Automaton In or Out: Run-time Plan Optimization for XML Stream Processing

Hong Su and Elke A. Rundensteiner
1 Oracle Corporation, Redwood Shores, CA 94065
2 Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609
hong.su@oracle.com, rundenst@cs.wpi.edu, mmani@cs.wpi.edu

ABSTRACT
Many systems such as Tukwila and YFilter combine automaton and algebra techniques to process queries over tokenized XML streams. Typically in this architecture, an automaton is first used to locate all query patterns in the input stream and compose the matched tokens into XML element nodes. These XML nodes are then passed to the tuple-based algebra operators for further filtering or restructuring. This common processing style is however not always optimal. At times it is more efficient to retrieve only a subset of the patterns in the automaton while retrieving the rest of the patterns on the XML element nodes. In this paper, we use a cost-based solution to explore this novel optimization opportunity. We design three plan optimization algorithms, namely, MinExhaust, GreedyBasic and FastPrune. We also study how to migrate from a currently running plan to a new plan in a safe and efficient manner. Our experiments have shown that the GreedyBasic or FastPrune algorithm can quickly find a plan that is close to optimal in most scenarios. Also we illustrate that the overhead in our approach for run-time statistics collection and plan migration are very lightweight.

1. INTRODUCTION
State-of-the-art XML stream engines commonly combine automaton and algebra for query processing [9, 10, 19]. Let us use the XQuery in Figure 1 (a) as an example. This query asks to pair certain seller and bidder elements located within the same auction parent element. Figure 1 (b) depicts the Tukwila plan [19] for this query. The query processing consists of two stages: automaton processing and algebraic processing. In the first stage, all patterns in the query such as $a = auction/auction and $b = a/seller are retrieved by the X-scan operator which is an abstraction of an automaton. If certain patterns need to be further filtered or returned, the automaton extracts the tokens matching the patterns from the stream and composes them into the tree structured XML element nodes. For example, in Figure 1 (b), X-scan composes auction and seller element nodes and binds them to variables $a and $b respectively in the output tuples. In the second stage, these generated tuples are further manipulated by tuple-based algebraic operators, for example, by Select$(f = \text{zipcode}=01609) operator.

$$\text{for } a \in \text{stream(open_auctions)/auctions/auction[reserve]}$$
$$\text{where } b/\text{profile contains 'frequent' and } c/\text{zipcode=01609}$$
return
$$<\text{auction}>(b, c) </\text{auction}>$$

(a) Example Query

(b) Tukwila Query Plan: Retrieving All Patterns in Automaton

(c) Alternative Plan: Retrieving Part of Patterns in Automaton

Figure 1: Alternative Tukwila Plans

Retrieving all patterns in the automaton requires only one single pass of read over the input. It has thus been assumed by the current literature [9, 10, 19] to be the most effective manner for pattern retrieval. We have demonstrated analytically and experimentally this commonly made assumption is not necessarily true [16, 17]. In the plan in Figure 1 (b), patterns are retrieved independently. For example, whether $a/reserve occurs in a binding of $a does not affect whether $a/seller will be retrieved, and vice versa. Now consider a variation of the Tukwila plan as shown in Figure 1 (c). This plan retrieves only auctions/auction and $a/reserve in X-Scan’. In the output tuples of X-Scan’, the bindings of $a contain only those auction elements that have reserve child elements. These tuples are further manipulated to locate the remaining patterns. For example, NodeNav$a/bidder$c navigates into the bindings of $a, i.e., the auction element nodes, to locate /bidder.

The latter plan essentially “serializes” the retrieval of $a/reserve and the other patterns including $a/seller, $a/bidder, $b/\text{profile} and $c/\text{zipcode}. If only a small number of auction elements has reserve child elements, very few output tuples are generated by X-Scan’. This plan then
saves the pattern retrieval of $a/seller$, $a/bidder$, $b\text{//}profile$ and $c\text{//}zipcode$ compared to the former plan. These savings can be significant because retrieving patterns $b\text{//}profile$ and $c\text{//}zipcode$ which contain the recursion navigation step “//” can be rather expensive [11].

In our previous work [16, 17], we have proposed an algebra to support plans that can retrieve patterns both in and out of the automaton. In this paper, we now address the automaton-in-or-out optimization problem, that is, deciding which patterns should be retrieved in the automaton versus out of the automaton. The major challenges tackled by our work are as follows.

First, we define a cost model for the plans that support pattern retrieval both in and out of the automaton. Although costing of tuple-based XML operators has been studied [1, 21], there is little research on costing token-based pattern retrieval. The novelty of our cost model lies in the costing of automaton computations.

