*$) and recursions (e.g., $\$a/b^+$ resolved from $\$a//b$ when b is recursive).

2. We assume a widely-adopted automata execution model for XML stream pattern retrieval. Based on the analysis of this model, we derive the criteria regarding what constraints are useful for a given query.

3. We design a set of optimization rules that utilizes the constraints satisfying the "usefulness" criteria. We derive a rule application order that ensures: no beneficial optimization is missed (completeness); and no redundant optimization is introduced (minimality).

4. We incorporate these SQO techniques into an algebraic framework for XML stream processing. We propose strategies for correctly and efficiently evaluating the query plans optimized with SQO.

2 Related Work

SQO for persistent XML may have some resemblance to the stream-specific SQO. XQRL [6] stores the XML data as a sequence of tokens. To find children of a certain type within a context element, the scan on tokens can stop early if the schema tells that no more children are relevant once a child of a particular type is found.

Since the token sequence can be repeatedly accessed, XQRL retrieves the patterns one by one. The earlier one pattern retrieval stops, the smaller the overall cost is. However, in the stream context, as shown in Section 1, not all early detections of failed patterns lead to cost savings. It requires more discretion to decide whether such detections are worthwhile. Moreover, in XQRL, when a pattern is found to fail, the retrieval can simply terminate and another pattern retrieval can start. In the stream context, this process is more complicated. In Example 1, when a *source* is found not to exist, we cannot simply jump to the next *auction* to skip the remaining computations in the current *auction*. We have to suspend the computations, clean up the intermediate results and resume as appropriate.

YFilter [7] and XSM [13] discuss SQO in the XML stream context. They use schema knowledge to decide whether results of a pattern are recursion-free and what types of child elements can be encountered re-

```
CoreExpr ::= ForClause WhereClause? ReturnClause
           | PathExpr
PathExpr ::= PathExpr "/"|"//" TagName|"*"
           | varName
           | streamName
ForClause ::= "for" "$"varName "in" PathExpr
           ("," "$"varName "in" PathExpr)*
WhereClause :: = "where" BooleanExpr
BooleanExpr ::= PathExpr CompareExpr Constant
           | BooleanExpr and BooleanExpr
           | PathExpr
CompareExpr ::= " >"|"! ="|" <"|" <="|" >"|" >="
ReturnClause = "return" CoreExpr
           |<tagName>CoreExpr ("," CoreExpr)* </tagName>
```

Figure 1: Grammar of Supported XQuery Subset

spectively. These in essence type inference techniques belong to general XML SQO.

The goal of FluXQuery [4] is to minimize the buffer size while ours is to reduce unnecessary computations. These two goals sometimes come hand-in-hand: when we reduce the buffering computation (like we do with computation (1) in Example 1), we naturally reduce the buffer size. But in many other cases, our techniques are complementary. Let us consider a query "for $\$a$ in $/news[source]$ return $<news>$ $\{\$a/source$, $\$a//keyword\}$ $</news>$". If given the constraint that *source* must occur before *keyword*, Flux will immediately output any located *keyword* elements, instead of buffering them until the end of the *news* to ensure they are output after any *source*. However Flux is unable to detect the non-existence of *source* and skip the retrieval of $\$a//keyword$ as our techniques do. A combination of their work and ours can boost the performance of both systems.

Finally, there is another class of XML stream query optimization which assumes indices are interleaved with XML streams [8]. The stream index SIX [8] gives the positions of the beginning and end of each element. If an element is found to be irrelevant, the processor can move to its end without parsing anything in the middle. How to combine such indices that arrive at run-time and the schema constraints available at compile-time is an interesting direction to explore in the future.

3 Type Inference on Query Trees

We first propose a query tree representation to capture the pattern retrieval in an XQuery. We then describe how to apply existing type inference techniques [14, 20] on the query trees when an XML Schema is given.

3.1 Query Tree

We support a subset of XQuery as shown in Figure 1. Basically, we allow "for... where... return..." expressions (referred to as FWR) where the "return" clause can further contain FWR expressions; and conjunctive predicates each of which is a comparison between a variable and a constant.

We propose *query trees* to represent the structural patterns in an XQuery. Figure 2 (b) shows such a tree for the XQuery in Figure 2 (a). Each navigation step in an XPath is mapped to a tree node. The descendant axis is also expressed as a tree node labeled "//". The blank node models the relationship between the inner FWR and the outer FWR. We say the node mapped from the first (resp. last) step on an XPath is the *context* (resp. *destination*) node of any node mapped from the same XPath. For example, in Figure 2 (b), the *auction* node represents $\$a$ and is the context node of *seller*. The *seller* node again represents $\$b$ and is the context node of * and *phone*. We also say *auction* is an ancestor context node of * and *phone*.

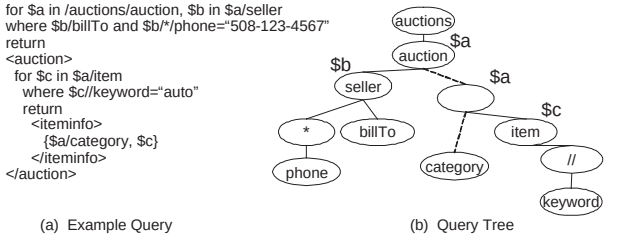


Figure 2: XQuery and Query Tree

There are two kinds of patterns in an XQuery. XPaths in "for" clauses describe required patterns, e.g., in Figure 2 (a), both $\$a$ and $\$b$ in the outer "for" clause must not evaluate to empty for the FWR expression to return any result. In contrast, XPaths in "return" clauses describe optional patterns, e.g., even if $\$a/category$ evaluates to empty, an *iteminfo* element will still be constructed. In the query tree, a solid (resp. dashed) line indicates the child is required (resp. optional) in its parent. For example, a dashed line connects the blank node with its parent, indicating $\$a/category$, $\$a/item$ and $\$c//keyword$ appear in the "return" clause of the outer FWR. A solid line connects the *item* and the blank node, indicating $\$a/item$ appears in the "for" clause in the inner FWR.

3.2 Type Inference

We assume that an XML schema is given for each stream source. An XML schema is modeled as a directed graph with ordered edges. A node in the *schema graph* represents an element type, a sequence group (labeled with "SEQ"), or a choice group (labeled with "CHO"). Each edge from node u to node v is labeled by ($minOccur$, $maxOccur$), indicating the minimal and maximal occurrence of v within u. The default edge label is (1, 1). Figures 3 (a) and (b) show the schema (for compactness, we use an equivalent DTD) and its graph representation.

