2006-01-30

Automaton Meet Algebra: A Hybrid Paradigm for Efficiently Processing XQuery over XML Stream

Hong Su
Worcester Polytechnic Institute

Automaton Meets Algebra: A Hybrid Paradigm for Efficiently Processing XQuery over XML Stream

by

Hong Su

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

Dec 14, 2005

APPROVED:

Prof. Elke A. Rundensteiner Advisor	Prof. Murali Mani Committee Member
Prof. George Heineman Committee Member	Prof. Mitch Cherniack External Committee Memeber Brandeis University
Prof. Michael Gennert Head of Department	

Dedicated to Mom, Dad and Wai-Po

Contents

List of Figures

Abstract

XML stream applications bring the challenge of efficiently processing queries on sequentially accessible token-based data streams. The automaton paradigm is naturally suited for pattern retrieval on tokenized XML streams, but requires patches for implementing the filtering or restructuring functionalities common for the XML query languages. In contrast, the algebraic paradigm is well-established for processing self-contained tuples. However, it does not traditionally support token inputs. This dissertation proposes a framework called *Raindrop*, which accommodates both the automaton and algebra paradigms to take advantage of both.

First, we propose an architecture for *Raindrop*. Raindrop is an algebra framework that models queries at different abstraction levels. We represent the token-based automaton computations as an algebraic subplan at the high level while exposing the automaton details at the low level. The algebraic subplan modeling automaton computations can thus be integrated with the algebraic subplan modeling the non-automaton computations.

Second, we explore a novel optimization opportunity. Other XML stream processing systems always retrieve all the patterns in a query in the automaton. In contrast, Raindrop allows a plan to retrieve some of the pattern retrieval in the au-

tomaton and some out of the automaton. This opens up an *automaton-in-or-out* optimization opportunity. We study this optimization in two types of run-time environments, one with stable data characteristics and one with fluctuating data characteristics. We provide search strategies catering to each environment. We also describe how to migrate from a currently running plan to a new plan at run-time.

Third, we optimize the automaton computations using the schema knowledge. A set of criteria are established to decide what schema constraints are useful to a given query. Optimization rules utilizing different types of schema constraints are proposed based on the criteria. We design a rule application algorithm which ensures both completeness (i.e., no optimization is missed) and minimality (i.e., no redundant optimization is introduced). The experimentations on both real and synthetic data illustrate that these techniques bring significant performance improvement with little overhead.

In conclusion, Raindrop accommodates the advantages of both automaton and algebra to efficiently process XQueries over tokenized XML streams. The proposed automaton-in-or-out and schema-based optimization techniques can be also applied to several well-known XML stream processing systems such as Tukwila and YFilter.

Acknowledgments

Rumor has it that every PhD thinks his/her PhD career is the toughest one. Well, definitely I think mine is tough. But without the patience, guidance and help from my advisor, Prof. Elke A. Rundensteiner, it could have been much tougher. My sincere thanks go to her, for everything she has done for making this dissertation possible.

I would like to thank my other committee members, Prof. Murali Mani, Prof. George Heineman and Prof. Mitch Cherniack, for their help and encouragement. I would especially like to thank Prof. Mani, who has collaborated on my dissertation work. Prof. Mani has spent enormous amount of time discussing with me and giving feedback on my papers in the past two years. My gratitude also goes to Prof. Carolina Ruiz who has provided tremendous help to me on my PhD research qualification exam and comprehensive exam.

I would like to thank Jinhui Jian, a former Raindrop team member who implemented part of the Raindrop system. My thanks also go to the Rainbow team and Cape team at DSRG lab, especially Xin Zhang, Song Wang, Ling Wang, Bradford Pielech and Luping Ding, who provided related code support.

My internships at IBM T.J. Watson Lab and HP software lab are eye opening experiences for me. It was great pleasure to work with my mentors, Dr. Ming-ling Lo (IBM), Dr. Sriram Padmanabhan (IBM) and Dr. Harumi Kuno (HP).

I feel honored to have been supported by IBM Cooperative Fellowship for three years. I thank IBM for giving me this great opportunity and offering me the summer internships at IBM Toronto lab. I deeply appreciate the warm-hearted welcome and help from my mentors there, Dr. Kelly Lyons, Mr. John Keenleyside and Mr. Calisto Zuzarte.

It is always glad to see I am not alone on this long journey. I cherish the friendship with DSRG members. The memory of the time we spent together will never fade away.

My fiance J.Wei is also my best friend and best technical support for software engineering related issues. I owe him a big, big “thank you”. My Mom and Dad have always been there for me through all the highs and lows. Their endless love and support is the best thing a daughter can ever ask for. My dearest “wai-po” (grandma at mother’s side) passed away last year. No word can express how much I miss her. I dedicate this dissertation to her.

Chapter 1

Introduction

1.1 Challenges of XML Stream Processing

There is a growing interest in data stream applications such as monitoring systems for stock, traffic and network activities [14]. Recently various research projects have targeted stream applications, such as Aurora [9], Borealis [23], STREAM [15], Niagara [22], TelegraphCQ [21], Cougar [28] and CAPE [67]. Many current research works (including all the works mentioned above) focus on relational or object applications, that is, they assume a tuple-like data model (a tuple can contain flat values and objects as in a relational or object database respectively).

Due to the proliferation of XML data in web services [54], there is also a surge in XML stream applications [18, 25, 30, 33, 32, 35, 52, 65]. The major task of a message broker is to route the XML messages to the interested parties [35]. In addition, the message brokers can also perform message restructuring or backups. For example, in an on-line order handling system [54], suppliers can register their available products at the broker. The broker will then match each incoming purchase order with the subscription and forward it to the corresponding suppliers, possibly in a restructured format at the request of the suppliers. Other typical applications include XML packet routing [8], selective dissemination of information [10], and notification systems [59].

A challenge that these XML stream applications pose is that the notion of a "tuple" no longer completely fits as a processing unit. In the XML context, we use the term "tuple" to mean a list of cells with each cell containing a set of XML element trees. This is because the XML query semantics [76] are defined as XML tree outputs computed on the given XML tree inputs. In other words, an XML tree (just like a flat value or an object in the relational or object model) is the natural granularity for processing. We use the the XML document (based on the XML benchmark XMark [7]) in Figure 1.1 (a) as an example. Each token in the XML document is annotated with a number in italic font serving as the identifier for ease of reference. This document is modeled as a tree as shown in Figure 1.2. A node in the tree represents an element, an attribute, or a PCDATA text fragment. The semantics of an expression, say, $\$s/open\_auctions/open\_auction$ in the query in Figure 1.1 (b), are defined as returning the *auction* element trees in the document, i.e., the trees rooted at the highlighted nodes in Figure 1.2.

However, XML streams are often handled as a sequence of primitive tokens, such as a start tag, an end tag or a PCDATA item. That is to say, a processing unit of XML streams has to be a token, which is at a lower granularity than an XML node. Such a processing style, i.e., a processing unit being at a lower granularity than the data model, has not been studied thoroughly by the database community as of now. This granularity difference is a specific challenge that has to be addressed for XML stream processing.

```
1<open_auctions>
 2<open_auction>
  3 <seller>
     4<sellerid> 5 001 6</sellerid>
    7<phone>8 508-1234567    9</phone>
   10<phone>11 508-0004567  12</phone>
 13 </seller>
 14 <bid>
 15  <bidder> 16<bidderid> 17 032 18</bidderid>19</bidder>
20  <bidder> 21<bidderid> 22 145 23</bidderid>24</bidder>
25 </bid>
26 <initial> 27 15.00 28 </initial>
29 </open_auction>
…
```

(a) Open_auctions Stream

```
for $a in stream( "open_auctions ")
          /open_auctions/open_auction[initial  ],
    $b in $a/seller,
    $c in $a/bid/bidder
Where
  $b/phone/text() =  "508-1234567 "
return
   <auction>
     {$b, $c}
   </auction>
```

(b) XQuery on Open_auctions Stream

Figure 1.1: Example XML Document and XQuery

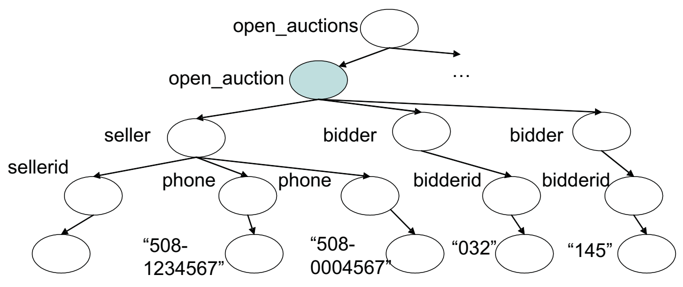


Figure 1.2: A Tree Representation of XML Document in Figure 1.1 (a)

1.2 State-of-the-Art of XML Stream Processing

Two camps of solutions have been proposed for modeling XML stream processing. The first camp of solutions uses tokens as the processing unit throughout the whole evaluation process. In contrast, the second camp of solutions uses different processing units in different stages of the evaluation. In the first stage, it consumes token inputs but generates tuple outputs. Tuple processing units are then used throughout the second stage. These two camps are further introduced below in Sections 1.2.1 and 1.2.2 respectively.

1.2.1 Pure Automaton Paradigm

The concept of an automaton was originally designed for fulfilling the functionality of matching expressions over strings. This functionality is very similar to one major XML query functionality, i.e., matching path expressions over tokens. Such close resemblance has inspired several recent projects [52, 80, 65, 35] to exclusively use automaton for the complete task of XML stream query processing. Such a *pure automaton paradigm* has to strike a balance between the expressive power of the query it can handle and the manageability of its constructs.

For example, XPush [35], using a push-down automaton, supports rather limited query capabilities. Since the push-down automaton has no output buffers, it cannot return the destination elements reachable via an XPath, not to mention restructure the destination elements. It only returns a boolean result indicating whether or not an XPath is contained in the input stream.

Some projects adopt more powerful automata in order to provide more query capabilities. Typical examples are XSM [52] and XSQ [65] supporting the XQuery and XPath languages respectively. However the support of such increased expressive power of the queries is not gained without sacrifice. The Turing-machine-like model they adopt describes the computations at a rather low level. Such a query model is somewhat similar to a procedural language that presents all internal details of the computations. Figure 1.3 gives an example of how a path expression "/a" is modeled in XSM. The automaton reads a token from the input buffer one at a time. The state transition indicates that if a certain token has been read (expressed as the part before "|"), then the corresponding actions (expressed as the part after

"|") will be taken. For instance, the transition from state 1 to state 2 indicates that if a token <a> has been read, it should be copied to a certain output buffer.

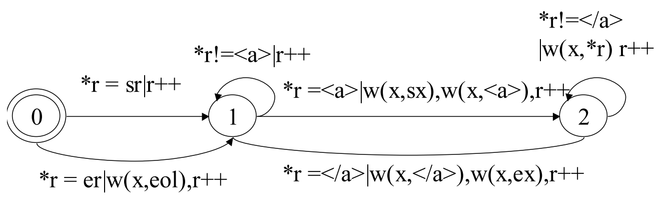


Figure 1.3: XSM Automaton for Encoding an XPath expression "/a"

Such a pure automaton paradigm has not been thoroughly studied as a query processing paradigm before by the database community. Many problems that have been well studied in tuple-based algebraic frameworks remain unexplored in this new paradigm. These include how to optimize the queries in a modular fashion, how to rewrite the queries, how to cost alternative processing plans, and how to derive efficient implementation strategies.

1.2.2 Loosely-Coupled Automaton and Algebra Paradigm

On the other hand, the tuple-based algebraic query processing paradigm[1] has been widely adopted by the database community at large for query optimization. Its success is rooted at (1) its modularity of composing a query from individual operators; (2) its support for iterative and thus manageable optimization decisions at different abstraction levels (i.e., logical and physical levels); and (3) efficient set-oriented processing capability.

It is thus not surprising that numerous tuple-based algebras (and optimization techniques based on it) for processing static XML have been proposed [78, 46, 62]

[1]In this paper, the term "algebra" specifically refers to the tuple-based algebra.

in recent years. Naturally it is expected that such an algebraic paradigm could also be utilized for XML stream processing so that existing techniques can be borrowed. However, as we have mentioned before, such an algebraic paradigm does not handle the token input data model.

Recent work, such as Tukwila [42] and YFilter [30], aims to bridge the token inputs and the tuple inputs typically assumed by the algebra paradigm. They process an XQuery in two stages. In the first stage, they use automata to handle *all* structural pattern retrieval. XML nodes are built from tokens and organized into tuples. These output tuples are then filtered or restructured in the second stage by a more conventional tuple-based algebraic engine.

We now give an example for this approach. Figure 1.1 shows an XQuery on the stream in Figure 1.1. This query pairs sellers with bidders of a certain open auction. Figure 1.4 shows the corresponding Tukwila query plan [42]. The portions underneath and above the line describe the computations in the first (i.e., automaton) and second stage (i.e., algebra) respectively. While the algebra processing is expressed as a query tree of algebra operators (skipped in the figure), the automaton processing is modeled as a single operator called *X*-Scan (YFilter also has a similar module called “path matching engine”). Tukwila assumes that retrieving a pattern in an automaton is rather cheap. Therefore they assume that all the patterns should be retrieved in the automaton. As a result, the *X-Scan* operator exposes a fixed interface to its downstream operators, namely, the bindings to all the XPath expressions in the query as annotated beside the *X-Scan* operator in Figure 1.1.

However, in our work we will illustrate that this assumption made by Tukwila does not necessarily always hold. For example, consider an alternative plan which only pushes the pattern retrieval *open_auctions*/*open_auction* and *$a*/*initial* into the

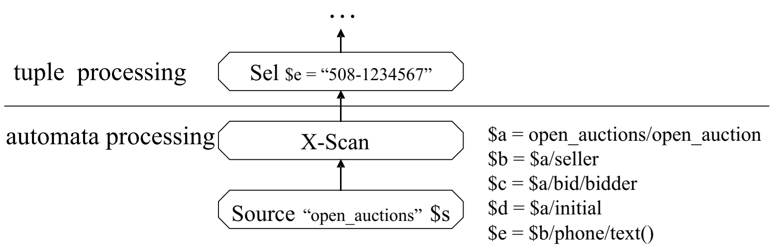


Figure 1.4: Tukwila Query Plan for Query in Figure 1.1 (b)

X-Scan operator. Only those *open_auction* elements that have *initial* child elements are extracted and XML nodes are formed out of them. They are further navigated into to locate the remaining patterns as we do when processing static XML data. Intuitively, patterns $\$a/initial$, $\$a/seller$ and $\$a/bid/bidder$ are retrieved in parallel in Tukwila while they are retrieved in a serialized manner in our alternative plan. The alternative plan is shown in Figure 1.5. When only a very small number of *open_auction* elements has *initial* child elements, this alternative plan saves most of the pattern retrieval including $\$a/seller$, $\$b/phone/text()$ and $\$a/bid/bidder$. It thus may perform better than the original Tukwila plan [2].

In summary, automaton processing, though accommodated in an algebraic framework as an operator, is not considered by the query processor to be rewritten with any other operators. Such a paradigm does not benefit from the opportunities that an algebraic framework is supposed to provide. We thus call this approach a *loosely-coupled automaton-algebra* paradigm due to the strict separation between the token-based automaton processing and the tuple-based algebraic processing.

[2] Although Tukwila provides a *follow* operator which retrieves patterns in XML nodes, it is explicitly mentioned in [42] that *follow* will be only used for retrieving XLinks instead of XPaths. It appears that Tukwila does not consider moving pattern retrieval out of the *X-Scan* operator.