Second, we develop several plan search algorithms catering to different scenarios. When $n$, the number of patterns in the query, is small, our MinExhaust algorithm guarantees to find the optimal plan in $O(2^n)$ time. Given its high complexity, we design a second algorithm called GreedyBasic which finds a plan in $O(n^3)$ time. A third algorithm called Past-Prune expedites GreedyBasic by pruning sub-optimal plans during the plan search, thus still generating the same plans as GreedyBasic.

Third, since the statistics of the stream source are often unavailable before the stream arrives, and worse yet they may continue to change over time [3], we have to perform the optimization at run-time. We study how to collect the statistics as the plan is running. We also study how to migrate the currently running plan to a better plan. In particular, we design an efficient, incremental migration strategy that avoids recreating the automaton for the new plan. We also define a migration time window in which the migration can be safely undertaken.

Our experiments illustrate that the optimization techniques reduce the processing time significantly in many cases. The experiments also demonstrate that the run-time statistics collection and plan migration have a very low overhead.

## 2. XML STREAM QUERY PLANS

We now briefly review the XML stream processing model that combines automaton and algebra. We use the Raindrop [15–17] XML stream processor for illustration purposes. The automaton used in Raindrop is similar to those in Tukwila [19] and YFilter [9]. In fact, it serves as the core of many other automaton-style XML stream engines [11, 12, 22]. Hence the techniques discussed in this paper are not limited to Raindrop engine. Any stream engine using the automaton and algebra processing model [9, 10, 19] can apply these techniques.

Table 1 describes the Raindrop operators used in this paper. Figure 2 depicts a Raindrop plan for the query in Figure 1 (a). The highlighted subplan retrieves $Sb = a/seller$ and $Sc = b\text{//}profile$. $TokenNav_{a/seller} Sb$ locates all the tokens that are part of the $seller$ elements. $Extract_{b/Sb}$ then composes these tokens into XML element nodes. Similarly, $TokenNav_{b//profile} Sc$ and $Extract_{c/Sc}$ locate and compose profile element nodes. $ StructuralJoin_{a}$ finally joins each $seller$ element node with its descendant $profile$ element nodes.

![Figure 2: Plan for Query in Figure 1 (a)](image)

Top of Figure 3 shows the automaton which implements the $TokenNav$ and $Extract$ operators in the Raindrop plan. A stack is used to store the history of state transitions. Bottom of Figure 3 depicts the snapshot of the stack after each token (annotated under the stack) in an example stream is processed. Initially, the stack contains only the start state $q_0$ (see the first stack). As we need to define the cost for pattern retrieval in the automaton, we now describe how the automaton functions.

1. When an incoming token is a start tag:

(a) If the stack top is not empty, the automaton checks whether the states at the stack top can be transitioned. For example, when $<\text{auctions}>$ is encountered, the automaton transitions $q_0$ to $q_1$ and pushes $q_1$ onto the stack (see the 2nd stack). If no states are transitioned to, the automaton...
Figure 3: Snapshots of Automaton Stack

pushes an empty set (denoted as $\emptyset$) onto the stack. For example, before $<$annotation$>$ is encountered, stack top contains a $q_2$ (see the 3rd stack). No transition with label "annotation" starts from $q_2$. Therefore, a $\emptyset$ is pushed onto the stack after $<$annotation$>$ is processed (see the 4th stack).

(b) If the stack top is empty ($\emptyset$), the automaton directly pushes another empty set onto the stack without any transition lookup (see the 5th stack after we see $<$emphasis$>$).

2. When an incoming token is a PCDATA token: the automaton makes no change to the stack.

3. When an incoming token is an end tag: the automaton pops off the states at the stack top (see the 6th stack after we see $</$emphasis$>$).

$TokenNav$ and $Extract$ consume tokens and are called automaton-inside. The other operators such as $NodeNav$ and $Tagger$ consume tuples containing XML element nodes and are called automaton-outside. Tree-like element nodes can be accessed in a non-sequential manner, and hence are advantageous over XML tokens, which can be accessed only in a sequential manner. For example, in a tree structure, from an entry node, we can access its second child node without having to access all descendants of the first child node. It is therefore more efficient to locate patterns in such tree structures than over those only sequentially accessible tokens. Therefore once an element node has been formed, it will not be converted back to tokens again. That is to say, the output of automaton-outside operators will not be consumed by automaton-inside operators.

3. AUTOMATON PULL-OUT OR PUSH-IN REWRITE

We now present the rewrite rules that move pattern retrieval into or out of the automaton. In the plan in Figure 4 (a), $sr2 = sr1/pl1$ is retrieved in the automaton. The pull-out rule eliminates $TokenNav_{a, seller}sr2$ and $Extract_{a, seller}sr2$ and introduces $NodeNav_{a, seller}sr2$ (see the rewritten plan in Figure 4 (b)). If $Extract_{a, seller}sr1$ operator, which forms the XML nodes bound to $sr1$ so that $NodeNav_{a, seller}sr2$ can navigate into, had not existed in Figure 4 (a), it would be introduced into the rewritten plan. We can also rewrite the plan in Figure 4 (b) back to the plan in Figure 4 (a) by pushing $sr2 = sr1/pl1$ into the automaton. We refer to the pull-out or push-in of a pattern retrieval as mode change of the corresponding $TokenNav$ or $NodeNav$ operator respectively.