Figure 4 shows the query tree from Figure 2 (b) after type inference [14, 20]. Each query tree node is now associated with a set of type nodes. Each type node identifies one possible deterministic navigation step that the query tree node represents. Type nodes are connected to capture the sequential relationship among navigation steps. The blank node shares the

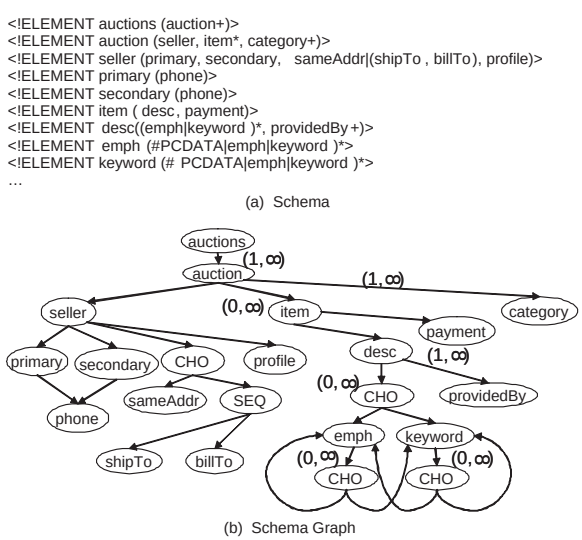


Figure 3: XML Schema and Schema Graph

type nodes with its parent. In the rest of this paper, we refer to a type node by the name of the type. To differentiate between the two type nodes that both represent *keyword* type in Figure 4, we refer to them as $keyword_1$ and $keyword_2$ respectively.

A "*" is resolved to a union of types. In Figure 4, "*" is associated with type nodes *primary* and *secondary*, indicating $\$b/*/phone = \$b/(primary|secondary)/phone$. A "//" node is resolved to a union of sequences of types, e.g., $\$c//keyword$ is resolved to $\$c/desc/(\emptyset|(emph^+/keyword^*)^+|(keyword^+/emph^*)^+)/keyword$, where p^* (resp. p^+) indicates repeating a path p zero or more times (resp. one or more times); $\emptyset$ represents an empty navigation step. The nondeterministic number of navigation steps in the expression (i.e., p^* or p^+) results from the recursive *keyword* or *emph* elements (refer to Figure 3). The "//" node in Figure 2 (b) is now expanded to a *desc* node and a "//" node in Figure 4.

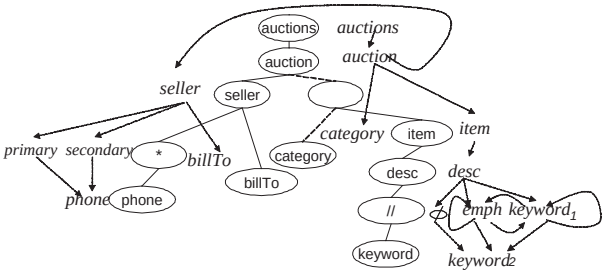


Figure 4: Query Tree after Type Inference

4 Guidelines for Stream XML SQO

We have to understand the processing style of pattern retrieval, in particular what contributes to its costs, to ensure the SQO techniques designed indeed improve the performance. Therefore, we first review a widely-adopted automata processing model and then generalize the guidelines for designing SQO techniques.

4.1 Automata-based Implementation

Automata are widely used [7, 8, 9, 11, 13, 15] for pattern retrieval over XML token streams. We describe one basic automata model [7, 11] that is general and serves as the core of most other automata [8, 9, 15]. The pattern retrieval in the automaton consists of three tasks as below.

Locating Tokens. Figure 5 shows the automaton for retrieving the patterns in Figure 4. Each tree node is mapped to transition edge(s) among states. The λ transition between states 2 and 3 is mapped from the blank node. This λ transition is necessary for executing the optimized plan as we will show in Section 6.

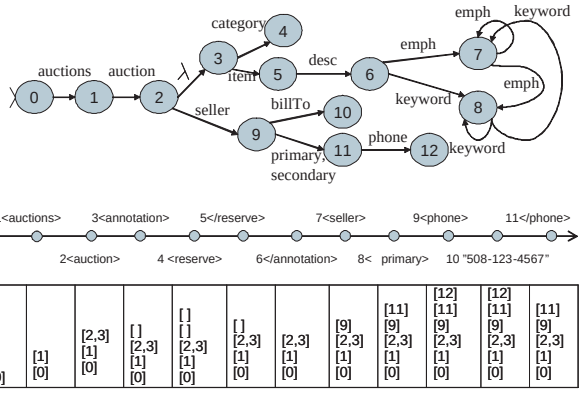


Figure 5: Automata Implementation

A stack is used to store the history of state transitions. Figure 5 shows the snapshot of the stack after each token is processed. An incoming start tag is looked up in the transition entries of every state at the stack top. The states that are transitioned to are activated and pushed onto the stack. For example, when $<auction>$ is encountered, q_1 is transitioned to from q_0 and pushed onto the stack. If no states are transitioned to, an empty set is pushed onto the stack, e.g., when $<annotation>$ is processed. When an end tag is encountered, the states at the stack top are popped out. The stack is therefore restored to the status before the matching start tag had been processed. For a PCDATA token, no change is made to the stack.

Buffering Tokens. Tokens are buffered if they need to be either further filtered or returned by the query. A state can be associated with an extraction operator. For example, in Figure 5, state 4 is associated with an extraction operator. Once state 4 is activated, the extraction operator raises a flag. As long as the flag is raised, the incoming tokens will be buffered. When a state 4 is popped out of the stack by a $</category>$, its extraction operator revokes the flag to terminate the buffering of the *category* element.

Manipulating Buffered Data. The buffered data are consumed by the data manipulation operators that perform selections or structural joins. More details are discussed in Section 6.1.

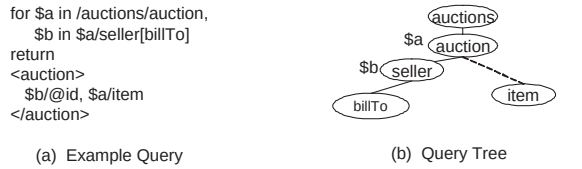
4.2 Design Guidelines for XML Stream SQO

There are two major optimization opportunities. First, we should avoid transitions whenever possible. This obviously reduces the cost of locating tokens. It may also reduce the cost of buffering tokens when those transitions, if not avoided, could otherwise activate states associated with extraction operators. It may even save manipulation cost on the buffered data.

The second opportunity is that an extraction operator should be prompted to revoke the buffering flag once the data it is extracting is known to be irrelevant to the final results. This saves buffering cost.

We now describe how to take advantage of the two opportunities. A pattern $\$v/p$ may "fail" if its p may not occur within $\$v$, or it is involved in a selection, or its required descendant patterns may fail. The failure of a required $\$v/p$ filters out $\$v$. If within a $\$v$, no result of XPath p can occur after any result of XPath p', we say a result of p' is an **ending mark** of p. When an ending mark of p is encountered, we can test whether p fails. This test is an **early filtering** because without the ending mark, we could have only concluded whether p fails when the end tag of $\$v$ is encountered. If p fails, any transitions or active buffering flags can be avoided or deactivated within this $\$v$.