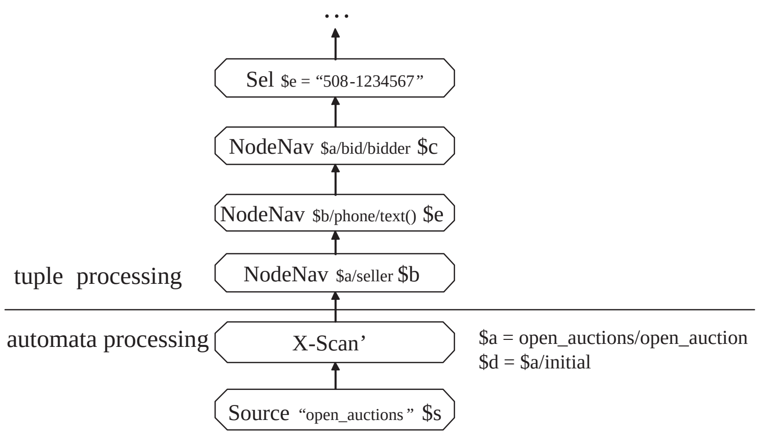


Figure 1.5: Alternative Tukwila Query Plan

1.3 Dissertation Research Focus

We instead propose a paradigm that overcomes the limitations in both the pure automaton and the loosely-coupled automaton-algebra paradigms. This paradigm tightly couples automaton and algebra style of query processing. Figure 1.6 shows an abstract comparison between the loosely-coupled and tightly-coupled approaches.

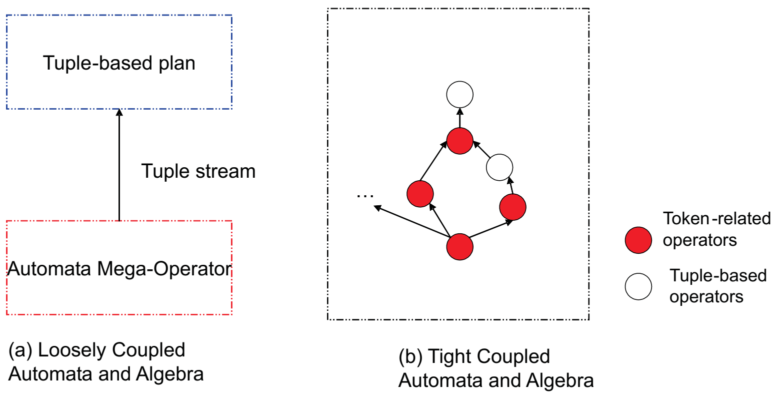


Figure 1.6: Comparisons of Two Automaton-Algebra Paradigms

In the loosely-coupled paradigm, the pattern matching type of computation on tokens (the one captured most naturally by automaton computation) and the remainder of the tuple-based computations (e.g., filtering and restructuring) communicate through a fixed interface. We express this relationship as a query plan divided into two separate boxes in Figure 1.6 (a). Instead, in the tightly-coupled paradigm, even token-based computation is modeled as a component of the query plan. This query plan is composed of multiple operators. Each such operator is at a "proper" granularity, i.e., smaller than the mega-operator *X-Scan* but still abstract enough for easy specification of the pattern retrieval semantics. In Raindrop model, these operators modeling the automaton are uniformly treated alongside with the tuple-based operators. In Figure 1.6 (b), we use one box containing all operators in the plan to express such uniformity. Rewriting rules can be applied to for example switch computations into or out of the automaton. Therefore pattern retrieval is no more restricted to be only performed in the automaton part. We now list the research issues that are addressed in this dissertation.

1.3.1 Architecture of Tightly-Coupled Automaton-Algebra Paradigm

We instead propose a paradigm that overcomes the limitations in both the pure automaton and the loosely-coupled automaton-algebra paradigms. We also model the pattern matching type of computation (the one captured most naturally by automaton processing) as a query plan composed of operators at a finer granularity than *X-Scan* [42]. Such a model offers several benefits. First, the portion of the plan modeling automaton processing can be reasoned over in a modular fashion. That is, optimization techniques can be studied for each operator separately rather than only for the automaton as a whole. Second, since the automaton processing is expressed as an algebraic plan just like the other computations, rewriting rules can be applied to, for example, switch computations into or out of the automaton. We have implemented a prototype system based on this *tightly-coupled automaton-algebra* paradigm [38].

The contributions of our system, called *Raindrop*, include:

- We accommodate both token-based processing and tuple-based processing within one *uniform algebraic model*. To model the token-based processing also as algebraic plans, we propose a data model for tokens as well as a set of algebra operators and plan structures that manipulate tokens. (Section 2.3)

- We present a three-level algebraic framework, i.e., *semantics-focused plan*, *stream logical plan* and *stream physical plan*. Each levels adds more details to the plan at the adjacent higher level. Such a layered framework enables us to reason at different abstraction levels, thus rendering optimizations tractable and practical. (Section 2.1)

- We offer a set of rewriting rules that pushes or pulls pattern retrieval into or out of the automaton. This unique optimization opportunity is not found in either pure-automaton or loosely-coupled automaton-algebra paradigms. (Section 2.4)

- We develop efficient implementations for operators modeling automaton processing. These implementations take full advantage of automaton behavior and thus are in many cases more efficient than the other implementations in the literature. (Section 2.5)

- The implementations of operators modeling automaton processing impose certain synchronization modes, i.e., certain operators must be invoked at a certain time to ensure both the correctness and efficiency of the execution of the plan. We propose a programming model to accommodate such modes. (Section 2.6)

- We perform extensive experiments illustrating that under different characteristics of the input sources, no single strategy that pushes computations into the automaton can ensure plan optimality. This confirms the necessity of reasoning about computation push-in or pull-out of the automaton. (Section 2.7)

1.3.2 Automaton-in-or-out Optimization

As mentioned in Section 1.2.2, the XML stream processing systems in the loosely coupled paradigm always retrieve all the patterns in a query in the automaton. In contrast, Raindrop allows a plan to retrieve some of the pattern retrieval in the automaton and some out of the automaton. This opens up a new optimization opportunity, called *automaton-in-out*, i.e., given a query, which pattern retrieval should be performed in the automaton and which should be performed out of the automaton.

Cost-based optimization is the mainstream optimization technique used in the database community [63]. Therefore we also use a cost-based approach to explore the automaton-in-or-out opportunity. There are three key components in a cost-based approach [69]: (1) a solution space of alternative plans, (2) a cost model for comparison of alternative plans, and (3) a search strategy for selection of a plan from the solution space. We now analyze the challenges in providing the above components that are specific to our scenario.

- Solution space can be delimited by a set of rewrite rules. Given an arbitrary initial plan of a query, the solution space is composed of all the alternative plans that can be rewritten from the initial plan by the rewrite rules. The rule that pushes or pulls pattern retrieval into or out of the automaton is the key rule we use to delimit the solution space. However, this rule alone is not enough. When we compare the costs of two plans before and after a pattern retrieval is pulled out of the automaton, in order for the comparison to be fair, we must place the pulled out pattern retrieval in an optimal position among the other automaton-outside operators. We therefore need to design more rewrite rules to move the automaton-outside operators around.

- For cost estimate, most previous research [63, 57] is on costing the tuple-based operators. For a Raindrop plan, in addition to costing the tuple-based operators, we also need to cost the token-based operators which has not been studied before. The costs of token-based and tuple-based operators must be consistently defined so that the costs of a pattern retrieval before and after it is pulled out are comparable.

- The search space in the automaton-in-or-out optimization can be exponential. Assume there are n patterns in the query, we can choose to pull zero patterns out of the automaton (C_n^0 possibility), or to pull one pattern out (C_n^1 possibilities) and so on. Even just considering pattern retrieval push-in or pull-out, we can have $C_n^0 + C_n^1 + C_n^2 + ... + C_n^n = 2^n$ alternative plans, not to mention that more alternative plans can be brought by other rewrite rules. How to efficiently find a "good" plan within such an exponential search space is a major challenge.

To complicate matters further, stream sources are often autonomous from the stream processors. It is very likely that the statistics about the stream source are unknown before the stream arrives. Ideally, we do not want to dedicate time solely for the statistics collection. The reason is that this would require buffering all the data that arrive during the statistics collection only period so that these data can be processed later. It not only puts strain on the system memory but also increases the query response time. Therefore, we instead target at run-time optimization, i.e., we run an initial plan on the stream, collect statistics and then optimize the initial plan using the statistics.

Compared to the compile time optimization, i.e., deciding a plan before any data are processed, run-time optimization faces an additional challenge, that is, plan migration [85]. In the compile time optimization, once an optimal plan is found, we simply start to run it on the data. In the run-time scenario, we however have to consider how to migrate from a currently running plan to a new plan found by the optimizer. We impose two requirements on the plan migration strategy. First, it must be correct, meaning the process with the plan migration should generate exactly the same result as that without the plan migration. Second, it should also be efficient. Otherwise the benefits of run-time optimization may be outweighed by its overhead.

When we process a query, two scenarios regarding the stream environment may arise. In the first scenario, the stream environment has stable data characteristics, i.e., the costs and selectivities of all operators in the query do not change over time. This means that we can start off with a plan, collect statistics for a moment, and then optimize the plan. After this optimization, we do not have to collect statistics or perform optimization any more since the current plan remains optimal for the rest of the execution.

In the second scenario, the data statistics change over time. Such variation commonly arises due to the correlation between the selection predicates and the order of data delivery [13]. Suppose a stream source about employees is clustered on age. A selection $salary >$ 100,000 can have higher selectivity when the data of elder employees are processed (elder employees usually have higher salary). In such a scenario, we need to constantly monitor these statistics and constantly optimize the plan. Compared to the first scenario where the optimization only needs to take place once, the second scenario poses stricter time requirement on finding a new plan quickly.

Targeting the above challenges, we have developed a set of techniques as below:

1). We design two types of rewrite rules to optimize the automaton-outside processing. One type of rules commutes the automaton-outside operators. The other type of rules changes the evaluation order of the input operators of *structural joins*. Structural joins are special joins in Raindrop that take advantage of the automaton computations to efficiently "glue" linear patterns into tree patterns. Correspondingly, we propose both heuristics and rank functions (a cost-based technique) to optimize the plan using these rewrite rules. (Sections 3.1, 3.3 and 3.4)

2). We design a cost-model for both the token-based and tuple-based computations. In particular, we observe that in the automaton computations, some cost is amortized across multiple pattern retrieval. That is to say, the cost of retrieving multiple patterns is not a simple summation of the cost of retrieving each individual pattern. We take this feature into account when developing the cost-model for the automaton computations. (Section 3.2)

3). For the stream environment with stable data characteristics, we propose an enumerative and a greedy algorithm to search through the solution space. We propose to expedite the search by reducing the time spent on costing each alternative plan. This is achieved by two techniques, incremental cost estimate and detection of same cost change. (Sections 3.5 and 3.6)

4). For the stream environment with fluctuating characteristics, we drop one type of rewrite rules which usually is less likely to affect the plan performance compared to other rewrite rules. This reduces the number of alternative plans in the search space. More importantly, within this search space, we are able to provide a greedy algorithm with pruning rules. The pruning rules exclude some alternative plans that are guaranteed not to be optimal. (Section 3.7)

5). We analyze the cost model and derive a minimal set of statistics that need to be collected at run-time. We enhance the Raindrop operators so that they can collect statistics at the same time when they are executed. (Section 3.8)

6). We design an incremental plan migration strategy that reuses the automaton of the currently running plan. We also propose a *migration window*, which is a period of time in which the migration can safely start without crashing the system nor generating incorrect results. We further show that this migration window is already “widest”. In other words, we cannot define another migration window that contains the proposed one but still guarantees that any plan migration within it is safe. (Section 3.9)

1.3.3 Schema-based Optimization for Automaton Processing

If the schema of the XML stream is known, we can use it to further optimize a Raindrop plan. Among the three major functionalities of an XML query language, namely, pattern retrieval, filtering (e.g., join) and restructuring (e.g., group-by), we can borrow existing techniques for the latter two functionalities. For example, semantic query optimization (SQO) has been well studied in relational databases. Classical techniques include join elimination, filter elimination, empty result detection etc. They utilize schema knowledge such as key/foreign key and domain constraints. As long as the counterpart schema knowledge is offered for the XML stream, these techniques can be equally applied.

In contrast, pattern retrieval is specific to the XML data model. Therefore, recent work on XML SQO techniques [11, 26, 30, 35, 53] 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* [11, 83] 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” [53] exploits the fact that an

index may have been built on element types. Given an element type, all the XML element nodes of this type (called “extents”) can be directly accessed using the index. With the schema, the element types of the query results can be inferred. The extents of these inferred element types can then be returned as query results. Since in persistent XML applications, the data is available before the query processing, it is practical to preprocess the data to build indices. This often is not the case for XML stream applications 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 (similar to the sequential access manner on magnetic tapes). 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 engines [17, 18, 30, 35, 52] have looked at the SQO opportunity. Among them, SQO in [30, 52] is not stream-specific (further discussed in Section 5.3) while SQO in [17, 35] is stream-specific but has the drawbacks listed below.

Limited Support for Queries. First, [17, 35] 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, boolean XPath matching does not differentiate */news[source]* from */news/source*. As for XSM [52], even though it supports XQuery, its SQO essentially optimizes only those parts of XQuery that are equivalent to boolean XPath matching. 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 by the time when the absence of *source* would be 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 [66], less patterns [11] or smaller extents [53]). 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 4.2.1).

Overlooking Synergy of General and Stream Specific Optimizations. Even within the scope of the queries and the constraints these SQOs address [26, 18, 52, 35], some optimization opportunities are overlooked. These opportunities arise from the synergy of general and stream-specific XML SQO. For example, type inference, which infers the types of the nondeterministic navigation steps such as "*" or "//", can be combined with the stream specific XML SQO to enable more optimization opportunities.

Lack of Strategies for Applying Possibly Overlapping Optimization Techniques. [17, 35] 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.

In this dissertation, we propose XML stream specific SQO techniques that overcome the above drawbacks. Our techniques have the below features:

- Our techniques target at XQuery, a query language that is more powerful and a super set of the boolean XPath matching and XPath query languages.
- We utilize type inference techniques in our SQO which enables more parts of a query (i.e., the parts containing "//" or "*") can be optimized.
- We design a set of optimization rules. Each rule utilizes a different schema type. We also design a rule application algorithm that ensures: no beneficial optimization is missed (completeness); and no redundant optimization is introduced (minimality).
- We incorporate these SQO techniques into our XML stream processing engine. We propose strategies for correctly and efficiently evaluating the Raindrop query plans optimized with SQO.

1.4 Dissertation Outline

We present the three research problems, namely, an automaton-algebra combined architecture, run-time optimization, and schema-based optimization, in Chapters 2, 3 and 4 respectively. Related work is described in Chapter 5. We conclude and discuss possible future directions in Chapter 6.

The materials in some chapters have been published as journal and conference papers. The materials in Chapter 2 have been presented in [47, 39, 40]. The materials in Chapter 4 have been presented in [38, 41].

Chapter 2

Raindrop Architecture: Combining Automaton and Algebra Processing Styles for XML Stream Processing

2.1 Three-level Algebraic Framework Overview

The Raindrop algebraic framework is composed of plans at three levels. A lower level plan adds more details to its adjacent higher level. An XQuery will be first compiled into the plan at the highest level. Step by step, it will be finally refined into the plan at the lowest level.

1. **Semantics-focused plan:** The plan at this level focuses on expressing the semantics of an XQuery. The nature of the input source, i.e., whether it is stored

data or tokenized stream data, is not exposed yet. General XQuery optimization techniques that are not specific to either stored or streaming data, such as XQuery decorrelation that removes nested subqueries [29, 71], and query tree minimization that removes redundant pattern retrieval [11], can be applied on this query plan.