Let us consider a more complicated case. Suppose we want to change the mode of $TokenNav_{a, seller}sr2$ in Figure 2 where $sr2$ is further navigated into by $TokenNav_{a, profile}sr2$. We say $sr2 = a/seller$ is the ancestor pattern of $sr2 = a//profile$; or $sr2 = a//profile$ is the descendant pattern of $sr2 = a/seller$. Changing $TokenNav_{a, seller}sr2$ to $NodeNav_{a, seller}sr2$ makes $/seller$ to be rewritten in XML element nodes bound to $sa$. Since $sr2$ is located within $sa$, bindings of $sr2$ must also be XML nodes. This dictates $sr2 = a//profile$ being rewritten in XML nodes. Therefore, the mode of $TokenNav_{a, profile}sr2$ has to be changed as well. An $Extract_{a, seller}$ operator is introduced so that $NodeNav_{a, seller}sr2$ can be performed. Figure 5 shows the rewritten plan with the new operators highlighted.

On the other hand, in Figure 5, if we push in $sr2//profile$, bindings of $sr2$ must be tokens. Since $sr2$ is located within $sa$ ($sr2 = a/seller$), bindings of $sr2$ must be tokens as well. As a result, the mode of $NodeNav_{a, seller}sr2$ has to be changed. This leads to the property below.

Property 1 Secondary effect of mode change: If we change the mode of $TokenNav_{a, path}sr2$ (resp. $NodeNav_{a, path}sr2$), then we must also change the mode of any $TokenNav$ that retrieves $path$’s descendant pattern (resp. any $NodeNav$ that retrieves $path$’s ancestor pattern).

When considering the mode change of a $TokenNav$ operator, if we were to put the newly generated $NodeNav$ operator in a suboptimal position out of the automaton, we may be biased towards disallowing this mode change. We thus adopt the commonly used commute rewrite rules to optimize the tuple-based portion of the query plan. There are such rules are traditional and omitted here, but can be found in [14]. Other rewrite rules for optimizing the tuple-based portion of the plan can be equally plugged into our optimization algorithms.

4. COSTING STREAM QUERY PLANS

We now define a cost model for comparing plans with different amount of pattern retrievals in the automaton. Since the stream can be infinite, we define the cost on a finite input unit instead of the entire input. In Raindrop, we refer to the elements retrieved by the bottommost $TokenNav$ operator as the bottom input elements. We define the cost of an operator as the average time of processing the data that
Costing TokenNav Operators. When costing a TokenNav operator, we need to be careful with "amortized" computations. For example, in Figure 3, the rightmost stack contains $g_5$ and $g_6$ at the top. An incoming $</profile>$ will lead to a stack backtrack. However we cannot solely assign this backtrack cost to $TokenNav_{/seller}$/$profile$. Suppose the query does not ask for $/seller$//profile. After a $<profile>$ is processed, the stack top would contain an empty set. Next, when a $</profile>$ arrives, the backtrack is still needed to restore the stack to the status before the matching $<profile>$ has been encountered.

To avoid repeatedly counting the amortized computations, we compare the costs of automata $A_{with}$ and $A_{without}$. $A_{with}$ encodes $s_1/p1$ and all the ancestor patterns of $s_1/p1$. $A_{without}$ encodes only the ancestor patterns of $s_1/p1$. The cost difference between $A_{with}$ and $A_{without}$ is then the cost of retrieving $s_1/p1$. Using the notations in Table 2, Equation 1 captures the cost of TokenNav operators in an automaton $A$.

\[ Cost(\text{TokenNav operators in automaton } A) = \text{state transition cost for processing start tags} + \text{stack backtrack cost for processing end tags} \]

\[ = \sum_{q \in Q(A)} n_{\text{active}}(q) \cdot C_{\text{nonEmp}} \]

(3.a)

In Equation 1, Expression (1) is expanded into Expressions (3.a) and (3.b). $\sum_{q \in Q(A)} n_{\text{active}}(q)$ is equal to the number of start tokens that are processed with a non-empty stack top. Expression (3.a) then denotes the cost of processing start tags with a non-empty stack top.

The number of start tags that are processed with an empty stack top is equal to $(n_{\text{start}} - \sum_{q \in Q(A)} n_{\text{active}}(q))$. Expression (3.b) then denotes the cost of processing start tags with an empty stack top.

The cost of processing an end tag is equal to the cost of popping out the states at the stack top, namely, $C_{\text{backtrack}}$. Since there are $n_{\text{end}}$ end tags in a bottom input element, Expression (4) denotes the cost of processing end tags.