In some cases, even if early filtering of p does not save within $\$v$, it may save within the ancestor context variables of $\$v$. For example, in Figure 6, early detection of the absence of *billTo* within a *seller* would not save any computation within this *seller*. However, since an *auction* has only one *seller*, the filtering out of this *seller* leads to the filtering out of its parent *auction* element. The schema in Figure 3 indicates *item* occurs after *seller* within an *auction*. The locating and buffering $\$a/item$ is saved. Figure 7 summarizes the guidelines of designing XML SQO.



(a) Example Query (b) Query Tree

Figure 6: Filtering Propagation

5 Stream-Specific XML SQO

We now introduce three SQO rules (each utilizing a different type of constraint). Note that our rule set is open-ended. New rules utilizing new constraints could be similarly developed following the guidelines and added into the rule set.

An SQO technique should find ending marks for a pattern $\$v/p$ that satisfies the following criteria:

1. *early filtering is possible*.
   (a) p is a required pattern in $\$v$.
   (b) p may possibly fail in a binding of $\$v$.
2. *early filtering is beneficial*: after the ending marks within a binding of $\$v$ or $\$u$ ($\$u$ is an ancestor context variable of $\$v$), there exist raised buffering flags or states that may be activated.

Figure 7: SQO Design Guidelines

5.1 SQO Rules

Each rule is defined with respect to a patten $\$v/p$. A rule has a pre-condition, a rule body and a post-condition. The precondition ensures that p satisfies criterion 1 in Figure 7. When the precondition holds, the rule body is fired to find the ending marks of p. The post-condition keeps only those ending marks that satisfy criterion 2. The pre-condition and post-condition checking is similar across the rules. We here only describe their different parts, the rule bodies.

Occurrence Rule.

This rule utilizes occurrence constraints. We use $maxOccur(t_1, t_2)$ to represent the maximal occurrence of type node t_1 within type node t_2. For each type t of $\$v$, we derive the maximal cardinality of the results of p within a binding of $\$v$ of type t. If the maximal cardinality is a bounded integer i, then the end tag of the i^{th} result of p is an ending mark in $\$v$ of type t.

Example 2 *In Figure 4, maxOccur(phone, seller) = 2. The end tag of the 2^{nd} phone is an ending mark of $/*/phone$ within a seller.*

Exclusive Rule.

This rule utilizes the the "CHO" node in the schema graph. For each type t of $\$v$, we find whether there is a path p' that never coexists with p within a binding of $\$v$ of type t. If yes, the start tag of the result of p' is the ending mark of p in $\$v$ of type t. This rule may introduce new nodes for p' into the query tree when p' is not specified in the query.

Example 3 *From Figure 3 we know /sameAddr is exclusive to /billTo in a seller element. <sameAddr> is the ending mark of /billTo within a seller.*

Order Rule.

This rule utilizes the order constraints. For each type t of $\$v$, we find whether there exists a path p' that must occur after p within a binding of $\$v$ of type t. If yes, the start tag of the first result of p' is an ending mark of p in $\$v$ of type t. Similar to Exclusive Rule, this rule may also introduce new nodes into the query tree.

Example 4 *In Figure 4, keyword either occurs as a child element of desc, or occurs within a child element emph or keyword of desc. Within desc, providedBy occurs after both emph and keyword. Also, $maxOccur(desc, item) = 1$. Therefore the start tag of the first result of /desc/providedBy within $\$c$ is an ending mark for //keyword.*

5.2 Desired Properties of Rule Application

We now consider the order of applying the rules on the patterns, i.e., on the destination nodes in the query tree (each destination node identifies a pattern). The application order should ensure two properties: *completeness* and *minimality*. *Completeness* means that no beneficial ending mark is missed while *minimality* means no redundant ending mark is introduced.

5.2.1 Completeness

We now define the *independence* of two rules, which is an important property for ensuring the completeness of our rule application algorithm.

Definition 1 *We use $dest(\mathcal{Q})$ to denote the destination nodes in a query tree $\mathcal{Q}$. We denote a new query tree after the application of rule r on a destination node n in $\mathcal{Q}$ as $apply(r, \mathcal{Q}, n)$. $dest(\mathcal{Q})$ - $dest(\mathcal{Q}')$ denotes the destination nodes in query tree $\mathcal{Q}$ but not in $\mathcal{Q}'$. $em(\mathcal{Q})$ denotes the set of ending marks already found for the patterns in $\mathcal{Q}$. Rules r_1 and r_2 are* ***independent*** *of each other if:*

$$em(apply(r_2, apply(r_1, \mathcal{Q}, n), n')) = em(apply(r_1, apply(r_2, \mathcal{Q}, n'), n)), \quad \forall n, n' \in dest(\mathcal{Q}) \quad (1)$$

$$em(apply(r_2, apply(r_1, \mathcal{Q}, n), n')) = em(apply(r_1, \mathcal{Q}, n)), \quad \forall n \in \mathcal{Q}, n' \in dest(apply(r_1, \mathcal{Q}, n)) - dest(\mathcal{Q}) \quad (2)$$

$$em(apply(r_1, apply(r_2, \mathcal{Q}, n), n')) = em(apply(r_2, \mathcal{Q}, n)), \quad \forall n \in \mathcal{Q}, n' \in dest(apply(r_2, \mathcal{Q}, n)) - dest(\mathcal{Q}) \quad (3)$$

Equation (1) says r_1 and r_2 can be applied on the destination nodes in any order and still find the same set of ending marks. Equations (2) and (3) (they are symmetric) say that if the application of one rule introduces new destination nodes into the query tree, the application of the other rule on these new nodes would not result in new ending marks.

Lemma 1 *If rules in a rule set are all independent of each other, then as long as each SQO rule is applied on each destination node in the query tree once, this application process ensures completeness.*

Lemma 2 *All possible pairs of rules r_1-r_2 in our current rule set are independent of each other.*

We briefly explain Lemma 2. First, when a rule in Section 5.1 is applied on a node, it is not affected by the ending marks previously found. Equation (1) in Definition 1 holds. Second, any newly introduced node represents an XPath that is not specified in the query. Such a path is optional and not qualified to have ending marks. Equations (2) and (3) in Definition 1 also hold. Lemmas 1 and 2 will be combined later to show our rule application algorithm achieves completeness.

5.2.2 Minimality

A plain node-by-node rule-by-rule application, though ensuring completeness (Lemma 1), may not ensure *minimality*. It may introduce redundant ending marks.