2. **Stream logical plan:** The plan at this level is specialized to account for the input being XML token streams, instead of assuming random access to the complete XML data. For this, the data model accommodates tokenized inputs. Correspondingly, new operators and new plan structures are also introduced to model the automata processing. Moreover, rewriting rules are defined to rewrite the plans involving these new constructs.

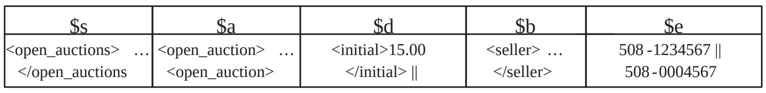
3. **Stream physical plan:** This level provides implementation details for each operator in the stream logical plan. In particular, the implementations of the operators that model automata processing have an important feature. That is, they require certain synchronization with other operators to ensure their correctness.

The semantics-focused plan is described in Section 2.2. Sections 2.3 and 2.4 discuss the stream data model and rewriting rules in the stream logical plan. The stream physical plan is presented in Section 2.5. Section 2.6 then presents a programming model for synchronizing the execution of physical operators. Experimental results are reported in Section 2.7.

2.2 Semantics-Focused Plan

Our *semantics-focused plan* is based on an XML algebra called the XML Algebra Tree (XAT) [78, 79, 77]. The algebra defines a set of operators including (1) XML-specific operators, e.g., operators for navigating into the nested XML structures, and operators for XML result construction, and (2) SQL-like operators such as *Select*, *Join*, *Groupby*, *Orderby*, *Union*, *Difference* and *Intersect*.

The input and output of the operators are a collection of *XAT tuples*. An XAT tuple is composed of cells. A cell in an XAT tuple can be one of the following types: (1) an atomic value, (2) an XML element node or (3) an unordered or ordered collection of XML element nodes[1]. Each cell is bound to a variable that is explicitly or implicitly specified in the query. Figure 2.1 depicts some example XAT tuples. $\$s$, $\$a$ and $\$b$ are explicitly defined variables. The cells bound to $\$s$, $\$a$ and $\$b$ contain one XML element node respectively. Results of $\$a/initial$ and $\$b/phone$ are not explicitly bound to a variable in the query. The query compiler assigns random variable names to their result, namely, $\$d$ and $\$e$. The cells bound to $\$d$ and $\$e$ contain a collection of XML nodes: the collection for $\$d$ ($\$d$ = $\$a/initial$) contains one element node while the collection for $\$e$ ($\$e$ = $\$b/phone/text$()) contains two text nodes. We use the notation "||" to separate items in a collection.



$s	$a	$d	$b	$e
<open_auctions> … </open_auctions	<open_auction> … <open_auction>	<initial>15.00 </initial> ||	<seller> … </seller>	508 -1234567 || 508 - 0004567

Figure 2.1: Example XAT Tuples

Table 2.1 gives the semantics of the XAT operators that will be used in this paper. The full set of XAT operators can be found in [77]. Each operator in Table 2.1 is defined in terms of the output expected when an input XAT tuple u is consumed. Some operators generate new columns. For example, a *NavUnnest* or *NavNest* operator (generally referred as *Navigate* operator when the difference be-

[1] XQuery supports both unordered and ordered expressions.

tween them is not critical) navigates into a context node and finds the destination element nodes. Such a navigate operator generates new columns containing the destination element nodes. For example, in Table 2.1, *NavUnnest* or *NavNest* has one output variable $\$col2$.

Operator	*Description*
Source$_{sourceName}$$\$col$	Bind data source specified by $sourceName$ to column $\$col$.
Tagger$_p$$\$col(u)$	Tag an input tuple u according to pattern p. Output a new tuple which is the concatenation of u and taggered data. The taggered data is bound to new column *$col*.
NavUnnest$_{\$col1,path}$$\$col2(u)$	Navigate into element node in column $\$col1$ of input tuple u. For each destination node n reachable via $path$, output a new tuple which is the concatenation of input tuple and n. n is bound to new column $\$col2$.
NavNest$_{\$col1,path}$$\$col2(u)$	Navigate into element node in column $\$col1$ of input tuple u. All destination nodes reachable via $path$ are aggregated into a collection N. Output a new tuple which is the concatenation of u and the collection. N is bound to new column $\$col2$.
Select$_c$$(u)$	If input tuple u satisfies condition c, output it.
Join$_c$$(u, v)$	If two input tuples u and v, each from a different input source, satisfy condition c, output a tuple which is the concatenation of u and v.

Table 2.1: Semantics of XAT Operators

Figure 2.2 shows the semantics-focused plan for the query in Figure 1.1 (b). It also shows the output XAT tuples of some operators. We now highlight the difference of two types of navigate operators, namely, *NavUnnest* and *NavNest*. A variable binding in a "for" clause is modeled as a *NavUnnest* operator. For example, "*for* $\$a$ *in Stream("open_auctions")/open_auctions/open_auction*" is expressed as $NavUnnest_{\$s,/open\_auctions/open\_auction}\$a$ where $\$s$ represents the input stream. The "for" clause iterates over the items in the expression results and binds the variable to each item in turn. Therefore $\$a$ of an output tuple contains only *one element node* (refer to the output of $NavUnnest_{\$s,/open\_auctions/open\_auction}\$a$ in Figure 2.2).

In contrast, a binding in a "let", "where" or "return" clause is expressed as a

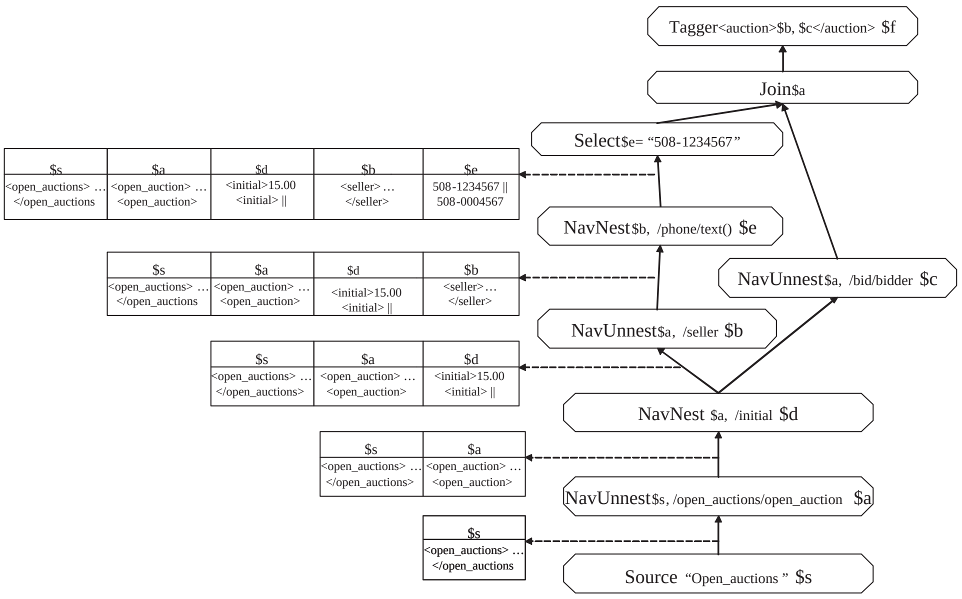

Tagger<auction>$b, $c</auction> $f
Join$a
Select$e= "508-1234567"
NavNest$b, /phone/text() $e
NavUnnest$a, /bid/bidder $c
NavUnnest$a, /seller $b
NavNest $a, /initial $d
NavUnnest$s, /open_auctions/open_auction $a
Source "Open_auctions" $s
$s
$a
$d
$b
$e
<open_auctions> … </open_auctions
<open_auction> … <open_auction>
<initial>15.00 <initial> ||
<seller> … </seller>
508-1234567 || 508-0004567

Figure 2.2: Semantics-Focused Plan (annotated with intermediate results) for querying data in Figure 1.1 (a))

NavNest operator. Each such clause binds a variable to the expression results without iteration. The name *NavNest* indicates that the output variable of an output tuple contains *a collection of element nodes* (refer to the outputs of $NavNest_{\$b,/phone/text()}\$e$ in Figure 2.2).

At this top level of the framework, we apply general optimization heuristics for query rewriting [78, 79]. For example, the operator cancel-out rule removes redundant construction of intermediate results. As another example, the navigation merge rule merges two path expressions into a longer path expression. Since these algebra rewriting heuristics are not stream specific, they are omitted here. See [78, 79] for complete details.

2.3 Modeling Token-based Processing in Algebra

The second level in the framework, i.e., the stream logical level, is targeted at processing the query on a tokenized input stream. In order to maintain the "closure" property of the algebra, i.e., use one data model throughout the algebraic framework, the XAT data model is extended at this level to accommodate data inputs. That is to say, besides the three data formats allowed in an XAT tuple cell as described in Section 2.2, a new data format called *contextualized token* is additionally supported. New operators and query plan structures are also introduced to manipulate this new data format.

2.3.1 Token-Based Data Format

The new data format, called *contextualized token*, consists of two parts: *token value* describes the local characteristics of the token; and *token context* describes the relationship between this token and the other tokens in the stream.

Token Value. A token value essentially is the information represented by a SAX event, namely, (1) the token's type (i.e., a start tag, end tag or PCDATA item), (2) the token's name (for a start or end tag) or the token's content (for a PCDATA item) and (3) the token's attributes if any (for a start tag).

Token Context. We support context regarding the forward ancestor-descendant relationships between tokens. These relationships are most commonly queried in XPath expressions using *child* and *descendant* axis specifications.

Definition 1 *A token t is* associated with *an element e if t is e's start tag, end tag or direct PCDATA content. Each token is associated with exactly one element.*

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

Example 2 *In Figure 1.1 (a), token 2 is associated with an* open_auction *element. Tokens 2 to 29 are all component tokens of this* open_auction *element.*

Three boolean functions are supported on the contextualized token types:

1. $Reachable(t_1, t_2, p)$ compares the accessibility relationship between tokens t_1 and t_2: if t_1 and t_2 are both start tags, the function returns whether the element associated with t_2 is reachable via p from the element associated with t_1.

2. $Within(t_1, t_2)$ compares the component relationship between tokens t_1 and t_2: if t_1 is a start tag, this function returns whether t_2 is a component token of the element associated with t_1.

3. $t_1 = t_2$ compares whether t_1 and t_2 are associated with the same element in terms of element identity (not only the same element content).

2.3.2 Token-Related Operators

Operator	*Description*
StreamSource$_{streamName}$ $\$col$	Bind stream source specified by $streamName$ to column $\$col$.
TokenNav$_{\$col1,path}$ $\$col2$	Locate tokens that are components of the element which is accessible via $path$ from $\$col1$.
ExtractUnnest$_{\$col1}$ $\$col2$	Compose tokens located by TokenNav$_{\$col1,path}$ $\$col2$ into XML nodes. For each destination node n reachable via $path$, output a new tuple which is the concatenation of input tuple and n. n is bound to new column $\$col2$.
ExtractNest$_{\$col1}$ $\$col2$	Compose tokens located by TokenNav$_{\$col1,path}$ $\$col2$ into XML nodes. All destination nodes reachable via $path$ are aggregated into a collection N. Output a new tuple which is the concatenation of input tuple and N. N is bound to new column $\$col2$.
StructuralJoin$_{\$e}$	Given two input tuples u and v, if $u.\$e$ = $v.\$e$, output a tuple which is the concatenation of u and v.

Table 2.2: Semantics of Token-Related Operators

We now introduce new operators that either generate or consume tuples containing contextualized tokens, as listed in Table 2.2. We denote the semantics of an operator by $Op_{params}outvar(U_n)$, where Op is the operator's name, $params$ is a list of input parameters, $outvar$ is the output variable and U_n is a collection of the first n input tuples. We use the monoid comprehension calculus [31] to express $Op_{params}outvar(U_n)$, i.e., the output of Op on U_n. Informally, a monoid comprehension is in the form of $mergeFunc\{f(a, b, ...)|$ $a \leftarrow A$, $b \leftarrow B$, ..., $pred_1$, $pred_2$, ...$\}$. In the part after "|", A (resp. B) is a collection on which variable a (resp. b) iterates. $pred_1$ (or $pred_2$) is a predicate defined over variables such as a and b. The function $f(a, b, ...)$ constructs a collection that contains only one tuple. This single tuple in the collection is composed of a, b and so on. In the part before "|", the function $mergeFunc$ merges multiple collections into one collection. In summary, a monoid comprehension returns a collection, which is generated as follows:

```
$result$ : = an empty collection;
for each $a$ in $A$, $b$ in $B$, ...,
    if $pred_1 \wedge pred_2 \wedge$ ...
        $result$ : = $result$ $mergeFunc$ $f(a, b, ...)$
return $result$.
```

For example, a monoid comprehension $\cup\{(a, b)|a \leftarrow \{1, 2\}, b \leftarrow \{4\}\}$ first creates two collections {(1, 4)} and {(2, 4)}, then merges them using the function $\cup$ and returns a collection {(1, 4), (2, 4)}.

The notations used for defining the semantics of operators are listed in Table 2.3. We illustrate each operator using the example in Figure 1.1. Each token in the

Notation	Explanation
$u.\$c$	get binding of cell $\$c$ from tuple u
$< c_1 = v_1, c_2 = v_2, ... >$	construct a tuple with cell c_1 assigned the value v_1, cell c_2 assigned the value v_2...
$u_1 \circ u_2$	construct a tuple by concatenating tuples u_1 and u_2. If u_1 and u_2 contain cells that are bound to the same variable, remove one of the redundant cells.
$+\!\!+$	merge operator for list (a list is represented as [ ])
$\oplus$	compose tokens into XML nodes

Table 2.3: Notations Used for Defining Token-Related Operators

input or output is annotated with its identifier.

StreamSource

This operator binds the sequence of the tokens from the stream specified by $strName$ to the output variable.

Example 3 *For* $StreamSource_{\text{“}open\_auctions\text{”}}\$s$*, its first 4 output tuples are:*

$\$s$	$\$\widetilde{s}$
<open_auctions>^{1}	*<open_auctions>*^{1}
<open_auctions>^{1}	*<open_auction>*^{2}
<open_auctions>^{1}	*<seller>*^{3}
<open_auctions>^{1}	*<sellerid>*^{4}

Suppose the operator now consumes the first n tokens, denoted as T_n, in the stream. n output tuples are constructed. Each output tuple contains $\$s$, the explicitly specified output variable of $StreamSource$ operator. $\$s$ is bound to the start tag of the root element in the stream, denoted as t_0. t_0 identifies the root element and thus also identifies the stream (in the rest of this section, we always use a start tag to identify its associated element). Simply identifying an element is not good enough. We are also interested in the element content. Therefore each output tuple also contains an implicit variable $\$\widetilde{s}$ for $\$s$. $\$\widetilde{s}$ is bound to a component token of the element associated with $\$s$. In short, an output tuple of $StreamSource$ contains an identifier of the stream and a component token of the stream.

$$\boxed{\begin{array}{l} StreamSource_{strName}\$s(T_n) = \\ +\!\!+ \{< \$s = t_0, \$\widetilde{s} = t> | t \leftarrow T_n\} \end{array}}$$

Token Navigate Operator TokenNav

$TokenNav_{\$col1,path}\$col2$ operator recognizes patterns over the token stream. It returns the component tokens of the destination element $\$col2$ accessible via $path$ from the context element $\$col1$. Each output tuple contains such a component token and the token identifying the destination element.