Let us use $A_{p1}$ to denote the sub-automaton that encodes $s1/p1$ only. We then have Equation 2.

\[ \text{EQUATION 2. } Cost(TokenNav_{/seller}/p1)$v2) = Cost(TokenNav operators in } A_{with}) - Cost(TokenNav operators in } A_{without}) \]

\[ = \sum_{q \in Q(A_{without})} Q(A_{without}) n_{\text{active}}(q) \cdot (C_{\text{nonEmp}} - C_{\text{emp}}) \]

\[ = \sum_{q \in Q(A)} n_{\text{active}}(q) \cdot (C_{\text{nonEmp}} - C_{\text{emp}}) \]

Costing Extract Operators. For an $Extract_{s8}$/$v2$, suppose the start token of a binding of $s2$ activates state $q$. $n_{\text{active}}(q)$ is then the number of elements bound to $s2$ in one bottom input element. Therefore, the cost of $Extract_{s8}$/$v2$ operator is $n_{\text{active}}(q) \cdot C_{\text{extract}}(q)$.

Costing NodeNav Operators. We implement $NodeNav_{/seller}/p1$/$v2$ as a width-first tree traverse. Suppose $p = p_1/p_2/.../p_n$ where $p_i$ ($1 \leq i \leq n$) is either a navigation
step or a descendant axis “//”. We first traverse all the children of the entry node $s_1$ and find those whose tag names match $p_1$. From these matched nodes, we again traverse their children to match $p_2$ and so on. We use $n_{p_1}$, $w_{p_1}$, and $C_{exist}$ to denote the number of nodes matching $p_1$/$...$/$p_n$, the number of children of these matched nodes and the time for visiting one node. The time $NodeNav_{p_{1,n}}s_1$ spends on processing one input tuple is then $\sum_{i=1}^{n} n_{p_{i-1}} w_{p_{i-1}} C_{exist}$.

Costing of Other Automaton-outside Operators. The full list of cost models can be found in [14]. Some automaton-outside operators such as Select, when appearing in one plan, must appear in all other alternative plans because we do not provide any rewrite rule to eliminate a Select operator. For these operators, we can always observe the actual time it spends on processing one input tuple in the current running plan. There is no need to provide an equation to estimate such single unit processing cost.

5. Run-Time Statistics Collection

Currently we use a simple statistics collection model. To optimize a query, we run an initial plan of this query on the incoming stream while at the same time collecting the statistics needed for this particular query. For example, we attach counters to the states in the automaton. Each time when a start tag arrives, the counter of each state at the top of the stack is incremented by 1. This way we can get $n_{active}(q)$, which is needed for costing the automaton (see Table 2), for each state $q$. The statistics collection is rather straightforward so that we omit the discussion here.


In this section, we first present a baseline search algorithm that guarantees to find an optimal plan. We then examine search redundancies in the baseline algorithm, i.e., same plans may be explored multiple times. Eliminating these redundancies gives us the MinExhaust algorithm.

6.1 Baseline Exhaustive Algorithm

Suppose an initial plan, denoted as $G$ in Figure 6, has $n$ patterns $p_1$, $p_2$, ..., $p_n$. For each pattern retrieval operator, we change its mode and get a new plan, denoted as $G_1$, $G_2$ and so on. Cycles on the new plans denote that we optimize the tuple-based portion of the new plans. Currently in Raindrop, we use the commuting optimization techniques in [18]. Any other optimization on tuple-based plans can be also plugged in here. We then treat the new plans as initial plans and repeat the above process. For example, from $G_1$, we change the mode of the operator retrieving $p_2$ (resp. $p_3$, ..., $p_n$) and get a new plan $G_{12}$ (resp. $G_{13}$, ..., $G_{1n}$). Note that we do not change the mode of the operator that retrieves $p_1$ in $G_1$ because that would generate a same plan as $G$. We continue the process until no new plans are generated. This process explores all possible plans and thus guarantees to find the optimal plan.

6.2 Eliminating Redundancy

In Figure 6, there is a path from the plan we start with, $G$, to any other plan, $G'$. We can encode the process to obtain $G'$ from $G$ in a sequence of patterns. We use $[p_1$, $p_2$, ..., $p_n]$, called a rewrite sequence, to denote that we change the mode of the operator retrieving $p_1$ first, then change the mode of the operator retrieving $p_2$ and so on. Two rewrite sequences are redundant to each other if they generate the same plans. We now present two lemmas about redundancy of rewrite sequences. Below, we use navOp to generally represent a pattern retrieval operator, i.e., either a TokenNav or a NodeNav. Also, recall the definition of ancestor and descendant patterns in Section 3.