Example 5 ***(Rules Applied on Same Node)*** *Exclusive and Order Rules, if applied on node billTo in Figure 4, introduce /sameAddr and /profile respectively. However the latter ending mark is redundant: if billTo does not appear, its absence will be caught by ending mark /sameAddr first; if billTo does appear, ending mark /profile then leads to unnecessary checking. In either case, /profile does not help.*

Example 6 ***(Rules Applied on Ancestor and Descendant Nodes)*** *Suppose the schema for auction in Figure 3 is changed to <!ELEMENT auction (..., item, ...)>. The Order Rule on node keyword finds an ending mark: /desc/providedBy (see Example 4) in an item. Also, Order Rule on node item finds an ending mark /category in an auction since item must occur before category. The latter ending mark is meant to detect whether any $\$c$ (item) that satisfies $\$c$//keyword = "Auto" exists in a $\$a$ (auction). This is equivalent to detecting whether the only $\$c$ in $\$a$ satisfies the predicate (a $\$a$ has exactly one $\$c$). However this will always be first detected by ending mark /desc/providedBy in a $\$a$. Therefore the ending mark /category is redundant.*

An ending mark of $\$v/p$ is said to be *surely-working* if it is able to catch all failure of $/p$ in a binding of $\$v$. Not all ending marks are surely-working. For example, if the DTD in Figure 3 is instead <!ELEMENT *item* (*desc*?, *payment*)>, */desc/providedBy* does not necessarily occur in an *item*. The failure of *//keyword* in $\$c$ thus is not ensured to be caught by this ending mark. Based on this concept, we have Observations 1 and 2 which generalize the cases illustrated in Examples 5 and 6 respectively.

Observation 1 *For a $\$v/p$, any ending marks after a surely-working one are redundant.*

Observation 2 *Any ending marks of $\$v/p$ are redundant if (1) within $\$v'$ where $\$v' = \$v/p$, any pattern $\$v'/p'$ satisfying Criterion 1 (a) and (b) in Figure 7 has a surely-working ending mark, and (2) $\$v'$ occurs within $\$v$ exactly once.*

5.3 Rule Application Algorithm

The rule application algorithm has two main components: the *traverser* and the *rule applier*. The *traverser* traverses the query tree and directs *rule applier* to operate on every destination node. From Lemmas

1 and 2, we know the algorithm achieves *completeness*. The *rule applier* outputs a set of event-condition-action constructs in the form of (an ending mark, a pattern, a type node of an ancestor context node). When an ending mark is encountered (event happens), if the pattern fails (condition holds), all computations within the ancestor context node will be suspended (actions are taken). The *rule applier* follows Observations 1 and 2 and thus achieves *minimality*.

```
Algorithm: traverser(tn, atn)
-Input: tn - a type node of a context node $v
       atn - a type node of $v's farthest ancestor context node
         that has maxOccur(tn, atn) = 1
-Output: a set of event-condition-actions
01 Set ecas;
02 for each destination node $v' of $v
03    ecas = ecas ∪ applyRule($v', tn, atn);
04    for each type node tn' of $v'
05          if maxOccur(tn', tn)=1 and
            $v' has only one type node that is a descendant of tn
06              ecas = ecas ∪ traverser(tn', atn);
07          else
08              ecas = ecas ∪ traverser(tn', tn).
09 return ecas.
```

Figure 8: Traverser

The traverser algorithm in Figure 8 takes two inputs. The first input is a type node of a context node $\$v$. The traverser picks qualifying destination nodes of $\$v$ for the rule applier. The second input is a type node of an ancestor context node. This type node will appear as the action part of the event-condition-action output of the rule applier.

Initially, the traverser is called with *tn* and *atn* both set to the only type node of the query tree root (the root must have only one type node that identifies the type of the root element in the stream). Starting from the root, the rule applier operates on each destination node $\$v'$ (lines 2-3). Next, the subtree rooted at $\$v'$ is recursively traversed (lines 4-8). The filtering out of a binding of $\$v'$ leads to the filtering out of the binding of an ancestor context variable $\$v$ (see Figure 6), if the binding of $\$v'$ is the only one occurring in the binding of $\$v$ (line 5). We now walk through an example to show how this works, especially when a context node has multiple type nodes.

Example 7 *Figures 9 (a) and (b) show a query and a schema. The traverser starts from the root node in Figure 9 (c) and finds its destination node* $\$v$. *The rule applier operates on* $/a/*$, *namely,* $/a/(c|d)$ *according to the type inference. An ending mark* $/a/e$ *is found. Next, the traverser navigates into the subtree rooted at* $\$v$ *which has two type nodes* c *and* d. *With respect to* $\$v$ *of type* c *(resp. of type* d*), an ending mark, i.e., the second occurrence of* $/b$ *(resp. the first occurrence of* $/b$*), is found for* $\$v/b$. *Filtering of any binding of* $\$v$ *will not be propagated up to the root. This is because even a binding of* $\$v$ *of type* c *does not contain element* b *that satisfies* $text() =$ "001", *another binding of* $\$v$ *of type* d *may still contain such* b.

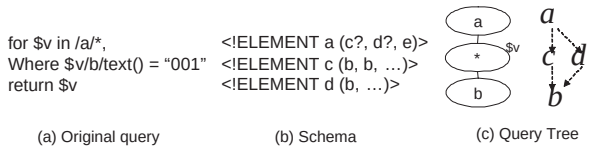


Figure 9: Traverser on Context Node with Multiple Types

```
Algorithm: applyRule(dest, tn, atn)
Input: dest - a destination node;
       tn - a type node of the context node of dest;
       atn - a type node of an ancestor node of dest
Output: a set of event-condition-actions
01 Set ecas;
02 T = type nodes of dest that are descendants of tn
03 find t' where
     (i) t'∈T and t' occurs after all other types in T
     (ii) maxOccur(t', tn) = 1
04 if t' exists {
05    for each destination node dest' of dest
06          applyRule(dest', t', atn);
07    if every dest' has a surely-working ending mark
08          return an empty set;
09 }
10 ecas = ecas ∪ localApplyRule(dest, tn, atn).
11 return ecas.
```

Figure 10: Rule Applier

In Figure 10, *applyRule* algorithm operates on a destination node with respect to its context node of type *tn*. Following Observation 2, it first checks whether ending marks for the pattern identified by *dest* will always be redundant (lines 2 - 9). If not, *localApplyRule* algorithm is applied on *dest*. *localApplyRule* follows Observation 1, that is, if a surely-working ending mark is found, we terminate the rule application. Due to the space limitations, we skip *localApplyRule* algorithm here. Interested readers are referred to [10].

6 Execution of Optimized Queries

We have incorporated the proposed SQO techniques into *Raindrop* [19, 18], an XQuery stream processing engine. We describe (1) how to encode the event-condition-actions derived in Section 5 in the query plans and (2) how to execute such query plans. The described techniques for optimized execution are general to any system that wants to apply the stream-specific XML SQO in Section 5.