Example 4 *If $TokenNav_{\$s,/open\_auctions/open\_auction}\$a$ takes the first 4 output tuples from $StreamSource_{\text{``}open\_auctions\text{''}}\$s$ in Example 3 as input, its output is:*

$\$s$	$\$a$	$\$\widetilde{a}$
<open_auctions>^{1}	<open_auction>^{2}	<open_auction>^{2}
<open_auctions>^{1}	<open_auction>^{2}	<seller>^{3}
<open_auctions>^{1}	<open_auction>^{2}	<sellerid>^{4}

For example, the second output tuple represents that token 2, i.e., <open_auction>, is reachable via $/open\_auctions\ /open\_auction$ from token 1. It also represents that token 3, i.e., <seller>, is a component of the element associated with token 2.

$$\boxed{\begin{array}{l} TokenNav_{\$col1,path}\$col2(U_n) = \\ +\!\!+\{u_1\circ < \$col2 = u_1.\$\widetilde{col1}, \$\widetilde{col2} = u_2.\$\widetilde{col1} > | \\ u_1 \leftarrow U_n,\ u_2 \leftarrow U_n,\ Reachable(u_1.\$col1, u_1.\$\widetilde{col1}, path), \\ Within(u_1.\$\widetilde{col1}, u_2.\$\widetilde{col1})\} \end{array}}$$

An input tuple $u_1 \in U_n$ to $TokenNav_{\$col1,path}\$col2$ contains bindings of variable $\$col1$ and the explicit variable $\$\widetilde{col1}$. If $Reachable(u_1.\$col1, u_1.\$\widetilde{col1}, path)$ is true, then $u_1.\$\widetilde{col1}$ is the start tag of a destination element. For each component token of this destination element, i.e., for each $u_2.\$\widetilde{col1}$ that has $u_2 \in U_n$ and $Within(u_1.\$\widetilde{col1},\ u_2.\$\widetilde{col1})$ is true, an output tuple is constructed. The output tuple is the concatenation of u_1, the start tag of the destination element ($\$col2 = u_1.\$\widetilde{col1}$), and the component token (i.e., $\$\widetilde{col2} = u_2.\$\widetilde{col1}$).

Composition Operator ExtractUnnest

Sections 2.3.2 and 2.3.2 present two extract operators, $ExtractUnnest_{\$col1}\$col2$ and $ExtractNest_{\$col1}\$col2$ (generally referred as *Extract* operator). Both of them must have an input operator in the form of $TokenNav_{\$col1,path}\$col2$. The input $TokenNav_{\$col1,path}\$col2$ locates the component tokens of $\$col2$ while the extract operators composes these component tokens into XML nodes.

Example 5 *Suppose $ExtractUnnest_{\$s}\$a$ consumes the first 3 output tuples of $TokenNav_{\$s,/open\_auctions/open\_auction}\$a$ in Example 4. It generates the below tuple.*

$\$s$	$\$a$	$\$\widetilde{a}$
<open_auctions>^{1}	*<open_auction>*^{2}	*<open_auction>*^{2}*<seller>*^{3} *<sellerid>*^{4}

The cell $\$\widetilde{a}$ contains a yet-to-be-completed element node. It is composed of tokens 2, 3 and 4. This tuple is a partial output for the input seen so far. Eventually, $\$\widetilde{a}$ would contain a complete element node composed from token 2 to token 29.

$$\boxed{ExtractUnnest_{\$col1}\$col2(U_n) = Group_{\{\$col2\},\oplus(\$\widetilde{col2})}U_n}$$

$Group_{\{\$col2\},\oplus(\$\widetilde{col2})}U_n$ is a function that groups input tuples on destination node $\$col2$ so that the component tokens (i.e., $\$\widetilde{col2}$) of the same destination node are all collapsed into one group. The component tokens within one group are then composed (represented as $\oplus$) into one element node.

Composition Operator ExtractNest

The difference between $ExtractNest$ and $ExtractUnnest$ is analogous to the difference between $NavNest$ and $NavUnnest$ mentioned in Section 2.2. The destinations found within the same context are aggregated into one single collection.

Example 6 *Suppose $ExtractNest_{\$b}\$e$ consumes the first 2 tuples generated by $TokenNav_{\$b,/phone/text()}\$e$ which are,*

$\$s$	$\$a$	$\$b$	$\$e$	$\$\widetilde{e}$
<open_auctions>^{1}	*<open_auction>*^{2}	*<seller>*^{3}	*<phone>*^{7}	*508-1234567*^{8}
<open_auctions>^{1}	*<open_auction>*^{2}	*<seller>*^{3}	*<phone>*^{10}	*508-0004567*^{11}

the output tuple below is generated:

$\$s$	$\$a$	$\$b$	$\$e$
<open_auctions>^{1}	*<open_auction>*^{2}	*<seller>*^{3}	$508-1234567\|\|508-0004567$

The output tuple represents that within an open_auction *element with a start tag 2 (bound to $\$a$), there is a* seller *child element with a start tag 3 (bound to $\$b$). So far, two* phone *subelements of this* seller *have been formed and aggregated into one collection (bound to $\$e$).*

$$\boxed{\begin{array}{l} ExtractNest_{\$col1,path}\$col2(U_n) = \\ Group_{\{\$col1\},+\!\!+(\$\widetilde{col2})}(Group_{\{\$col2\},\oplus(\$\widetilde{col2})}U_n) \end{array}}$$

From the definitions of $ExtractNest$ and $ExtractUnnest$, we can see $ExtractNest$ has a further grouping on the output of $ExtractUnnest$ by the context node $\$col1$. In this way, all the destinations found within the same context are grouped together and aggregated (represented as $++$) into one collection.

Structural Join

In Figure 1.1 (b), path expressions $\$a/seller$ and $\$a/bid/bidder$ share the same context variable $\$a$. To capture this relationship, *StructuralJoin* takes outputs of two *Extract* operators as inputs and "glues" bindings of individual path expressions.

Example 7 *Suppose output tuples of $ExtractNest_{\$a}\$b$ and $ExtractNest_{\$a}\$c$ are joined on $\$a$. Assume the left input is one XAT tuple:*

$\$s$	$\$a$	$\$b$
<open_auctions>^{1}	*<open_auction>*^{2}	*<seller><sellerid>001 ...</seller>*

and the right input corresponds to two XAT tuples:

$\$s$	$\$a$	$\$c$
<open_auctions>^{1}	*<open_auction>*^{2}	*<bidder><bidderid><032> ...</bidder>*
<open_auctions>^{1}	*<open_auction>*^{2}	*<bidder><bidderid><145> ...</bidder>*

Then two output tuples are constructed as below:

$\$s$	$\$a$	$\$b$	$\$c$
<open_ auctions>^{1}	*<open_ auction>*^{2}	*<seller><sellerid> 001 ...</seller>*	*<bidder><bidderid> 032...</bidder>*
<open_ auctions>^{1}	*<open_ auction>*^{2}	*<seller><sellerid> 001 ...</seller>*	*<bidder><bidderid> 145...</bidder>*

The output tuples represent that within an open_auction *element with start tag 2 (bound to $\$a$), there is a* seller *element (bound to $\$b$) and two* bidder *elements (bound to $\$c$).*

Below, we use UL_{n1} and UR_{n2} to denote the first $n1$ and $n2$ input tuples from the left and right upstream operators respectively.

$$\boxed{\begin{array}{l} StructuralJoin_{\$e}(UL_{n1}, UR_{n2}) = \\ \\ +\!\!+ \{ul \circ ur | ul \leftarrow UL_{n1}, ur \leftarrow UR_{n2}, ul.\$e = ur.\$e\} \end{array}}$$

2.3.3 Stream-Specific Plan Structures

XML streams arrive on the fly so that unless a token is explicitly stored, it can be accessed only once. The token-related operators must be connected in a way which ensures that no repetitive token access occurs. An automaton can read data once and concurrently recognize multiple patterns. We therefore propose a special plan structure that models the automata behavior.

Each pattern is defined as a sequence of states in the automaton. The input drives the transition between these states. Figure 2.3 depicts a stream logical plan adopting this processing style. $TokenNav_{\$a,/seller}\$b$ and $TokenNav_{\$a,/bid/bidder}\$c$ share the same upstream operator *TokenNav*$_{\$s,/open\_auctions/open\_auction}\$a$. This sharing indicates that, for every token read from the common upstream operator, we try to match either $\$a/seller$ or $\$a/bid/bidder$. The two downstream extract operators $ExtractUnnest_{\$a}\$b$ and $ExtractUnnest_{\$a}\$c$ compose the *seller* and *bidder* element nodes respectively. Later on, $StructuralJoin_{\$a}$ glues the *seller* and *bidder* elements which are subelements of the same *open_auction* element into one output tuple.

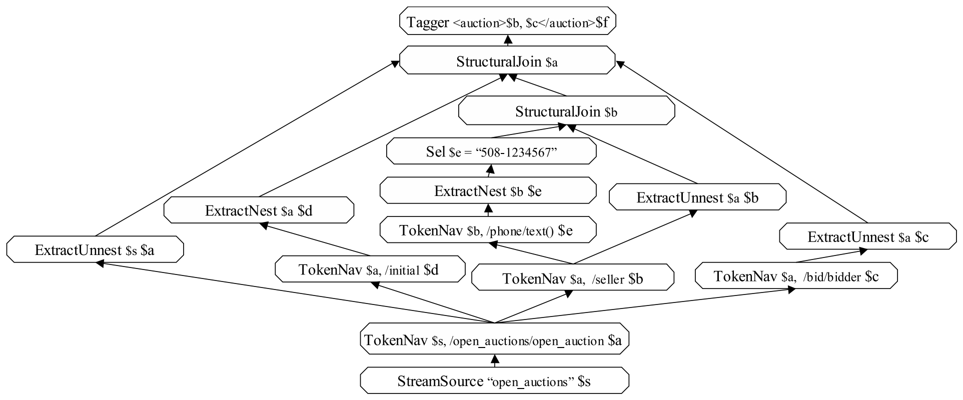


Figure 2.3: Stream Logical Plan for the Semantics-focused Plan in Figure 2.2

2.3.4 Regular Tuple-based Operators

Apart from the token-based operators, the rest of the operators in a Raindrop plan consume or generate the "regular" cells of tuples, i.e., they do not consume or generate tokens. $NavUnnest$, $NavNest$, $Select$ and $Tagger$ defined in Table 2.1 are examples of such operators.

2.4 Rewrite Rules Involving Token-Related Operators

We now present two rewrite rules that involve token-related operators. The first rewrite rule maps the semantics-focused plan to a default stream logical plan while the second rewrite rule provides alternative stream logical plans other than the default one.

2.4.1 Default Mapping Rewrite Rule

The *default mapping rewrite rule*, shown in Figure 2.4, provides a default mapping from a semantics-focused plan (left in Figure 2.4) to a stream logical plan (right in Figure 2.4). First, the general *Source* element is replaced by a more specific *StreamSource* element. Second, the bottommost $NavUnnest$ operator (resp. $NavNest$) is mapped to a $TokenNav$ and an $ExtractUnnest$ (resp. $ExtractNest$) pair. The purpose of this rewriting is to avoid the extraction of the complete incoming stream. Extracting the complete incoming stream not only increases the response time but also may be impossible when the input is infinite. Therefore by default, we push the bottommost node navigate operator into the automata.

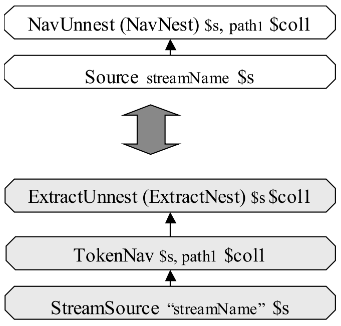

NavUnnest (NavNest) $s, path1 $col1
Source streamName $s
ExtractUnnest (ExtractNest) $s $col1
TokenNav $s, path1 $col1
StreamSource "streamName" $s

Figure 2.4: Default Mapping Rewrite Rule

For example, this rule can be applied on Figure 2.2 to derive a default stream logical plan as shown in Figure 2.5. In the default stream logical plan, all the *open_auction* elements are extracted. The composed element nodes will be navigated by later operators.

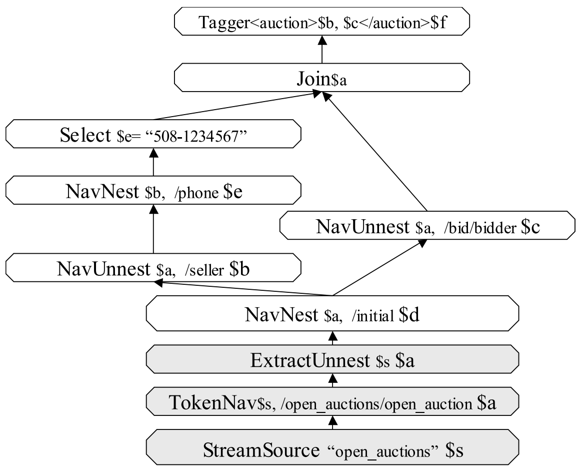

Tagger<auction>$b, $c</auction>$f
Join$a
Select $e= "508-1234567"
NavNest $b, /phone $e
NavUnnest $a, /bid/bidder $c
NavUnnest $a, /seller $b
NavNest $a, /initial $d
ExtractUnnest $s $a
TokenNav$s, /open_auctions/open_auction $a
StreamSource "open_auctions" $s

Figure 2.5: Plan Rewritten from Figure 2.2: Default Mapping Rewrite Rule Applied on $NavUnnest_{\$s,/open\_auctions/open\_auction}\$a$

2.4.2 Token-or-Node Mode Change Rule

We provide *token-or-node mode change rule* which rewrites an operator that retrieves pattern on XML nodes (i.e., *NodeNav*) to an operator that retrieves pattern on tokens (i.e., *TokenNav*). There are two circumstances for applying this rule. Figure 2.6 shows the rule in the first circumstance. In this circumstance, the top plan does not contain a $StructuralJoin_{\$col1}$. When rewrite rule is applied on $NavUnnest(NavNest)_{\$col1,path1}\$col2$, the top plan is rewritten to the bottom plan in which a $Structuraljoin_{\$col1}$ is introduced. We call this rewrite rule a *mode change with introducing/eliminating StructuralJoin* rule.

We now explain why this rewriting results in an equivalent plan. The internal logic of $NavUnnest(NavNest)_{\$col1,path1}\$col2$ can be divided into two parts. First, it locates the destination element nodes $col2. This is achieved by $TokenNav$ and $ExtractUnnest(ExtractNest)$ in the rewritten plan. Second, it

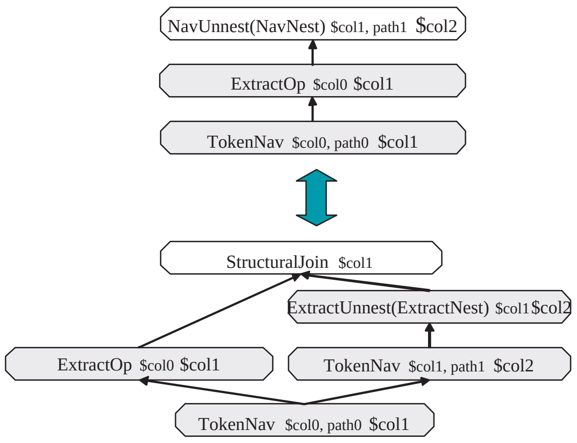

NavUnnest(NavNest) $col1, path1 $col2
ExtractOp $col0 $col1
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col0, path0 $col1

Figure 2.6: Mode Change with Introducing/Eliminating StructuralJoin

generates an output tuple for each destination element node. Each output tuple is a concatenation of the input tuple and the destination element node. This is equivalent to the cartesian product of the input tuples and the set of destination element nodes. $StructuralJoin$ in the rewritten plan captures this part. In summary, the rewritten plan has the same logics as the original plan.