Lemma 1. Redundancy due to Pattern Dependency:

Suppose $p_1$ is the ancestor or descendant pattern of $p_2$ (we say $p_1$ and $p_2$ have dependencies). Given a rewrite sequence $S$ containing both $p_1$ and $p_2$, there always exists another rewrite sequence that contains no patterns with dependency and yet produces the same plan.

Example 1. In Figure 7 (a), $S_1 = \$a/seller$ and $S_2 = \$b//phone$ have a dependency. We apply a rewrite sequence $[\$a/seller, \$b//phone]$ on this plan. We first pull out $S_1 = \$a/seller$. Due to the secondary effect (see Property 1 in Section 3), $S_3 = \$b//profile$ is also pulled out. Figure 7 (b) shows the plan after this rewrite. Next, we push in $S_2 = \$b//phone$. Due to the secondary effect, $S_2 = \$a/seller$ is pushed back into the automaton which undoes part of the first rewrite. We get a final plan in Figure 7 (c).

We can also derive the final plan by pulling out $\$b//profile$ and pushing in $\$b//phone$ in Figure 7 (a). The corresponding rewrite sequence is $[\$b//profile, \$b//phone]$ in which the two patterns have no dependency. The first rewrite sequence is redundant.

Lemma 2. Redundancy due to Order Insensitivity:

If a rewrite sequence $S$ does not contain patterns that have dependencies with one another, then $S$ generates the same plan as any other rewrite sequence that contains the same set of operators in $S$ but in a different order.

Example 2. In Figure 7 (a), we can either apply the rewrite sequence $[\$b//profile, \$b//phone]$ or $[\$b//phone, \$b//profile]$ to derive the plan in Figure 7 (c).

Proofs for both lemmas can be found in [14]. Based on the above two lemmas, we design an MinExhaust algorithm that eliminates the generation of any redundant plan in the
baseline exhaustive search. According to the redundancy due to order insensitivity lemma, this algorithm enumerates the combinations (instead of permutations) of \( n \) patterns. Changing the modes of all operators in one such combination leads to one alternative plan. Also, according to the redundancy due to pattern dependency lemma, we exclude any combinations that include two patterns with dependencies. Each remaining combination now leads to one unique alternative plan. It can be easily seen that for a query with \( n \) patterns, MinExhaust explores up to \( 2^n \) alternative plans.

7. FAST SEARCH WITH PRUNING

We now present the GreedyBasic and FastPrune algorithms whose complexity is more practical for large queries than MinOptimal. Given an initial plan with \( n \) patterns, GreedyBasic changes the mode of each pattern retrieval operator and gets a new plan respectively, denoted as \( G_1, \ldots, G_n \). Among the new plans, if the best plan is better than the initial plan, it is picked as the current plan. Suppose \( G_1 \) is picked after the first iteration. In the second iteration, GreedyBasic explores changing the modes of operators that retrieve \( p_2, \ldots, p_n \) in \( G_1 \) respectively. This time we get a new current plan. The iterations continue until no new plan is found to be better than the current plan.

We can further improve GreedyBasic by applying a pruning technique of sub-optimal plans. Below we use NavOp for convenience to denote a pattern retrieval operator, be it either TokenNav or NodeNav. Let us use CostCut(NavOp, \( G \)) to denote (cost of the plan after mode change of NavOp − cost of plan \( G \)). Suppose for a NavOp we can estimate a constant \( c \) such that CostCut(NavOp, \( G \)) > \( c > 0 \) for any plan \( G \). For any plan \( G \), the mode change of NavOp increases the cost. We can then safely exclude the mode change of NavOp during the plan search.

We now consider the case where NavOp is a TokenNav\( _{b/phone} \) with \( \$r1 = \$o/p0 \). Let us use \( G \) and \( G' \) to denote the plan before and after the mode change of TokenNav\( _{b/phone} \). Equation 3 denotes the cost cut by pulling out \( \$r1/p1 \).

**Equation 3.** CostCut(TokenNav\( _{b/phone} \), \( G' \)) = Cost\( (G') \) − Cost\( (G) \)

\( = \) automaton cost in \( G' \) − automaton cost in \( G \) (1)

+ non-automaton cost in \( G' \) − non-automaton cost in \( G \) (2)

\( = \) Cost(Extract\( _{b/phone} \)∗) isIntroduced (3.a)

− Cost(TokenNav\( _{b/phone} \)) (3.b)

+ Cost(NodeNav\( _{b/phone} \)) (4.a)

+ (cost of automaton-outside operators except NodeNav\( _{b/phone} \)) in \( G' \) − cost of automaton-outside operators in \( G \) (4.b)

In Equation 3, (1) is expanded into (3.a) and (3.b). An Extract\( _{b/phone} \) operator may be introduced during the rewrite (see Figure 4 in Section 3). isIntroduced in (3.a) is a boolean indicating whether an Extract\( _{b/phone} \) is introduced or not.