6.1 Raindrop Overview

Raindrop represents an XQuery as an algebraic plan. The algebra consists of XML specific operators and SQL like operators such as *Select*. The input and output of the operators are a collection of tuples. A cell in a tuple can contain a token, a single XML node or a collection of XML nodes. Table 1 gives the semantics of the XAT operators that will be used later while the full set of XAT operators can be found in [23].

The top part in Figure 11 shows the plan for the XQuery in Figure 2 (a). For ease of illustration, each operator is annotated with an identifier. For example, the inner FWR expression in Figure 2 (a) is modeled as

Operator	*Description*
$\text{Source}_{sourceName}\$s$	Bind data source to column $\$s$
$\text{Tagger}_{pattern}\$v$	Tagger an input tuple according to *pattern*.
$\text{TokenNav}_{\$v1,path}\$v2$	Locate elements $\$v2$ that are descendants accessible via *path* from a context element $\$v1$.
$\text{ExtractUnnest}_{\$v1}\$v2$	Take inputs from $TokenNav_{\$v1,path}\$v2$ to compose tokens into XML nodes. It captures the variable binding semantics in a "for $\$v2$ in $\$v1/path$" clause.
$\text{ExtractNest}_{\$v1}\$v2$	Similar to $\text{ExtractUnnest}_{\$v1}\$v2$ but differs in that it captures the variable binding semantics in a "where $\$v1/path$" or "return $\$v1/path$" clause.
Sel_c	Apply filter c on an input tuple.
$\text{StructuralJoin}_{\$b}$	Joins input tuples whose column $\$b$ contains the same element.

Table 1: Semantics of XAT Operators

the subplan within the box in Figure 11. The patterns $\$a//item$ and $\$c//keyword$ are located by *TokenNav* operators 4 and 8 respectively. *item* and *keyword* elements are extracted by operators 7 and 11. Finally, an *item* is coupled with the *keyword* elements located within it by $StructuralJoin_{\$c}$.

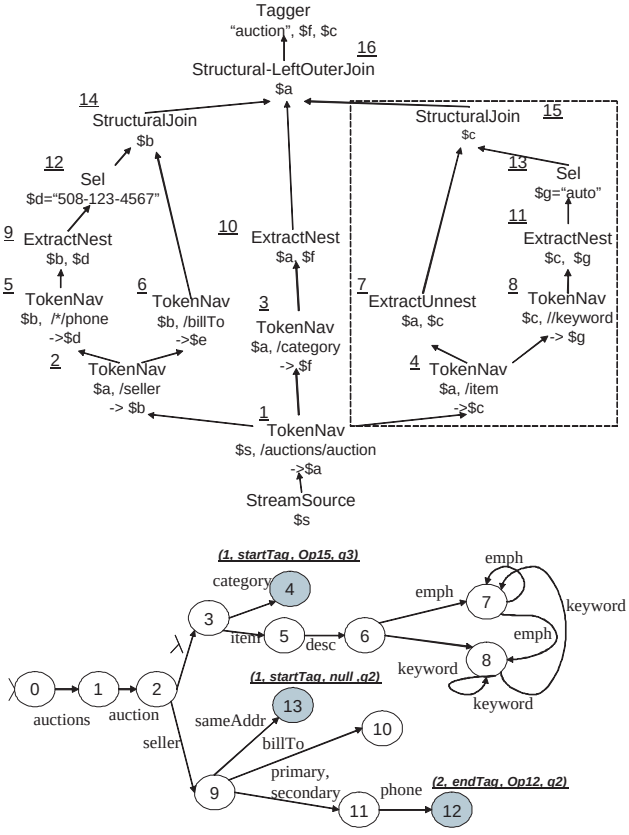


Figure 11: Encoding SQO into Algebraic Plan

6.2 Encoding Event-Condition-Actions

The bottom of Figure 11 also depicts the automaton for locating the patterns. The automaton has encoded three event-condition-actions derived in Section 5.1. Compared to the original automaton in Figure 5, new states have been added for the newly introduced patterns, e.g., state 13 for $\$b/sameAddr$ (see Example 3). The property below must hold in the automata in order for the event-condition-actions to work correctly.

Property 1 *Suppose tn and tn′ are type nodes of $\$v$ and $\$v'$ ($\$v' = \$v/p$) respectively. A set of automata states $\mathcal{S}$ will be activated by bindings of $\$v'$ of type tn′ within a binding of $\$v$ of type tn. We say the pair (tn, tn′) is mapped to $\mathcal{S}$. In the query tree, if for any two pairs of type nodes which are mapped to $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S} \cap \mathcal{S}' = \emptyset$, the "conflict-free" property holds in the automaton.*

Figure 12 shows two alternative automata constructed for the query tree in Figure 9. Both the type node pairs (c, b) and (d, b) in Figure 9 are mapped to state 4 in Figure 12 (a). The automaton in Figure 12 (a) does not satisfy the "conflict-free" property and is incorrect. This is because when state 4 is activated, we cannot infer whether the binding of $\$v$ is type c or d. We however need to know this to decide which ending mark to use for $\$v/b$. Figure 9 (b) shows a correct automaton where the above type node pairs are mapped to states 4 and 5 respectively.

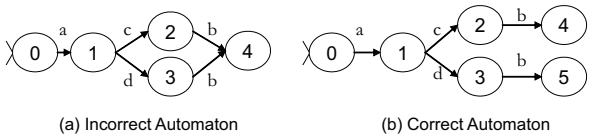


Figure 12: "Conflict-free" Property of Automata

To encode the event-condition-actions, i.e., (ending mark, $\$v/p$, type node *atn* of an ancestor context node $\$u$ of $\$v$), we first find a set of states $\mathcal{S}$ that will be activated or deactivated by the ending mark. For each state q in $\mathcal{S}$, we associate a construct (i, *tagType*, *checkOp*, p) with it, where i is the occurrence number for the ending mark found by the Occurrence Rule; *tagType* is either *startTag* or *endTag*; *checkOp* is the operator which holds the results of $\$v/p$; p is a state that will be activated by bindings of $\$u$ of type *atn*.

For example, in Figure 11, state 4 is associated with (1, *startTag*, Operator 15, state 3). It indicates when a start tag of *category* is encountered, operator 15 is checked. If operator 15 does not have any output, i.e., no $\$c$ that satisfies $\$c//keyword =$ "auto" exists, computations that would occur after state 3 is activated are all suspended. The locating of *seller* within the *auction* is not affected due to the separation of state 2 from state 3. This captures the query semantics in Figure 2. A binding of $\$a$ may still appear in the final results even if it does not contain any qualifying bindings of $\$c$.