Figure 2.7 shows the rewrite in the second circumstance. The top plan is the bottom plan in Figure 2.6 which contains a $StructuralJoin_{\$col1}$. When the rewrite rule is applied on $NavUnnest(NavNest)_{\$col1,path1}\$col2$, it will not introduce another $StructuralJoin_{\$col1}$. The resulted $TokenNav_{\$col1,path1}\$col2$ and $ExtractUnnest(ExtractNest)_{\$col1}\$col2$ will be placed under the existing $StructuralJoin_{\$a}$. We call this a *mode change without introducing/eliminating StructuralJoin* rule.

Figure 2.8 shows an *Extract Elimination* rule. In the top plan, no regular tuple-based operators consume $\$col1$. That is to say, $\$col1$ need not appear in the output tuples of $StructuralJoin_{\$col1}$. We can then eliminate $Extract_{\$col0}\$col1$ which

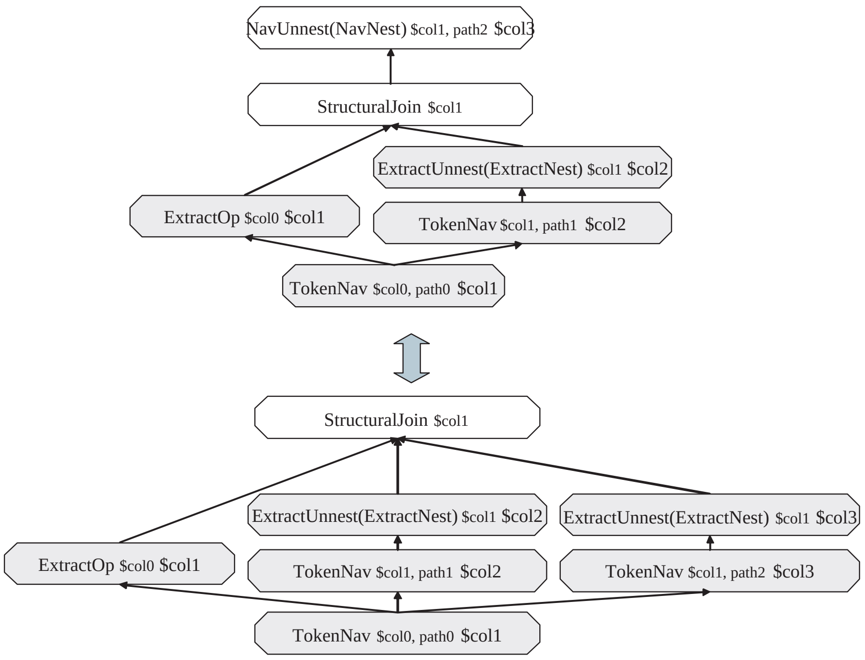

NavUnnest(NavNest) $col1, path2 $col3
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1

Figure 2.7: Mode Change without Introducing/Eliminating StructuralJoin

extracts component tokens of bindings of $\$col1$ into XML element nodes. For example, we can apply this rule on the plan in Figure 2.3 and eliminate the operator $ExtractUnnest_{\$s}\$a$ since no operator above $StructuralJoin_{\$a}$ needs to consume bindings of $\$a$.

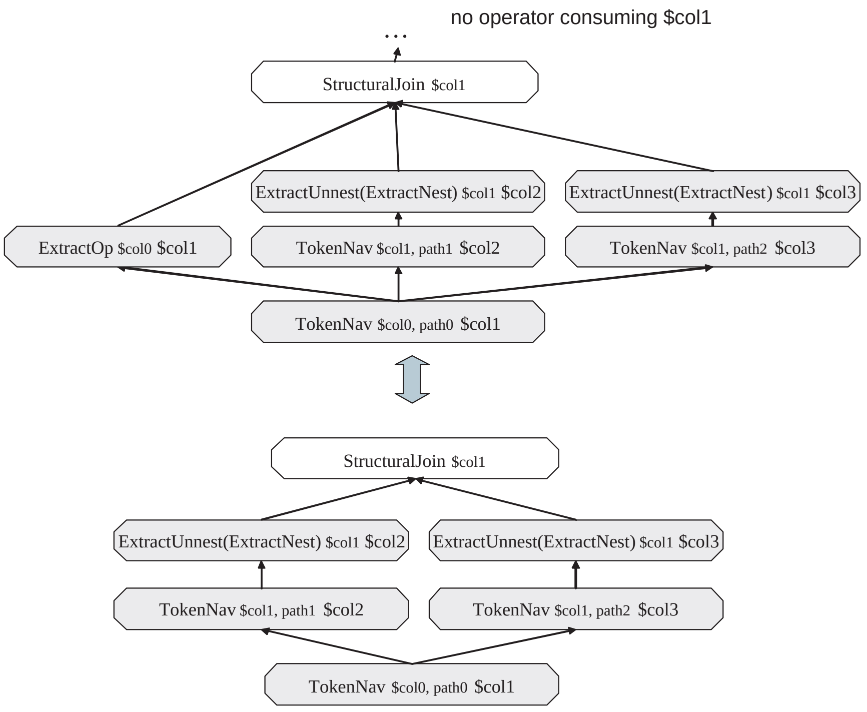

no operator consuming $col1
...
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1

Figure 2.8: Eliminate $Extract_{\$col0}\$col1$ when no Regular Tuple-based Operator Consumes $\$col1$

2.4.3 Secondary Effect of Mode Change of Pattern Retrieval

Changing the mode of a pattern retrieval operators op may force the other operators which have *pattern dependency* relationships with op to have mode changes as well.

Definition 3 *Suppose we have two pattern retrieval operators $navOp_1$ and $navOp_2$ which retrieve \$$v$ = \$$u/p1$ and \$$y$ = \$$x/p2$ respectively. $navOp_1$ and $navOp_2$ can be either a $TokenNav$ type or $NodeNav$ type and they two do not have to be the same types. If \$$x$ = \$$v/p3$, we say \$$u/p1$ is the* ancestor pattern *of \$$x/p2$; \$$x/p2$ is the* descendant pattern *of \$$u/p1$ of \$$v$). We also say $navOp_1$ and $navOp_2$ have a* pattern dependency *relationship.*

When we retrieve a pattern in the automaton, its ancestor patterns have to retrieved in the automaton. When we retrieve a pattern out of the automaton, its descendant patterns have to be retrieved out of the automaton. We therefore have Property 1.

Property 1 ***Secondary effect of Mode Change of Pattern Retrieval:*** *When the mode of a token pattern retrieval is changed, the mode of all token pattern retrieval on the descendant patterns will be also changed; when the mode of a node pattern retrieval is changed, the modes of all node pattern retrieval on the ancestor patterns will be also changed.*

For example, if "StructuralJoin introduced" rewrite rule in Figure 2.6 is applied to change the mode of $NavNest_{\$b,/phone}\$e$ in Figure 2.5, all ancestor patterns of $\$b/phone$, namely, $\$a/seller$ and $\$s/open\_auctions/open\_auction$, must all be performed on tokens. $\$s/open\_auctions/open\_auction$ is already performed on tokens. Therefore we only need to push in $\$a/seller$ before we push in $\$b/phone$. Figure 2.9 shows the plan after the mode change rule is applied on $NavUnnest_{\$a,/seller}\$b$ in Figure 2.5. We then apply "StructuralJoin introduced" rule on $NavNest_{\$b,/phone}\$e$ in Figure 2.9 and get a plan shown in Figure 2.10.

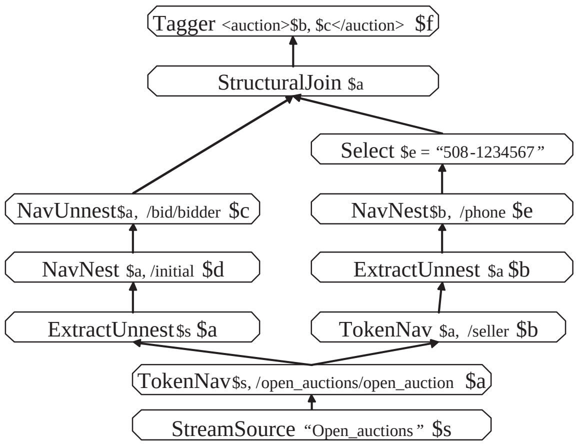

Tagger <auction>$b, $c</auction> $f
StructuralJoin $a
Select $e = "508-1234567"
NavUnnest$a, /bid/bidder $c
NavNest$b, /phone $e
NavNest $a, /initial $d
ExtractUnnest $a $b
ExtractUnnest$s $a
TokenNav $a, /seller $b
TokenNav$s, /open_auctions/open_auction $a
StreamSource "Open_auctions" $s

Figure 2.9: Plan Rewritten from Figure 2.5: Pattern Retrieval on Token-or-Node Mode Change Rule Applied on $NavNest_{\$a,/seller}\$b$

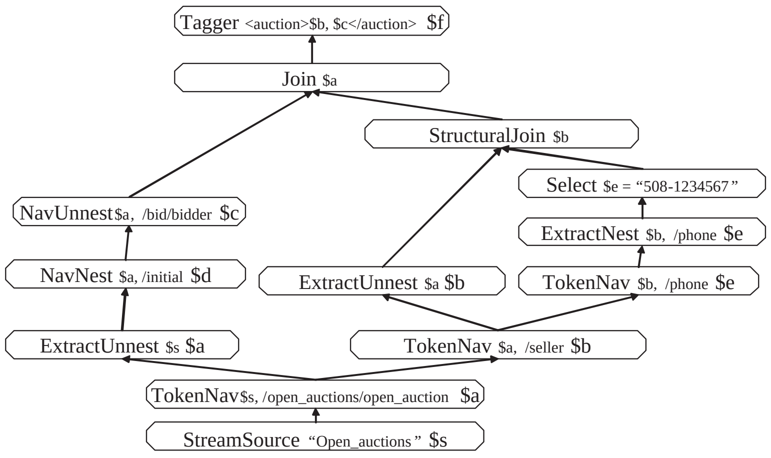

Tagger <auction>$b, $c</auction> $f
Join $a
StructuralJoin $b
Select $e = "508-1234567"
NavUnnest$a, /bid/bidder $c
ExtractNest $b, /phone $e
NavNest $a, /initial $d
ExtractUnnest $a $b
TokenNav $b, /phone $e
ExtractUnnest $s $a
TokenNav $a, /seller $b
TokenNav$s, /open_auctions/open_auction $a
StreamSource "Open_auctions" $s

Figure 2.10: Plan Rewritten from Figure 2.9: Pattern Retrieval on Token-or-Node Mode Change Rule Applied on $NavNest_{\$b,/phone}\$e$

If we continue to apply the mode change rewrite rules on every $NodeNav$ operators in Figure 2.10, we will finally get a plan shown in Figure 2.3. In this plan, the only regular tuple-based operator is $Tagger_{<auction>\$b,\$c</auction>}$. This operator does not consume $\$a$. Therefore we can apply the "extract elimination" rule on $ExtractUnnest_{\$s}\$a$ to eliminate it.

We can see that by applying different rewrite rules, we can end up with plans with different amounts of pattern retrieval performed on the tokens. For example, in the plan depicted in Figure 2.5, only one pattern, i.e., $\$s/open\_auctions/open\_auction$, is retrieved on the tokens. In the plan depicted in Figure 2.10, two more pattern, i.e., $\$a/seller$ and $\$b/phonel$, are retrieved on the tokens. We show later that the different amount of computations in the automata can have a major impact on the performance.

2.5 Implementation Strategies for Token-Related Operators

In this section, we present the stream physical level, i.e., the implementation for the stream logical operators. Since the implementation for the regular tuple-based operators can reuse the one developed for the static context in a pipelining style (i.e., operate on each input tuple rather than the whole input), we omit their discussion here. Instead, we focus on the implementation for the token-related operators which have no counterpart in the static context. A logical operator may have several physical implementations. Our purpose here is not to enumerate every possible alternative, but instead to show one solid base implementation for each of the operators. Clearly, there is room in the future to refine the proposed techniques or introduce other alternatives.

2.5.1 Implementation of TokenNav

A *StreamSource* operator can be viewed as a special *TokenNav* operator which locates the root element in the stream. Therefore we discuss *StreamSource* together with the *TokenNav* operator.

Using Automata for Path Recognition.

We use automata to recognize the path expressions on token streams. Figure 2.11 (a) shows such an automaton for the plan in Figure 2.3. The automaton is composed of several smaller automata, each corresponding to a different *TokenNav* operator in the plan. Each final state (shown as a state with double circles) corresponds to the end of a path in a *TokenNav* operator.

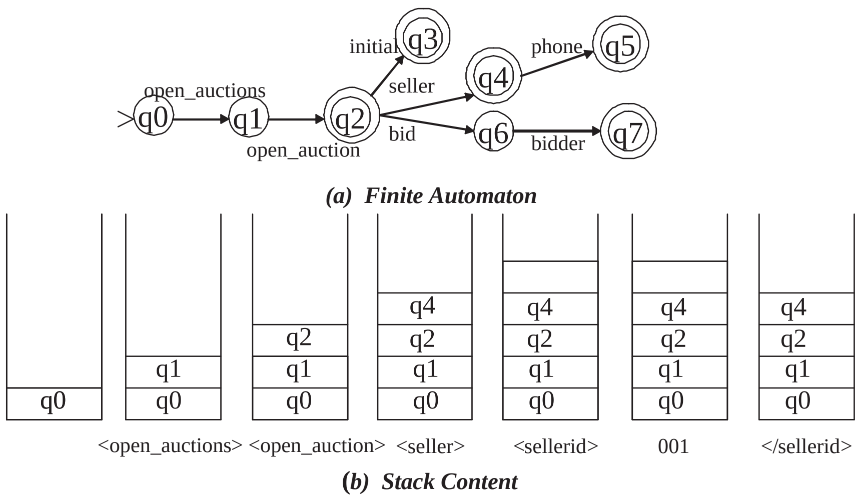


Figure 2.11: Implementation of *StreamSource*/*TokenNav*

A stack [80, 42] stores the history of the state transitions. Figure 2.11 (b) shows the snapshot of the stack after each token has been processed. The stack contains instances of the states. Initially, the stack contains only the instance of the start state q_0. Each incoming start tag is looked up in the transition entries of each state instance at the stack top. For any state that is transitioned to, we push its instance onto the stack. If no transition is found, we push an empty set. In our example, this would be the case when $<sellerid>$ is processed. When an end tag is encountered, the state instances at the stack top are popped off; thus the stack is restored to the status before its matching start tag had been processed. For a PCDATA item, no change is made to the stack.

Synchronization of Automaton with Token-Related Operators

The output tuples of *TokenNav* described in Section 2.3 are only logical concepts. At the physical level, no XAT tuples are actually output by $TokenNav$. Output of $TokenNav_{\$col1,path}\$col2$ includes (1) token value, (2) information needed for grouping the tokens that are the components of the same XML node (i.e., identifiers of $\$col2$), and (3) information needed for grouping XML nodes that are subelements of the same node (i.e., identifiers of $\$col1$). The semantics of $TokenNav$'s output expected by its downstream token-related operators are captured by triggering the corresponding downstream operators when certain automaton events happen.