(2) is expanded into Expressions (4.a) and (4.b). For (4.a), Cost(NodeNav\( _{b/phone} \)) can vary in different plans depending on the position of NodeNav\( _{b/phone} \) in the plan. We can easily compute this minimal cost of NodeNav\( _{b/phone} \) by applying the commute rules to pull up NodeNav\( _{b/phone} \) from the plan. This way NodeNav\( _{b/phone} \) consumes the least input and thus costs the least. We denote this minimal cost as \( \min(\text{Cost}(\text{NodeNav}_{b/phone})) \). We then have Exp. (4.a) > \( \min(\text{Cost}(\text{NodeNav}_{b/phone})) \).

Exp. (4.b) is guaranteed to be no less than 0 if no Select or NodeNav operators in \( G \) select on or navigate into \( \$r2 \).

The optimal ordering of the automaton-outside operators is determined by their rankings [18]. The ranking function of an operator is defined on two factors, the operator’s selectivity and its processing time on one input tuple. Therefore, the optimal ordering of automaton-outside operators in \( G' \) remains the same as in \( G \). However, some operators executed before the automaton-outside operators in \( G \) can now be executed after them. For example, in Figure 7 (c), NodeNav\( _{b/phone} \) is executed before TokenNav\( _{b/phone} \).

In contrast, in Figure 7 (c), NodeNav\( _{b/phone} \) is executed after TokenNav\( _{b/phone} \). The cost of an automaton-outside operator in \( G' \) is always no less than that in \( G \). In summary, we have Exp. (4.b) > 0.

In summary, CostCut(TokenNav\( _{b/phone} \), \( G' \)) ≥
min(Cost(NodeNav_{q,p−1,p}$v2)) – Cost(TokenNav_{q,p}$v2). This leads to the following lemma.

**Lemma 3. Pruning by Bounding Cost Cut.** Given a NavOp = TokenNav_{q,p}$v2 where $v2$ is not further selected on nor navigated into, if \( \min(Cost(NodeNav_{q,p−1,p}$v2)) – Cost(TokenNav_{q,p}$v2)) \geq 0\), mode change on NavOp always leads to a worse plan.

We apply the pruning strategy on GreedyBasic, now called FastPrune. The correctness of this pruning strategy does not depend on the search strategy, i.e., it could be applied to any other algorithms including MinOptimal.

**8. RUN-TIME OPTIMIZATION**

**Run-time Statistics Collection.** To optimize a query, we run an initial plan of this query on the incoming stream while at the same time collecting the statistics. For example, we attach counters to the states in the automaton. Each time a start tag arrives, the counter of each state at the attach counters to the states in the automaton. Each time run an initial plan of this query on the incoming stream while Run-time Statistics Collection.. To optimize a query, we

**9. EXPERIMENTAL EVALUATION**

We run experiments on two Pentium III 800 Mhz machines with 512MB memory each. One machine sends XML token streams via sockets to the second machine which then processes the received data. We compare the plan search time and the quality of the plan found by MinExhaust, GreedyBasic and FastPrune algorithms. We test queries conforming to the three pattern trees shown in Figure 8, similar to previous work on XQuery optimization [4, 13, 25]. In our pattern tree, a node represents an XML element. The top node in the pattern tree represents the bottom input element. The label \( p \) on the edge from a parent node \( u \) to a child node \( v \) indicates that a path \( p \) exists within the element represented by \( u \). The bottom input elements that contain all the specified patterns are returned as the query results.

**Pruning by Bounding Cost Cut.** Given a NavOp = TokenNav_{q,p}$v2 where $v2$ is not further selected on nor navigated into, if \( \min(Cost(NodeNav_{q,p−1,p}$v2)) – Cost(TokenNav_{q,p}$v2)) \geq 0\), mode change on NavOp always leads to a worse plan.

We apply the pruning strategy on GreedyBasic, now called FastPrune. The correctness of this pruning strategy does not depend on the search strategy, i.e., it could be applied to any other algorithms including MinOptimal.

**8. RUN-TIME OPTIMIZATION**

**Run-time Statistics Collection.** To optimize a query, we run an initial plan of this query on the incoming stream while at the same time collecting the statistics. For example, we attach counters to the states in the automaton. Each time a start tag arrives, the counter of each state at the attach counters to the states in the automaton. Each time run an initial plan of this query on the incoming stream while Run-time Statistics Collection.. To optimize a query, we

**9. EXPERIMENTAL EVALUATION**

We run experiments on two Pentium III 800 Mhz machines with 512MB memory each. One machine sends XML token streams via sockets to the second machine which then processes the received data. We compare the plan search time and the quality of the plan found by MinExhaust, GreedyBasic and FastPrune algorithms. We test queries conforming to the three pattern trees shown in Figure 8, similar to previous work on XQuery optimization [4, 13, 25]. In our pattern tree, a node represents an XML element. The top node in the pattern tree represents the bottom input element. The label \( p \) on the edge from a parent node \( u \) to a child node \( v \) indicates that a path \( p \) exists within the element represented by \( u \). The bottom input elements that contain all the specified patterns are returned as the query results.