6.3 Execution Strategy

We now present how a plan encoding event-condition-actions is executed. A construct (i, $tagType$, $checkOp$,

p) associated with state q indicates when p is activated (when $tagType$ is start tag) or deactivated (when $tagType$ is end tag) i times, if $checkOp$ does not have any output, we suspend any computations related to the states after p. p and q are activated by bindings of $\$u$ and $\$v$ respectively where $\$u$ is an ancestor context variable of $\$v$. Due to space limitations, we do not discuss the event detection and condition checking. We focus on taking actions. This process consists of three steps, namely, *computation suspension*, *temporary data cleanup* and *recovery preparation*.

In the first step, all computations within the current binding of $\$u$ identified by p are suspended. In a naive implementation, we suspend states including (1) p, (2) any states reachable via λ transitions from p, and (3) intermediate states between p and q. For example, to take action for the construct (2, *endTag*, operator 12, state 2) associated with state 12 in Figure 11, we need to remove the transitions from $q2$, $q3$ as well as $q9$, $q11$ and $q12$. We need not suspend states 4 to 8 since suspension of state 3 has ensured no transition would ever start from them. In contrast, the intermediate states between $q2$ and $q12$ such as $q9$, even though $q2$ has been suspended, still need to be suspended. Otherwise, a subsequent token after the ending mark (i.e., a </phone>) such as <*billTo*> still triggers the transition from state 9 to state 10.

We actually can reduce the number of states to be suspended so as to reduce the suspension overhead. For example, in an optimized implementation, $q11$ and $q12$ do not have to be suspended. No transition would ever start from them after the ending mark anyway.

In the second step, the temporary results originating from the current binding of $\$u$ are cleaned. For example, in a naive implementation, we clean the output buffers of operators 10 and 15 in case *category* and qualified *item* (i.e., satisfying $\$c//keyword$ = "auto") have been located within the current *auction*. However, similar to the optimization in the first step, we actually only need to clean the buffers which may have contained outputs generated within this $\$u$ before the ending mark. Therefore in the above example, we need not clean any output buffers, since *item* and *category* elements occur only after the ending mark within an *auction* (refer to Figure 3).

Third, since the suspended states need to be resumed later, we prepare for the recovery. For example, when states 2, 3 and 9 are suspended, i.e., transitions from them are removed, we set a "suspended" flag for these states and backup their transitions. Later, when a start tag of *auction* (resp. *seller*) activates states 2 and 3 (resp. *seller*), the "suspended" flag triggers the backup transitions to be recovered. Computations start again.

7 Experimentation

We implemented the SQO techniques in *Raindrop* [18, 19] using Java 1.4. Experiments are run on two

for $\$a$ in $/ProteinDatabase/ProteinEntry[p_{11}][p_{12}]$...
where $\$a/p_{21} = val_{21}$ and $\$a/p_{22} = val_{22}$ ...
return
<result> $\$a/p_{31}$, $\$a/p_{32}$, ..., </result>

Figure 13: Query Template

Pentium III 800 Mhz machines with 768M memory. One machine sends the XML stream to the second machine, i.e., the query engine. We implemented an XML parser which, assuming the incoming data is well-formed, does not check the well-formedness. The parsing time in the overall execution time thus is negligible.

7.1 Practicability of SQO Techniques

We now report the performance of our SQO techniques on a real dataset from the Protein Sequence Database (PSD) [1]. From its DTD, we can see that the data can be highly irregular. This dataset contains a sequence of *ProteinEntry* elements. A *ProteinEntry* element has 13 subelements: 8 of them can be optional; and 4 of the remaining 5 required subelements can again have optional subelements. Many real-life queries access the optional subelements, according to a biologist we have consulted.

We design a set of queries in the format in Figure 13. The notations p_{11}, ..., p_{21}, ..., p_{31}, ... stand for XPath expressions and val_{21}, val_{22}, ... stand for constant strings. Table 2 shows the features of each query.

Query	# of Filters in "for" clause	# of Paths in "return" clause	# of Selection Predicates
Q_1	1	1	0
Q_2	1	5	0
Q_3	6	5	0
Q_4	1	8	0
Q_5	1	8	0
Q_6	0	8	10

Table 2: Query Characteristics

Figure 14 shows 5 bars for each query: one for the original plan; the other three for plans applied on by the Occurrence, Exclusive or Order Rule respectively; and the fifth for the plan applied on by all three rules. Q_1, Q_2 and Q_3 are common in that no ending marks can be found by the Occurrence or Exclusive Rule. Therefore, the plans after the Occurrence or Exclusive Rule is applied are the same as the original plan. The only filter in Q_1 has a selectivity of 23%. Order Rule reduces the original execution time by 13%. Q_2 has more paths within the "return" clause so that more savings can be gained with early filtering. Order Rule reduces the original execution time by 36%. Q_3 has more filters than Q_1 and Q_2. Order Rule reduces the execution time by 40%. The performance gain difference between Q_2 and Q_3 is not major because the additional filters in Q_3 are not very selective.

Both Q_4 and Q_5 have a pattern for which Exclusive rule can find ending marks. The selectivities of the patterns are 78% and 2% respectively. For both queries, the plan optimized with the Exclusive Rule is

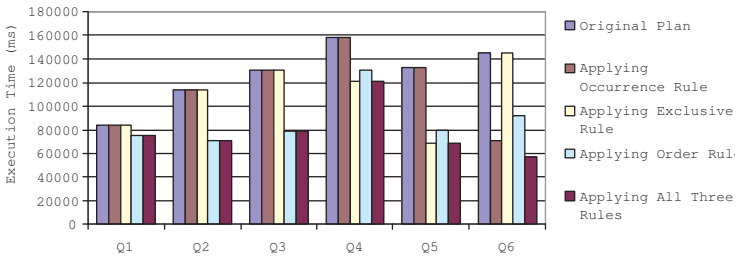


Figure 14: Effect of SQO on Queries Using a 800M PSD Dataset

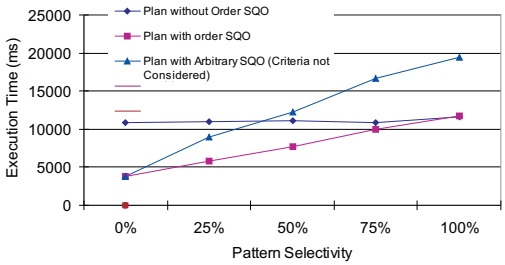


Figure 15: Comparing Plans Only Adopting Ending Marks Satisfying Criteria and Plans Adopting All Ending Marks

better than the plan optimized with the Order Rule because Exclusive Rule detects the failure of the pattern before Order Rule. The performance gain in Q_5 is more obvious due to the low selectivity of the pattern.

Q_6 contains 10 predicates. The Occurrence Rule is most useful when the occurrence number of elements is deterministic (i.e., minimal occurrence = maximal occurrence). If an element occurs less than the maximal occurrence, the Order Rule helps to catch the failure of the predicates. When these two rules are combined, the performance is the best.