Algorithm 1 illustrates the automaton behavior. *storeMgr* in automaton stores the data extracted from the stream. *storingCounter* maintains the number of extract operators that request to store the token currently being processed. A token may be requested by multiple extract operators to be stored. For example, suppose a query

Algorithm 1 Pseudocode of Automaton

```
public class Automaton {
int storingCounter;
StorageManager storeMgr;
void handleStartTag(Token startTag){
for each state on top of stack do
    state.transit(startTag);
end for
for each state pushed onto stack do
    if state is associated with extract operator then
        storingCounter++;
    end if
end for
if (storingCounter $>$ 0) then
    storeMgr.store(startTag);
end if
for each state on top of stack do
    trigger corresponding operators;
end for
}

void handleEndTag(Token endTag){
pop out all states at stack top;
if (storingCounter $>$ 0) then
    storeMgr.store(endTag);
end if
for each state popped off do
    if state is associated with extract operator then
        storingCounter $- -$;
    end if
end for
for each state popped off do
    trigger corresponding operators;
end for
}

void handlePCData(Token pcdata){
if (storingCounter $>$ 0) then
    storeMgr.store(pcdata);
end if
}
...
}
```

Automaton Event	Operators Triggered
Instance of q_n pushed onto stack	$ExtractUnnest_{\$col1}\$col2$, $ExtractNest_{\$col1}\$col2$
Instance of q_n popped off stack	$ExtractUnnest_{\$col1}\$col2$, $ExtractNest_{\$col1}\$col2$, $ExtractNest_{\$col2}\$col3$, $StructuralJoin_{\$col2}$

Table 2.4: Association between Automaton Events and Operators Triggered (q_n is a final state of $\$col2$ where $\$col2$ is an output variable of $TokenNav_{\$col1,path}\$col2$)

asks for returning both *open_auction* and its *seller* elements, then a component token of *seller* is requested to be stored by two extract operators, one for extracting *open_auction* and one for extracting *seller*.

The three methods, namely, *handleStartTag*, *handleEndTag* and *handlePCData*, describe the process of handling a start tag, an end tag and a PCDATA item respectively. For example, in *handleStartTag*, the processing takes three steps. First, the automaton performs the state transitions and pushes state instances onto the stack (lines 4 - 6). Second, the automaton computes whether the current token needs to be stored: if yes, the token is put into the storage manager (lines 12 to 14). Third, the operators associated with the state instances on the stack top are invoked (lines 15 - 17).

In *handleEndTag*, the processing takes similar three steps. First, the automaton backtracks its stack (line 20). Second, the token is stored if needed (lines 21 - 22) and the *storingCounter* is maintained (lines 24 - 27). Third, the automaton invokes the operators associated with the states that are just popped off (lines 29 - 31).

The method *handlePCData* is straightforward. The automaton does not trigger any stack transitions nor operators. It simply stores the token if needed (lines 34 - 36).

In the rest of this section, we first review the properties of our automata. We then describe the implementations of the *Extract* and *StructuralJoin* operators to illustrate that indeed the association between automaton events and the execution of these operators achieves the expected semantics.

Property of Automaton Implementation.

Our automata are designed to satisfy the "exclusive reach" property whenever possible. This property is important for two reasons. First, it ensures the correctness of synchronizing the automaton events and the token-related operators (i.e., line 16 in *handleStartTag* and line 30 in *handleEndTag* in Algorithm 1). Second, it enables us to implement the structural join operator more efficiently than previous literature. This will be illustrated when we describe the implementation strategies of token-related operators in Sections 2.5.2 to 2.5.4.

Property 2 ***Final State Reached by Destination Node Only (Exclusive-Reach).*** *Given a $TokenNav_{\$col1,path}\$col2$ operator, the instance of a final state of $path$ can be only pushed onto the stack (resp. popped off the stack) when a start tag (resp. end tag) of the destination node $\$col2$ is encountered.*

An automaton must be carefully constructed in order to satisfy the "exclusive reach" property. For example, for the XQuery "*for $\$v$ in /a return $\$v$//b*", we will construct the automaton in Figure 2.12 (a) instead of the one in Figure 2.12 (b). In both figures, q_1 is the final state of path $/a$. The bottom parts of Figures 2.12 (a) and (b) show the stack contents as tokens $<a><c><b>$... are processed. In Figure 2.12 (a), q_1 is pushed onto the stack only by the token $<a>$. In Figure 2.12 (b), besides $<a>$, q_1 can also be pushed onto the stack by $<c>$ and $<b>$ which are not bindings of $\$v$.

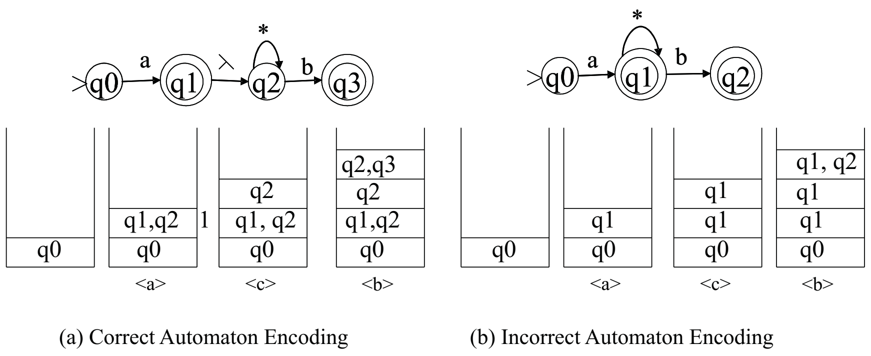

a
/
*
b
q0
q1
q2
q3
q1,q2
q0
q1, q2
q2,q3
<a>
<c>
<b>
(a) Correct Automaton Encoding
(b) Incorrect Automaton Encoding

Figure 2.12: Automaton Encoding for Paths Involving “//”

An XPath can be seen as a sequence of items where an item can be “/”, “//” or a navigation step. If we divide the sequence into two parts, we call the second part a *postfix* of the path.

Theorem 1 *If the “exclusive-reach” property holds, a final state can have at most one instance in the stack (we say the automaton is “final state duplicate free”) except in two circumstances: (1) if there is a* $TokenNav_{\$col1,path}\$col2$ *where* $path$ *contains a “//” and the data is recursive; and (2) if there is a* $TokenNav_{\$col1,path}\$col2$ *where a postfix of* $path$ *is a “//” followed by zero or more “*”.*

The proof of the theorem can be found in Appendix A. Figures 2.13 (a) and (b) illustrate circumstances (1) and (2) in Theorem 4 respectively. In Figure 2.13 (a), the automaton encodes $\$v//a$. Given a recursive XML token stream, e.g., <a><a></a></a>..., two instances of final state q_2 appear in the stack when the second <a> is processed. In Figure 2.13 (b), the automaton encodes $\$v/a//$. Even if the XML stream is not recursive, there can still be two instances of final state q_1 in the stack since the start tags of any descendant of $\$v/a$ push q_1 into the stack.

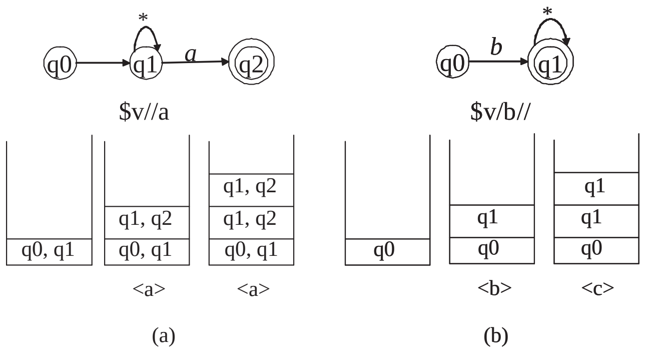


Figure 2.13: final state duplicates

When the automaton is ensured to be "final state duplicate free", we can efficiently implement the operators necessary to apply the $Within(t_1,\ t_2)$ and t_1 = t_2 boolean functions introduced in Section 2.3, i.e., functions for testing component or equivalence relationships between tokens. When final state duplicates may exist, our implementations of these two functions are similar to existing techniques in [42, 80]. Therefore in the following sections, i.e., Sections 2.5.2, 2.5.3 and 2.5.4, we focus on the circumstances where the automata are final state duplicate free because the corresponding implementations are distinguished from (and more efficient than) those in the other systems [42, 80]. We briefly describe our implementation when automata are not final state duplicate free in Section 2.5.5.

2.5.2 Implementation of ExtractUnnest

At the logical level, an $ExtractUnnest_{\$col1}\$col2$ operator consumes outputs from a $TokenNav_{\$col1,path}\$col2$ operator. This producer-consumer relationship is captured by the association of the final state q_n of $path$ with $ExtractUnnest$. $ExtractUnnest$ is invoked twice, both when q_n is pushed onto (line 8 in *handleStartTag* in Algorithm 1) and later when it is popped off (line 7 in *handleEndTag* in Algorithm 1) the stack. The two invocation processes are described below:

1). When an instance of q_n is pushed onto the stack, $ExtractUnnest_{\$col1}\$col2$ is invoked (line 16 in Algorithm 1). From the "exclusive-reach" property we know a start tag of $\$col2$ has been encountered. This $ExtractUnnest$ prepares a new XAT tuple. This tuple contains only one cell which is a placeholder of bindings of $\$col2$.

2). When an instance of q_n is popped off the stack, an end tag of $\$col2$ has been encountered. $ExtractUnnest_{\$col1}\$col2$ is invoked again (line 31 in Algorithm 1). A complete element node of $\$col2$ is added into the corresponding placeholder. The XAT tuple is then complete and can be output.

2.5.3 Implementation of ExtractNest

$ExtractNest_{\$col1}\$col2$ is associated with q_n and q_0 where q_n and q_0 correspond to the end and the beginning of $path$ in $TokenNav_{\$col1,path}\$col2$:

1). When an instance of q_n is pushed onto the stack: if this is the first time an instance of q_n is pushed within a binding of $\$col1$, $ExtractNest$ creates a tuple with a placeholder. All the destination nodes located within the same $\$col1$ would be put into this placeholder.

2). When an instance of q_n is popped off, $ExtractNest$ adds the newly completed destination node to the placeholder.

3). When an instance of q_0 is popped off the stack, by Theorem 4 we know there cannot be another instance of q_0 in the stack. Therefore the placeholder contains only those destination nodes located within this binding of $\$col1$. Since $\$col1$ has been completely processed, $ExtractNest$ outputs the tuple.

Example 8 *Figure 2.14 depicts the stream physical plan in Figure 2.3 with* Token-Nav *operators replaced by an automaton. Figure 2.14 (b) shows the processing of token 7, i.e., <$phone$>. First, q_5 is pushed onto the stack. $ExtractNest_{\$b}\$e$ is invoked. It creates a tuple with one cell which will store the binding of $\$e$. Next, the storing counter is increased by 1. This non-zero storing counter indicates that token 7, namely, <$phone$> needs to be buffered (refer to* handleStartTag *method in Algorithm 1). Note, token 7 is not necessarily physically stored as a token. It can also be stored as a structure that is more convenient for later manipulation, such as a DOM-like tree structure if later on a node navigate operator is performed on $\$e$.*

Figure 2.14 (c) shows the processing of token 9, i.e., </$phone$>. q_5 is popped off. This leads to the decrease of the storing counter to 0. $ExtractNest_{\$b}\$e$ is again invoked. The reference to the $phone$ element in the storage manager is passed to the placeholder. The dashed line in the placeholder indicates that the placeholder is "open", in other words, there may be more $phone$ elements that could still be located within the same seller.

The cell is "closed" in Figure 2.14 (d) when token 13 is processed. $ExtractNest_{\$b}\$e$ is informed that the binding of $\$e$ is now complete and the tuple is ready for output.

2.5.4 Implementation of StructuralJoin

A $StructuralJoin_{\$col1}$ operator must have an upstream operator in the form of $TokenNav_{\$col0,path}\$col1$. This $StructuralJoin$ is invoked when an instance of

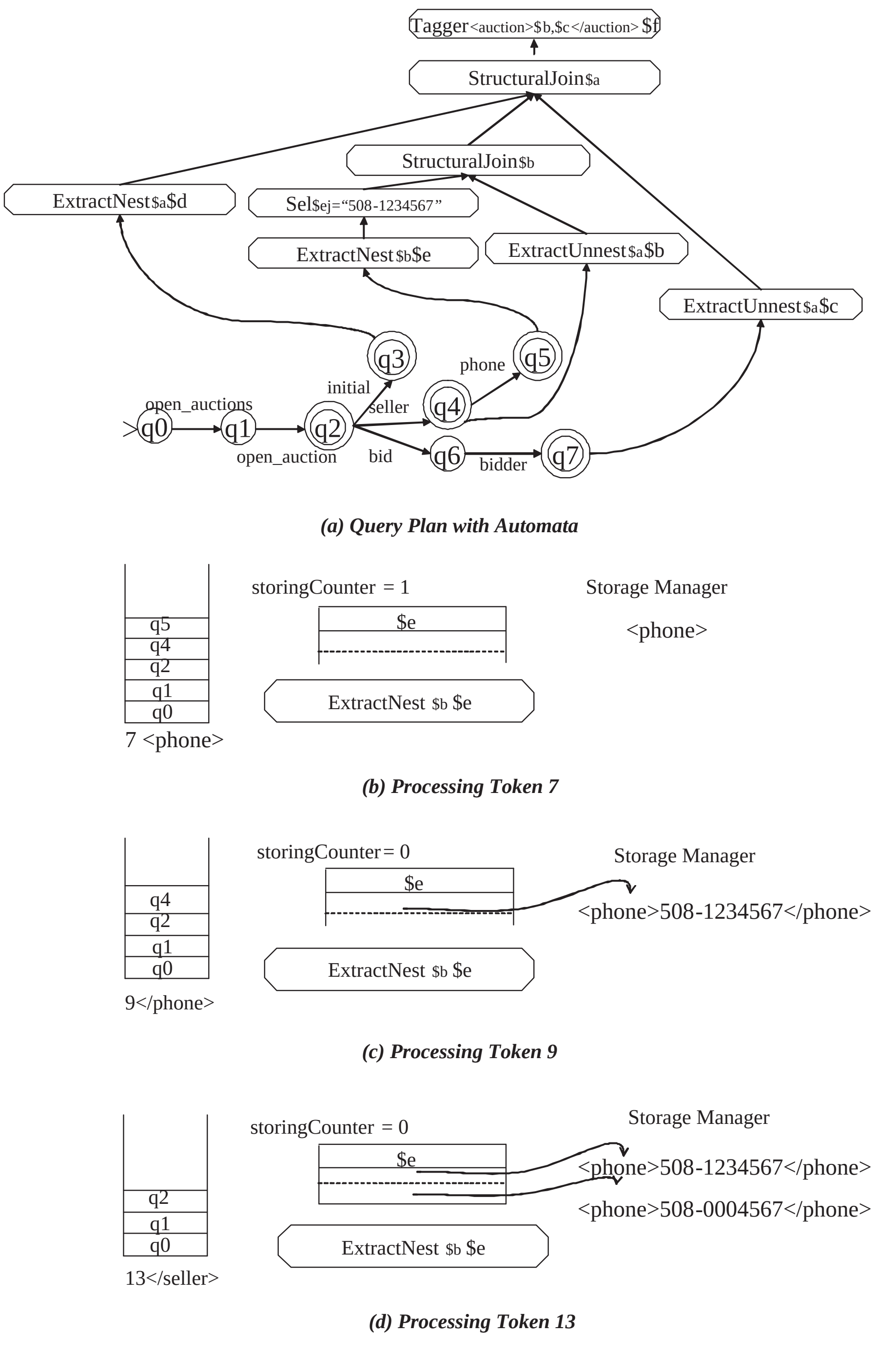


Figure 2.14: Invoking $ExtractNest$ Operator

q_n, the state that corresponds to the end of $path$, is popped off the stack. The input tuples to $StructuralJoin$ contain only elements located within this binding of $\$col1$. Therefore *StructuralJoin* can simply perform a Cartesian product on its input tuples. The input tuples are purged after the Cartesian product so that they would not participate in the next Cartesian product for a different binding of $\$col1$. Since our structural join must be invoked when a certain automaton event happens, we call it an *in-time structural join*.