**Figure 8: Pattern Trees**

**9.1 Wide-and-Simple Pattern Trees**

**Query Sets:** We design three queries that conform to the wide-and-simple pattern tree in Figure 8 (a). These three queries differ in the number of patterns in the query, i.e., the value of \( n \) in Figure 8 (a) is 5, 10 and 20 respectively.

**Data Sets:** We use the XMark DTD [2] which describes auction data. We add more child elements to the auction root element in XMark DTD so that we are able to issue queries that contain up to 20 patterns. We use ToXGene [7] to generate two streams each of which has a size around 52M. In stream 1 (stream 2 resp.), for any of the three queries, 4/5 of the patterns have a selectivity of 10% (90% resp.) while 1/5 of the patterns have a selectivity of 90% (10% resp.). These two streams are used to test the algorithms in the extreme cases. In stream 1, most pattern retrieval operators have a low selectivity and are favored to be retrieved in the automaton. Therefore, in stream 1, the initial plan which retrieves all patterns in the automaton is close to the optimal plan. In contrast, in stream 2, most pattern retrieval operators have a high selectivity so that they are more favorable to be pulled out from the automaton in the initial plan. We expect that more changes need to be made to the initial plan to get the optimal plan in this case.

For each stream, we run an initial plan that retrieves all patterns in the automaton, collect statistics from the stream and apply the search algorithm to get a new plan. We then
run the new plan on the same stream again and measure its execution time. Table 3 reports the result. The column “effectiveness” of a search algorithm is defined as (time spent on finding a plan + time spent on executing the plan found)/(time spent on executing the initial plan). The smaller the number is (i.e., spent less time on finding a plan that runs faster), the more effective the search algorithm is.

The number of plans explored by MinExhaust is fixed given a query. When n = 10, the optimization time already far exceeds the execution time on both XML streams 1 and 2. When n = 20, MinExhaust is impractical so that we do not report it. In contrast, the number of plans explored by GreedyBasic varies with different streams because GreedyBasic terminates whenever no single mode change in the current plan yields a better plan. Although GreedyBasic explores much less plans than MinExhaust, it still succeeds to find optimal plans on both streams.

FastPrune can prune the pull-out of a pattern that has no descendant patterns. In the wide and simple queries, p1, p2, ..., and pn all have no descendant patterns. The technique of “pruning by bounding cost cut” is tried on all of them. It excludes the pull-out of those TokenNav with selectivity of 10%. The optimization time is improved most significantly in the third experiment (see row 3). This is because the search in the third experiment goes through most iterations. In each iteration, we avoid exploring the pull-out of certain patterns. So accumulatively, we save most plan explorations.

### 9.2 Wide-and-Complex Pattern Trees

We generate XML streams conforming to the DTD describing Ebay’s auction data [24]. We design a query as shown in Figure 9 with $\text{b}$, $\text{c}$, $\text{d}$ and $\text{e}$ having $2$, $2$, $12$ and $5$ filters respectively. This query conforms to the wide-and-complex pattern tree in Figure 8 (b). We test on a set of data streams with different data characteristics as shown in Table 4. The purpose is to generate a “random” data set.

for $\text{a}$ in /listing
let $\text{b} := \text{a/seller_info}[\text{seller_rating} > 4][\text{seller_name} contains “SF”];
\text{c} := \text{a/bid_list}[\ldots][\ldots];
\text{d} := \text{a/auction_info}[\ldots][\ldots];
\text{e} := \text{a/item_info}[\ldots][\ldots];
where $\text{b}$ and $\text{c}$ and $\text{d}$ and $\text{e}$ return $\text{a}$

Table 3: MinExhaust, GreedyBasic and FastPrune on Wide-and-Simple Queries (all time in ms.)

<table>
<thead>
<tr>
<th>Stream</th>
<th>Selectivity of $\text{b}$</th>
<th>Selectivity of $\text{c}$</th>
<th>Selectivity of $\text{d}$</th>
<th>Selectivity of $\text{e}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10%</td>
<td>50%</td>
<td>70%</td>
<td>90%</td>
</tr>
<tr>
<td>2</td>
<td>90%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>70%</td>
<td>90%</td>
<td>10%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 4: Random Data Sets Conforming to Ebay’s DTD: Each Stream around Size 55M

Table 5: GreedyBasic and FastPrune for Query in Figure 9 on XML Streams in Table 4

### 9.3 Deep-and-Complex Pattern Trees

It is interesting to observe that for queries conforming to the deep-and-complex pattern tree in Figure 8 (d), GreedyBasic terminates very quickly. According to redundancy due to pattern dependency lemma in Section 6, two operators that have a pattern dependency cannot both undergo mode changes. Suppose from a current plan, the mode change on $p_{12}$ ($1 < i < n$) in Figure 8 is chosen, then the mode changes on its ancestor and descendant patterns, including $p_{11}$, $p_{12}$, ..., $p_{(n-1)2}$, need no longer be considered. Suppose the mode change on $p_{11}$ is chosen. Then even more mode changes are disqualified for consideration, including mode changes on patterns $p_{11}$, $p_{21}$, ..., and $p_{n1}$.