7.2 Necessity of "Usefulness" Criteria

The data sets used in the rest of the paper are generated by an XML generator ToXGene [5]. They conform to the schema used in XMark [17]. We now illustrate the necessity of introducing only ending marks that satisfy the criteria in Figure 7.

For the query in Figure 2 (a), we turn off the criteria checking and adopt all ending marks found for the required patterns (we do not allow ending marks for optional patterns since they lead to incorrect results). Among 30 ending marks, only one ending mark for the pattern $\$b/billTo$ satisfies the criteria. The result is shown in Figure 15. When the selectivity of $/billTo$ is low, the only necessary ending mark of $/billTo$ often suspends transitions, including those activating the unnecessary ending marks. However, as the selectivity of $/billTo$ reaches above 30%, the overhead of unnecessary ending marks makes the plan perform even worse than the original plan.

7.3 Factors on Performance Gains

How useful an ending mark of a pattern p is depends on two factors: how often p occurs within its context node, i.e., the selectivity of p; and how much computation can be saved when an early filtering occurs, i.e., the unit gain. We now study the influence of these factors on the effectiveness of the SQO techniques.

We design three sets of queries. Each query set is meant to test the effectiveness of SQO on saving certain types of computations, i.e., path location, data buffering, or selection evaluation. Each query set is composed of three queries that differ in the unit saving. For example, in the query set for testing the saving on path recognition, the evaluation of 1, 9 and 18 path expressions can be saved when an early filtering occurs in queries 1, 2 and 3 respectively. In other words, minor, medium and major gains can happen in the three queries respectively.

Figures 16, 17 and 18 report the results on the three query sets. In each such figure, (a), (b) and (c) correspond to queries with minor, medium and major gains respectively while (d) gives a summary of the ratio of the execution time of the plan without SQO to that of the plan with SQO. The higher the ratio is, the more effective the SQO is. We can see that the lower the selectivity of the pattern with ending marks, or the bigger the unit saving is, the more effective the SQO is. In the best case of three types of queries (i.e., selectivity is 0% and unit gain is major), plans optimized with SQO reduce the execution time of original plan by 79%, 44% and 86% respectively.

7.4 Overhead of SQO

We now test the overhead of our SQO techniques. For a SQO technique, we design a query and a schema so that the SQO technique can be applied on a pattern p in the query. This query is run on a data set in which the selectivity of p is 100%. In other words, none of the ending marks of p will ever lead to any computation savings. The performance difference between such a plan and the original plan is then the overhead of SQO in worst case. Due to space limitation, we only report the overhead of Order Rule.

Order Rule may introduce multiple ending marks for one pattern in the query. For example, if we have a DTD <a (b?, c?, d?)>, both c and d can serve as the ending mark of b within a. If all b, c and d always appear within an a, the existence of b will be checked twice (equvalent to the number of its ending marks). The overhead of the plan with a different number of ending marks on different data sets is reported in Figure 19. We can see that when ending marks occur frequently (refer to the third group of bars), the more ending marks are introduced, the more expensive the query is to evaluate. However when ending marks frequently occur, the ratio of the execution time of the plan with 20 ending marks to that of the original plan is 108%, which indicates the overhead is still small.