2.5.5 Implementations in Automata with Final-State Duplicates

In the two circumstances when automata have final-state duplicates as described in Theorem 4, there can be more than one final state in the stack. We describe the modification to the above implementations for *Extract* and *StructuralJoin* respectively.

Extract

Given a $Extract_{\$col1}\$col2$, suppose q_0 and q_n are the final states of $\$col1$ and $\$col2$ respectively. When the automaton are not final state duplicate free, two modifications have to be made to the above implementations of $Extract$. The first modification addresses the situation that multiple instances of q_n may exist in the stack. When an instance of q_n is popped off, we now have to identify which element node has been completed. For example, in Figure 2.13 (a), when a q_2 is popped off due to a </a>, we need to know whether this </a> matches the first <a> or the second <a>. This can be achieved by simply maintaining the number of final states pushed onto or popped off the stack. Once we know which element node is completed, we then know in which placeholder in the XAT tuples to add this element node.

The second modification addresses the possibility that multiple instances of q_0 may exist in the stack. For an instance of q_n in the stack, we have to identify it is transitioned from which instance of q_0. This information is necessary since we may perform a $StructuralJoin_{\$col1}$ later on the output of $Extract_{\$col1}\$col2$. Each output tuple of $Extract_{\$col1}\$col2$ not only contains an element node of $\$col2$ but also an identifier of $\$col1$.

StructuralJoin

Given a $StructuralJoin_{\$col1}$, since its input tuples now carry the identifiers of $\$col1$, it will perform joins over $\$col1$. Such a $StructuralJoin$ is similar to that developed in Tukwila [42] and YFilter [80]. We call it an *identifier-based* structural join.

2.5.6 Comparison between In-time and Identifier-based StructuralJoins

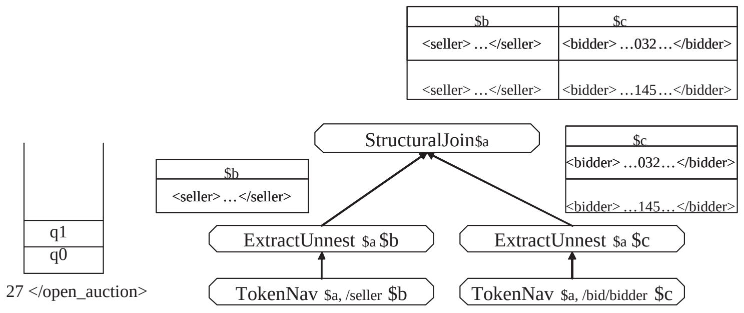
Raindrop only chooses an identifier-based physical implementation for a structural join in the "final state duplicate existing" circumstances in Theorem 4. In all other circumstances, an in-time physical implementation is chosen. In contrast, both Tukwila and YFilter provide only the identifier-based structural join implementations. We now compare the in-time and identifier-based implementations in the final state duplicate free environment.

In Figure 2.15 (a) which depicts an in-time structural join, each input tuple has only one cell, containing *seller* or *bidder*. In contrast, in Figure 2.15 (b) which depicts an identifier-based structural join, an input tuple to $StructuralJoin_{\$a}$[2] must

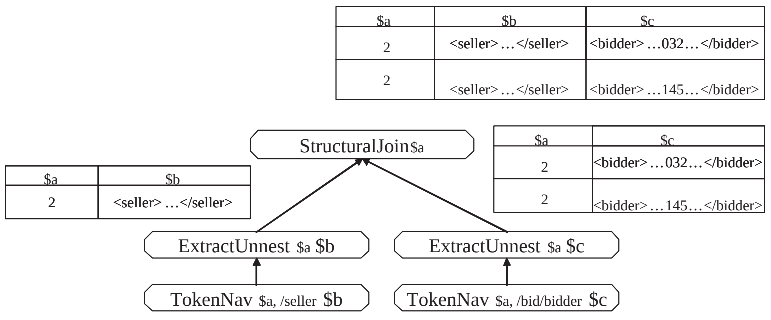
[2][42] does not have an explicit, separate structural join operator since automaton computations are

contain an identifier for $\$a$, i.e., the *open_auction* element[3]. $StructuralJoin_{\$a}$ in Figure 2.15 (b) joins the input tuples on the identifiers of *open_auction*.

Clearly, the in-time structural join is more efficient than identifier-based structural join. First, in-time structural join takes less memory since the size of an input tuple is smaller than that in the identifier-based structural join. Second, in-time structural join does not perform any value comparison.



(a) In Time Structural Join



(b) Identifier -based Structural Join

Figure 2.15: Comparing In-time Structural Join and Identifier-based Structural Join

expressed in one mega operator as mentioned in Section 1.1. However structural joins are performed within this mega operator.

[3]The structural join in [80] would even contain an identifier for each navigation step on the path, for example, a right input tuple in Figure 2.15 (b) would also contain an identifier for the *bid* element.

2.6 Programming Model for Synchronizing the Execution of Operators

In a traditional query plan, synchronization among operators is usually achieved by the *iterator* mode [35], namely, an operator is always invoked by its immediate downstream operator. However, only using this model does not meet the needs of a *Raindrop* plan. First, execution of $TokenNav$ operators, i.e., the automaton, must be data-driven. Given two $TokenNav$ operators such as $TokenNav_{\$a,/seller}\$b$ and $TokenNav_{\$a,/bid/bidder}\$c$, whether the bindings of $\$b$ or the bindings of $\$c$ will be retrieved first is completely decided by the data. Second, to ensure the correctness, $Extract$ and $StructualJoin$ operators have to be invoked at a certain time. For example, the $StructuralJoin_{\$a}$ operator must be invoked by its ancestor upstream operator $TokenNav_{\$s,/open\_auctions/open\_acution}\$a$ when an end tag of a binding of $\$a$ is encountered.

We propose to support three invocation modes in *Raindrop*. For each mode, we describe (1) what operators can be invoked in this mode; (2) when these operators are invoked in this mode; and (3) why the operators are invoked in this mode.

2.6.1 AncestorUpstreamDriven Mode

If an operator is invoked by its ancestor upstream operator, we say this operator is invoked in the *AncestorUpStreamDriven* mode.

Operators that can be invoked in this mode: An operator in the format of $ExtractNest_{\$col1}\$col2$ or $StructuralJoin_{\$col1}$ can by invoked by its ancestor upstream operator in the format of $TokenNav_{\$col0,p}\$col1$.

When invoked in this mode: an end tag of a binding of $\$col1$ is encountered.

Why invoked in this mode: Both $ExtractNest_{\$col1}\$col2$ and $StructuralJoin_{\$col1}$ must be informed right at the point when an end tag of a binding of $\$col1$ is encountered. $ExtractNest_{\$col1}\$col2$ can then output a newly-formed tuple and $StructuralJoin_{\$col1}$ can perform cartesian products on its input. Assume $\$col1$ = $\$col0$/p, then $ExtractNest_{\$col1}\$col2$ and $StructuralJoin_{\$col1}$ must be invoked by $TokenNav_{\$col0,p}\$col1$ when $TokenNav_{\$col0,p}\$col1$ detects the finish of a binding of $\$col1$.

Example 9 *Algorithm 2 shows the pseudocode of an $ExtractNest$ operator. The $ExtractNest$ operator implements an* ancestorUpstreamDriven *method. In this method, $ExtractNest$ outputs the tuple that is newly formed.*

Algorithm 2 Programming Model for ExtractNest

```
public class ExtractNest {
 1: Boolean isImmediateUpstreamDriven;
 2: List[ ] inputQueues;
 3: List outputQueue;
    ...
 4: public void ancestorUpstreamDriven(){
    //perform process (3) in Section 2.5.3;
 5: enqueue a tuple that is just completely formed into outputQueue;
 6: }
 7: public List downstreamDriven(){
 8: return outputQueue.dequeueAll();
 9: }
}
```

2.6.2 DownstreamDriven Mode

This *DownstreamDriven* mode is similar to the traditional iterator mode, namely, an operator is invoked by its immediate downstream operator.

Operators that can be invoked in this mode: any operator that is neither a *StreamSource* nor a *TokenNav* can be invoked by its downstream $StructuralJoin$ operator (if any).

When invoked in this mode: When a $StructuralJoin_{\$col1}$ is invoked in an *AncestorUpstreamDriven* mode, $StructuralJoin_{\$col1}$ invokes its upstream operators to generate output for it to consume. Each upstream operator, when invoked, recursively invokes its own upstream operators. From the perspective of the immediate and ancestor upstream operators of this $StructuralJoin$, they are invoked by their downstream operators.

Why invoked in this mode: When an end tag of a binding of $\$col1$ is finished, $StructuralJoin_{\$col1}$ is invoked. $StructuralJoin_{\$col1}$ must invoke its upstream operator to ensure they have all processed the current binding of $\$col1$. Otherwise, $StructuralJoin_{\$col1}$ does not have input to consume.

For example, in Figure 2.3, when an $</seller>$ is encountered, $StructuralJoin_{\$b}$ is invoked. $ExtractNest_{\$b}\$e$ has finished the processing of the current $seller$ element, i.e., it has extracted $phone/text()$ within this $seller$. Now, the operator $Sel_{\$e=\text{“}508-1234567\text{”}}$ must be invoked to consume the output of $ExtractNest_{\$b}\$e$ to generate the input to $StructuralJoin_{\$b}$.

Example 10 *Algorithm 3 shows the pseudocode of* $StructuralJoin$. $StructuralJoin$ *implements both* ancestorUpstreamDriven *and* downstreamDriven *methods. Each time when a </seller> is encountered, line 30 in* handleEndTag *in Figure 1 will call* *the* ancestorUpstreamDriven *method of* $StructuralJoin_{\$b}$*. This method then calls the* downstreamDriven *methods of both* $Sel_{\$e=\text{“}508-1234567\text{”}}$ *and* $ExtractUnnest_{\$a}\$b$ *(lines 8 - 11 in Algorithm 3).* $Sel_{\$e=\text{“}508-1234567\text{”}}$ *again invokes the* downstream-Driven *method of* $ExtractNest_{\$b}\$e$*. At this time,* $ExtractNest_{\$b}\$e$ *simply returns all tuples generated within this just-finished* $seller$ *element (see line 8 in Algorithm 2). Eventually,* $StructuralJoin$ *consumes the output tuples of its upstream operators (line 12 in Algorithm 3).*

Synchronization of Operators Invoked in AncestorUpstreamDriven and DownstreamDriven Modes

When an end tag of a binding of $\$col1$ is encountered, both $ExtractNest_{\$col1}\$col2$ and $StructuralJoin_{\$col1}$ must be invoked in the *AncestorUpstreamDriven* mode. However $ExtractNest_{\$col1}\$col2$ must be invoked in *AncesetorUpstreamDriven* mode before $StructuralJoin_{\$col1}$ is invoked in *AncesetorUpstreamDriven* mode. Only in this way, the operators invoked in the *AncestorUpstreamDriven* mode can work correctly with the operators invoked in the *DownStreamDriven* mode.

For example, when a $</seller>$ is encountered, line 30 in Algorithm 1 calls the *ancestorUpstreamDriven* method of $ExtractNest_{\$b}\$e$ first and the *ancestorUpstreamDriven* method of $StructuralJoin_{\$b}$ next. When the *ancestorUpstreamDriven* method of $ExtractNest_{\$b}\$e$ is called, $ExtractNest_{\$b}\$e$ puts the tuple that is generated within the current $seller$ element into its output queue. Next, the *ancestorUpstreamDriven* method of $StructuralJoin_{\$b}$ is called. This *ancestorUpstreamDriven* method calls the *downstreamDriven* method of $ExtractNest_{\$b}\$e$ (line 10 in Algorithm 3). This *downstreamDriven* method (lines 7 - 9 in Algorithm 2) then returns the tuple that is generated by the *ancestorUpstreamDriven* method

Algorithm 3 Programming Model for In-Time Structural Join

```
public class StructuralJoin {
Boolean isImmediateUpstreamDriven;
List[ ] inputQueues;
List outputQueue;
...

//when the operator is invoked by its ancestor upstream operator
public ancestorUpstreamDriven(){
List outputTuples;
List[] inputTuples;
$n$ = number of upstream operator of this StructuralJoin.
for int i = 1; i $\leq n$; $i$++ do
   let $upstreamOp$ denotes the $i^{th}$ upstream operator;
   $inputTuples[i]$ = $upstreamOp$.downstreamDrive();
end for
outputTuples = join $inpuTuples[1]$, $inputTuples[1]$, ..., and $inputTuples[n]$;
if outTuples are not empty then
   for each downstream operator downStreamOp of this StructuralJoin do
      if downStreamOp.isImmediateUpstreamDriven then
         downStreamOp.immediateUpstreamDrive(outputTuples);
      else
         outputQueue.enqueue(outputTuples);
      end if
   end for
end if
}

//when the operator is required by its downstream operator to run
public List downstreamDriven(){
returnoutputQueue.dequeueAll();
}
}
```

of $ExtractNest_{\$b}\$e$.

2.6.3 ImmediateUpStreamDriven

The *ImmediateUpStreamDriven* mode is also called *data driven*. An operator is said to be invoked in the *ImmediateUpStreamDriven* mode if it is invoked by its immediate upstream operator.

Operators that can be invoked in this mode: regular tuple-based operators that do not have a $StructuralJoin$ in its downstream.

When invoked in this mode: As the name *data driven* suggests, the operator is invoked once its immediate upstream operator generates output.

Why invoked in this mode: If an operator does not have a $StructuralJoin$ in its downstream, e.g., $Tagger_{<auction>\$b,\$c</auction>}\$f$ in Figure 2.3, it will not be invoked in a downstream driven method. We thus design the *ImmdidateUpStream-Driven* mode so that such an operator op is invoked by its immediate upstream operator $upstreamOp$ once $upstreamOp$ has generated output for op to consume.

Example 11 *Algorithm 4 shows the pseudocode of $Select$. $Select$ implements an* immediateUpstreamDriven *method. The $Select$ operator first consumes input tuples from its upstream operator (line 5). If this $Select$ operator does not have downstream $StructuralJoin$, then its downstream operators must not have downstream $StructuralJoin$ as well. That is to say, the downstream operators of $Select$ must also be invoked in the* immediateUpstreamDriven *mode. Therefore, when this $Select$ has generated output, it invokes its downstream operator to consume its output (lines 6 - 9).*

Algorithm 4 Programming Model for *Select*

```
public class Select {
1:  Boolean isImmediateUpstreamDriven;
2:  List[ ] inputQueues;
3:  List outputQueue;
    ...
4:  public void immediateUpstreamDriven(List inputTuples){
5:  List outputTuples = selection predicate evaluation on inputTuples;
6:  if outputTuples are not empty then
7:      let $downstreamOp$ denotes the downstream operator of this $Select$;
8:      downstreamOp.immediateUpstreamDriven(outputTuples);
9:  end if
10: }
}
```

Operator	Invocation Modes Operator Supports
ExtractUnnest	DownstreamDriven
ExtractNest	DownstreamDriven, AncestorUpstreamDriven
StructuralJoin	DownstreamDriven, AncestorUpstreamDriven
Regular tuple-based Operators	DownstreamDriven, ImmediateUpstreamDriven

Table 2.5: Operators and the Invocation Modes They Support

2.6.4 Summary

We define a programming model which supports multiple invocation modes. Different operators can be invoked in different modes. Even one single operator can be invoked in different modes. Table 2.5 summarizes what operators can be invoked in which modes.