Table 6 reports the result. Since GreedyBasic already explores a very small number of alternative plans, FastPrune brings fairly small gains and thus is not reported. Even for the queries involving a large number of patterns, GreedyBasic...
Table 6: MinExhaust and GreedyBasic for Deep-and-Complex Queries on a 51M XML Stream

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>127</td>
<td>2296</td>
<td>7456</td>
<td>19</td>
<td>205</td>
<td>8669</td>
<td>5122</td>
</tr>
<tr>
<td>4</td>
<td>695</td>
<td>8674</td>
<td>10086</td>
<td>14</td>
<td>364</td>
<td>11202</td>
<td>13569</td>
</tr>
<tr>
<td>5</td>
<td>2387</td>
<td>38590</td>
<td>12176</td>
<td>17</td>
<td>487</td>
<td>12176</td>
<td>17045</td>
</tr>
<tr>
<td>6</td>
<td>5555</td>
<td>180078</td>
<td>13468</td>
<td>20</td>
<td>647</td>
<td>14280</td>
<td>20555</td>
</tr>
</tbody>
</table>

9.4 Overhead of Statistics Collection and Plan Migration

With plan search time already studied above, we now study the overhead of statistics collection and plan migration in the run-time optimization.

Query Sets: We design two queries conforming to the pattern tree in Figure 8 (b). The two queries differ in the number of patterns in the query ($n$ in Figure 8 (b) is 5 and 10 respectively). We therefore can compare the overhead of two plans that collect different amount of statistics.

Data Sets: We also design two streams. For a query running on XML stream 1, the optimal plan is only slightly different from the initial plan. In contrast, the optimal plan of the same query on XML stream 2 is significantly different from the initial plan. We therefore can compare the overhead of a simple plan migration with a more complicated plan migration process.

In the XML stream query field, there are three major approaches. One approach is to use automata or automaton-like SAX event handlers to process the whole query [5, 6, 20, 22]. In this approach, there is no traditional algebraic query plan. Non-pattern-retrieval functionalities such as filtering or restructuring are also encoded in the automata. The input, output, and intermediate data in the processor are all tokens. No XML nodes would ever be formed. The second approach is to use algebra only. The BEA/XQRL streaming XQuery processor [8] models the query as an expression tree where an expression can be seen as an operator in an algebraic query plan. Both the input or output of expressions are tokens. The third approach is to use a dynamic programming algorithm to search for the best plan. Timber [25], another static XML processor, proposes a dynamic programming algorithm with pruning techniques to choose an optimal order for structural joins.

There have been work on run-time optimization in relational streams [3, 23, 26]. In one of the representative paradigms, namely, Eddy [3], no fixed query plans are ever constructed. Instead, each tuple, driven by the processing cost or selectivity of the operators and tuple arrival rate, can go through operators in a flexible order. The query plan is reformulated on a tuple-by-tuple basis. Eddy’s plan reformulation focuses on changing the order of operators. It is not clear how to apply this technique to choose among plans that have a different set of operators as in the automaton-in-or-out optimization techniques.

11. CONCLUSION

We have identified a unique optimization opportunity for XML stream processing. The previous literature on XML stream processing considers only query plans where all pattern retrieval is pushed into the automaton. We however find...
that for different queries and data characteristics, different automaton pushdown strategies are needed for generating optimal plans.

To explore this optimization opportunity, we use a cost-based approach. We design three plan optimization algorithms. MinExhaust enumerates all possible plans while avoiding repeated exploration of the same search space. Given a query with \( n \) patterns, it guarantees to find an optimal plan in \( O(2^n) \) time. In contrast, GreedyBasic uses heuristics to quickly find a plan in \( O(n^2) \) time. FastPrune further prunes the sub-optimal plans in the search by bounding the cost change from one plan to another plan. Our experimental study illustrates that the plans found by GreedyBasic or FastPrune algorithm are often close to the optimal plan found by MinExhaust.

In order to optimize at run-time, we design an incremental and thus efficient plan migration strategy. The migration window we define ensures that the migration can safely undertake without generating wrong result. Our experiments illustrate that our run-time statistics collection and plan migration strategies are very lightweight.

12. REFERENCES