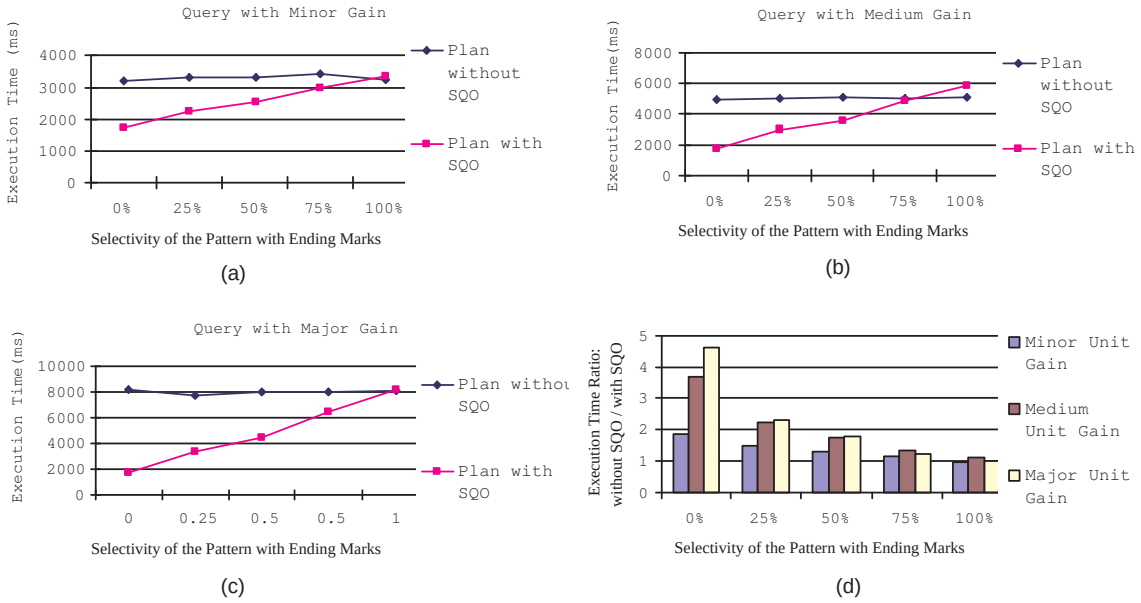


Figure 16: Effect of Pattern Selectivity/Unit Gain on Saving Path Location Cost

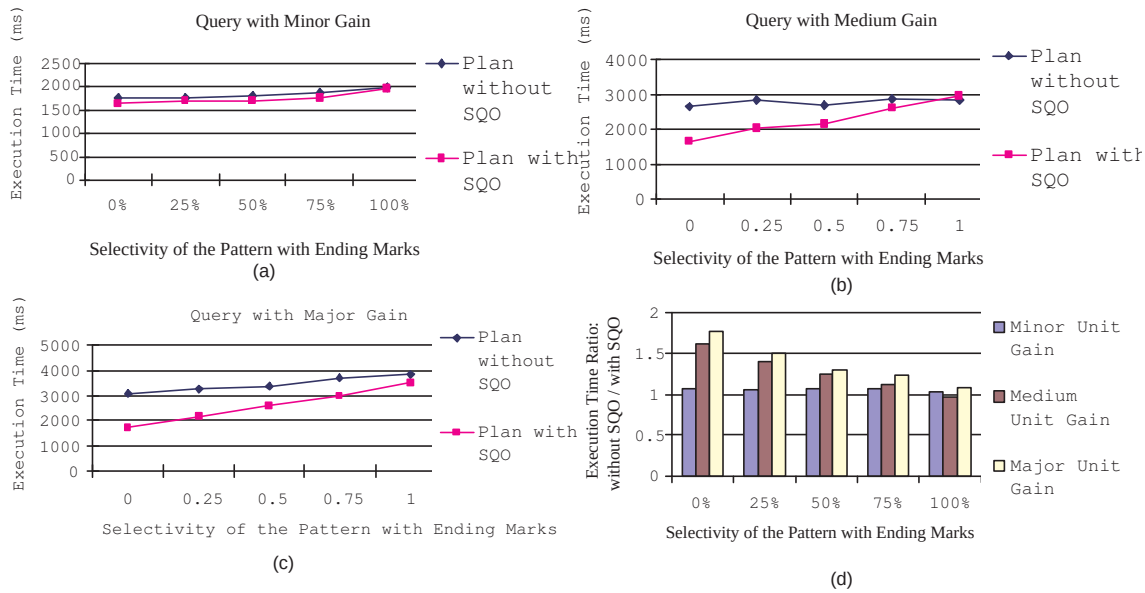


Figure 17: Effect of Pattern Selectivity/Unit Gain on Saving Buffering Cost

7.5 Experimental Summary

Our experiments on real data reveal that our SQO is practical in two senses. First, the constraints the techniques rely on do occur frequently. Second, the savings brought by the techniques can be significant.

Our experiments on synthetic data focus on three aspects. First, we show the necessity to follow the SQO design guidelines. Second, we study the impact of various factors on the effectiveness of our techniques. These factors include the kind of computation (i.e., pattern location, buffering, or selection evaluation), the unit gain, and the frequency of the occurrence of optimization. Third, we test the overhead of the SQO techniques which turns out to be rather low.

8 Conclusion

Our work provides SQO support for optimizing XQuery over XML token stream. We derive criteria for deciding what schema constraints are useful for an XQuery. Correspondingly, we develop a set of SQO rules that are able to utilize those useful constraints. An optimal rule application order is also proposed to guarantee the quality of the optimized queries. Our

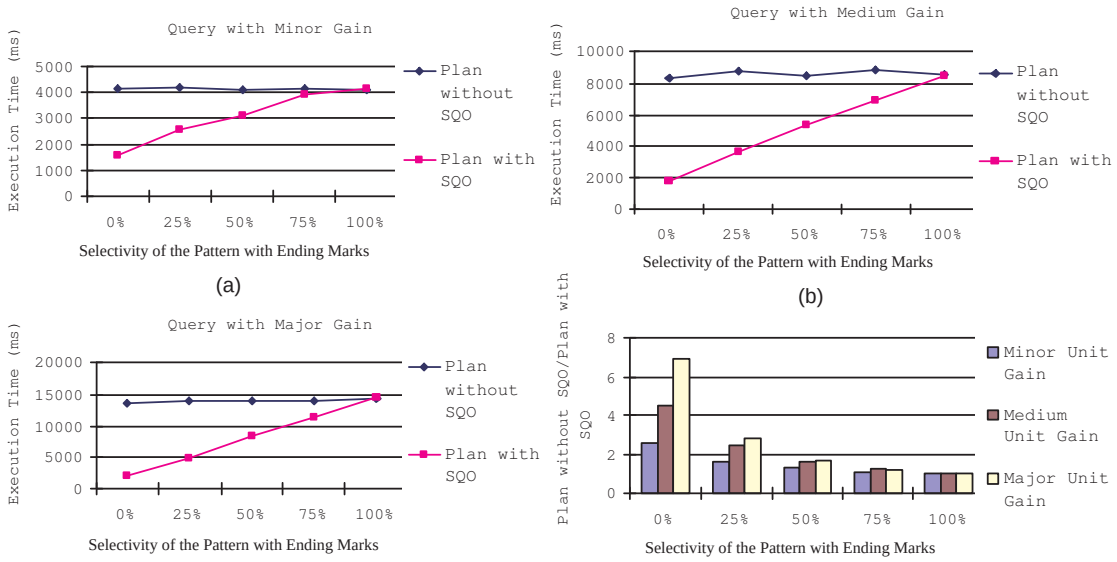


Figure 18: Effect of Pattern Selectivity/Unit Gain on Saving Selection Evaluation Cost

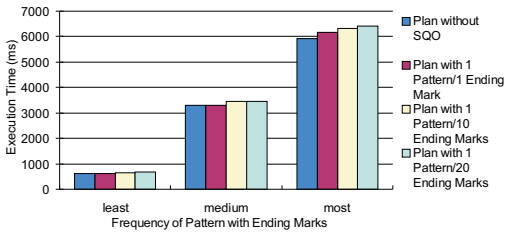


Figure 19: Overhead of Applying Order Rule in Worst Case

experiments show that these SQOs can improve the performance significantly while at the same time introducing negligible overhead in most cases.

Acknowledgement. Hong Su would like to thank IBM for PhD Fellowship support from 2001 - 2004.

References

[1] Protein Sequence Database. http://pir.georgetown.edu/.

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

[3] C. Chan, P. Felber and M. N. Garofalakis et al. Efficient Filtering of XML Documents with XPath Expressions. In *VLDB Journal 11(4)*, pages 354–379, 2002.

[4] C. Koch, S. Scherzinger and N. Scheweikardt et al. FluxQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, pages 228–239, 2004.

[5] 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.

[6] D. Florescu, C. Hillery, D. Kossmann et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB*, pages 997–1008, 2003.

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

[8] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDT*, pages 173–189, 2003.

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

[10] H. Su, E. A. Rundensteiner and M. Mani. Semantic Query Optimization for XQuery over XML Streams. Technical report, Worcester Polytechnic Institute, 2005.

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

[12] J. Grant, J. Gryz and J. Minker et al. Semantic Query Optimization for Object Databases. In *ICDE*, pages 444–453, 1997.

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

[14] M. F. Fernandez, D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE*, pages 14–23, 1998.

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

[16] Q. Cheng, J. Gryz and F. Koo et al. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB*, pages 687–698, 1999.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 974–985, 2002.

[18] H. Su, J. Jian, and E. A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *CIKM*, pages 279–286, 2003.

[19] H. Su, E. A. Rundensteiner, and M. Murali. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. In *VLDB Demo*, 2004.

[20] T. Milo and D. Suciu. Type Inference for Queries on Semistructured Data. In *PODS*, 1999.

[21] U. S. Chakravarthy, J. Grant and J. Minker. Logic-Based Approach to Semantic Query Optimization. In *ACM TODS, Vol. 15, No. 2*, pages 162–207, 1990.

[22] Z. Chen, H. Jagadish and L.V.S. Lakshmanan et al. From Tree Patterns to Generalized Tree Patterns; On Efficient Evaluation of XQuery. In *VLDB*, 2003.

[23] X. Zhang and E. A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, July 2002.