All modes are defined in a modular manner. This can be a significant advantage when a flexible configuration of the synchronization among operators is needed. For example, in Figure 2.14 (a), suppose a rewrite rule switches $Select_{\$e=\text{“}508-1234567\text{”}}$

with $StructuralJoin_{\$b}$ and $StructuralJoin_{\$a}$ so that $Select_{\$e=\text{“}508-1234567\text{”}}$ ends up as a downstream operator of $StructuralJoin_{\$a}$. Since $Select_{\$e=\text{“}508-1234567\text{”}}$ now has no downstream $StrcuturalJoin$ operator, it has to be invoked in an *ImmediateUpstreamDriven* manner instead of the *DownStreamDriven* manner as before. This can be simply achieved by setting the Boolean value of “isImmediateUpstreamDriven” (see line 1 in Figure 4) to true. Therefore any output from $StructuralJoin_{\$a}$ will be immediately sent to $Select_{\$e=\text{“}508-1234567\text{”}}$ for consumption (see lines 15 - 16 in Algorithm 3).

2.7 Experiments

We have implemented a prototype of *Raindrop* [38] with Java 1.4.1. We use ToXGene [24], an XML data generator, to generate the XML documents. We ran experiments on two Pentium III 800 Mhz machines with 512MB memory each. One machine sends XML token streams via sockets to another machine which would then process the received data. The execution time we report does not include the network transmission time. The experiments reported in this section focus on showing the performance differences among the plans with different amounts of computation pushed into the automata.

The cost of query execution consists of two parts: one for buffering the data, and the other for manipulating (e.g., filtering or restructuring) the buffered data. The queries we test can be divided into two categories. For a query in the first category, all of its candidate plans buffer the same amount of data. Therefore the performance differences among candidate plans only result from the differences in the data manipulation costs. For a query in the second category, some candidate plans trade buffering costs for manipulation costs, i.e., buffering more data in the hopes that later manipulation may be accelerated. The performance differences among the candidate plans then show the tradeoff between the two costs.

2.7.1 Testing Queries Having Alternative Plans with Same Buffering Cost

Figure 2.16 shows an example query template from an XML benchmark, XMark [7]. This query asks to return any *bidder* element that satisfies a set of filters where each filter is a linear XPath, i.e., XPath with no filters. Note in any alternative plan, a *bidder* element always has to be buffered because it may appear in the final answer. Therefore all plans have the same buffering costs.

```
for $a in  stream( “open_auctions ”)/open_auctions /
      open_auction/bidder[filter1][filter2]...[   filtern ]
return
   <auction>{$a}</auction>
```

Figure 2.16: Query with Filters

We now analyze which factors may lead to performance differences among candidate plans. Suppose $filter_1$ has a low selectivity, i.e., it is rarely satisfied, then we should evaluate this filter before the other filters. This follows the classical “push down operators of low selectivity” optimization technique. However, in the automata, all filter patterns are retrieved in parallel. In order to assure that the other filters are evaluated after $filter_1$ is evaluated, we must leave the other filters out of the automata.

Let us consider the opposite case where all filters are frequently satisfied. A plan that pushes all filters into the automata (called *maximal-navigation-pushdown* plan) can evaluate all filters in one single access of the tokens. The other plans, however, have to access the $bidder$ elements n times if there are n filters to be evaluated out of the automata. Therefore a maximal-navigation-pushdown plan should outperform the other plans.

Testing on Cheap Filters

We perform a series of experiments to confirm our analysis. In the first experimentation, we vary the selectivity of $filter_1$ in the data set. The selectivity of all the other filters is 100%. The length of each filter is 1, i.e., each filter has only one deterministic navigation step and does not have any descendant axis (e.g., $/bidderid$). The cost of evaluating such a filter is rather cheap. The average width of ***bidder***, i.e., the number of its children elements, is set to 30.

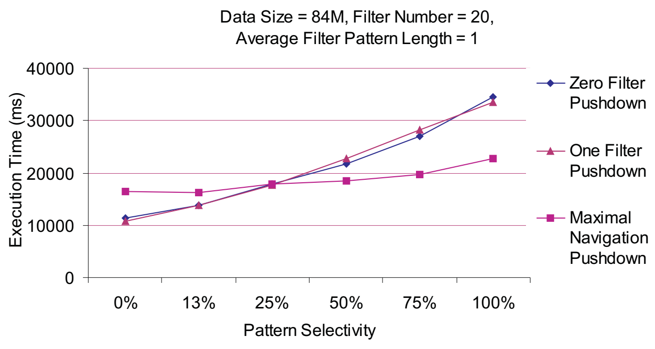


Figure 2.17: Performance of Alternative Plans for Queries with 20 Filters of Average Length 1

We test three plans. In the zero-filter-pushdown plan, all filters are evaluated out of the automata but $filter_1$ is evaluated before any other filters. In the one-filter-pushdown plan, only $filter_1$ is evaluated in the automata. The evaluation or-

Pattern Selectivity	0%	12%	25%	50%	100%
Query with 5 filters	1.06	1.02	1.03	0.95	0.90
Query with 10 filters	1.23	1.07	0.85	0.98	0.75
Query with 20 filters	1.47	1.18	1.00	0.85	0.65

Figure 2.18: Ratio of Execution Time of Maximal Pushdown with Execution Time of Zero Filter Pushdown for Queries with Different Numbers of Filters

der of the other filters does not matter here since they have the same selectivity. The third plan we test is the maximal-navigation-pushdown plan. It is seen from Figure 2.17 that at the lower end of selectivity (0% - 25%), the zero-filter-pushdown plan performs better than the maximal-navigation-pushdown plan. At the higher end of selectivity (25% - 100%), the maximal-navigation-pushdown plan performs better than the zero-filter-pushdown plan. At all times, the zero-filter-pushdown plan behaves similarly to the one-filter-pushdown plan. That is to say, evaluating a single pattern on tokens has a similar performance as evaluating the pattern on element nodes. Therefore in the following experimentations, we only illustrate one of these two plans.

Figure 2.18 further compares the performance of different queries. All queries conform to the query template in Figure 2.16 but differ in the number of filters. The ratio of the execution time of maximal-navigation-pushdown with that of the zero-filter-pushdown plan is reported.

The purpose of Figure 2.18 is to show that measures are needed to judge whether it is worthwhile to consider alternative plans. For a simple query, such as the one with 5 filters, both the zero-filter-pushdown and maximal-navigation-pushdown plans always perform similarly (the ratio is close to 1) when the selectivity of $filter_1$ varies. As the query gets more complicated, i.e., the number of filters increases, the differences among alternative plans get more significant.

Testing on Expensive Filters

We now test two queries with more expensive filters. In the first query, $filter_1$ still has a length of 1 but all other filters are longer. Correspondingly, savings from the evaluation on a $filter_i$ ($i \neq 1$) are larger than those in Figure 2.17. Figure 2.19 gives the experimental result. When the selectivity of $filter_1$ is 0%, the ratio of the execution time of a maximal-navigation-pushdown plan with that of zero-filter-pushdown plan can reach 1.46. This is the same as that in a query with 20 shorter filter patterns (refer to the first cell in third row in Figure 2.18). Also, the crossover between the two plans shifts from 25% in Figure 2.17 to 50% in Figure 2.19. In other words, the zero-filter-pushdown plan is more likely to win over the maximal-navigation-push down plan compared to the scenarios in Section 2.7.1.

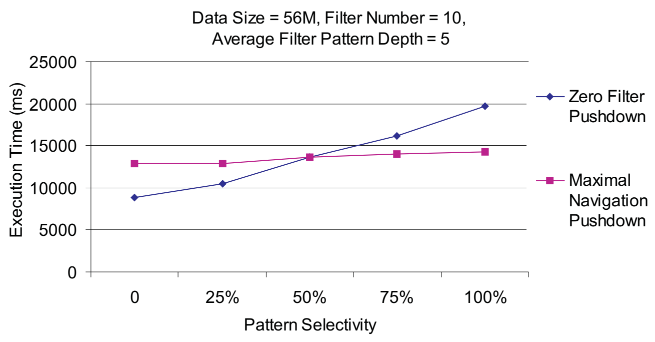


Figure 2.19: Performance of Alternative Plans for Queries with 10 Filters of Average Length 5

The second query we test has only two filters. $filter_1$ still has a length of 1 but $filter_2$ starts with a "//". In the automata, "//" is encoded as a self-cycle on a state (refer to Figure 2.12). Any component token of *bidder* will lead to an automata state transition. Computing such a filter is more expensive than a filter with only deterministic navigation steps, because to evaluate a filter with n deterministic navigation steps, component tokens of *bidder* that are more than n levels deep within $bidder$ would not induce any transitions. Figure 2.20 confirms that the performance difference among alternative plans of this query can be significant.

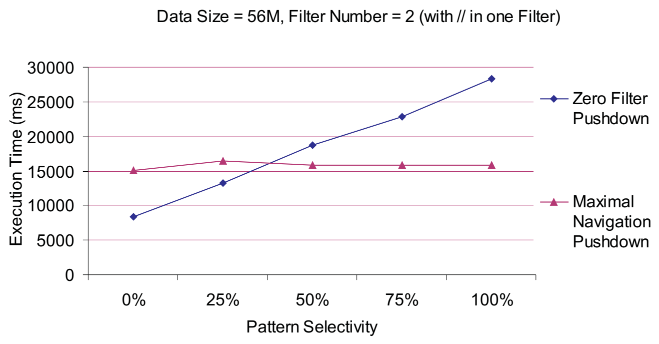


Figure 2.20: Performance of Alternative Plans for Queries with 2 Filters (One Filter has "//")

2.7.2 Testing Queries Having Alternative Plans with Different Buffering Costs

We now study a set of queries which conform to the template shown in Figure 2.21. This query pairs *seller* and certain *bidder* subelements that are located within the same *open_auction*. Figures 2.22, 2.23 and 2.24 show three alternative plans for this query, namely, pushing one, three, or all navigation operators down to the automata respectively. In the one-navigation-pushdown plan in Figure 2.22, each *open_auction* element has to be buffered since it will be navigated into later to find the $initial$, $seller$ and $bidder$ subelements. In contrast, both the three-navigation-pushdown and maximal-navigation-pushdown plans in Figures 2.23 and 2.24 buffer only a minimal amount of data, i.e., the *bidder* and *seller*, for later navigation or result construction.

```
for $a in stream(“open_auctions”)/open_auctions/
     open_auction/auction[initial],
     $b in $a/seller,
     $c in $a/bid/bidder[filter1][filter2] ...[filtern]
return
    <auction>{$b, $c}</auction>
```

Figure 2.21: Query with Multiple Bindings in For Clause

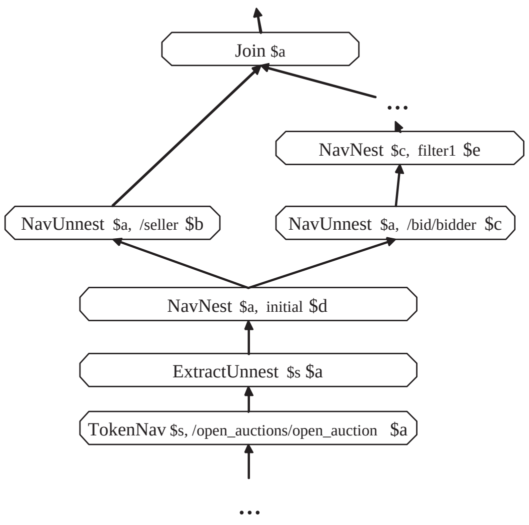


Figure 2.22: One Navigation Pushdown

We vary three factors in the data set. First, we vary the selectivity of the filter $/initial$ but keep the selectivity of all the other filters at 100%. Second, we vary the size of the data that are subelements of *open_auction* other than *seller* or *bidder*. We

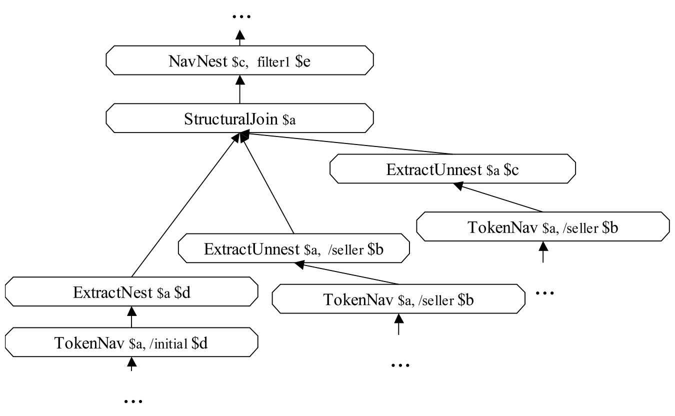


Figure 2.23: Three Navigation Pushdown

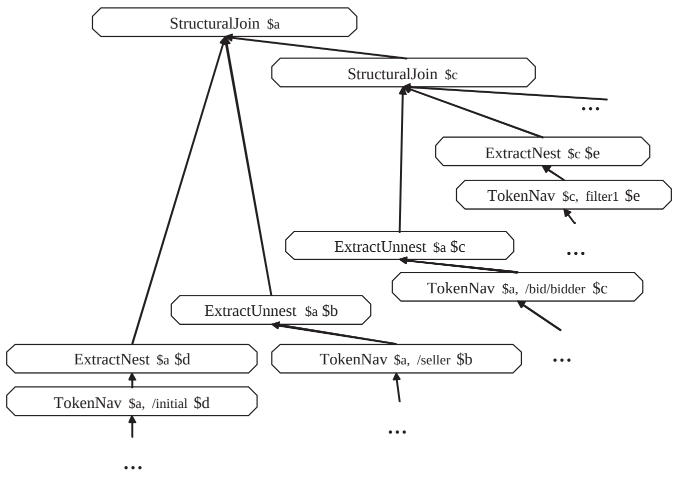


Figure 2.24: Maximal Navigation Pushdown

Data Set	extra buffering ratio%	average number of *seller*'s within an *open_auction*
Data Set 1	0%	1
Data Set 2	50%	1
Data Set 3	0%	10

Table 2.6: Data Characteristics of Three Data Sets

call the ratio K = (the size of the above data) / (the overall size of *seller* and *bidder*) an *extra buffering ratio*. Third, we vary the number of *seller* elements in each *open_auction*. We fix the average number of *bidder* elements in an *open_auction* to 20. We generated three data sets whose data characteristics are shown in Table 2.6.

Figures 2.25 and 2.26 show the results on the first two data sets. The X-axis shows the selectivity of $filter_1$. We make two observations from these two figures.

1). The three-navigation-pushdown plan is always better than the one-navigation-pushdown plan due to two reasons. First, three-navigation-pushdown plan never buffers more data than one-navigation-pushdown plan. In data set 2 where the extra buffering ratio is 50%, it buffers much less data. Second, the $Join_{\$a}$ operator in Figure 2.22 is an identifier-based join. It is more costly than the $StructuralJoin_{\$a}$ operator in Figure 2.23.

2). The crossover point of one-navigation-pushdown and maximal-navigation-pushdown plans occurs at a lower selectivity in Figure 2.26 than that in Figure 2.25. This is because in Figure 2.26, the cost that the one-navigation-pushdown plan saves in pattern retrieval is offset by the cost that the one-navigation-pushdown plan spends in buffering extra data.

Figure 2.27 reports the results on the third data set. The trend of the performance differences between one-navigation-pushdown and maximal-navigation-

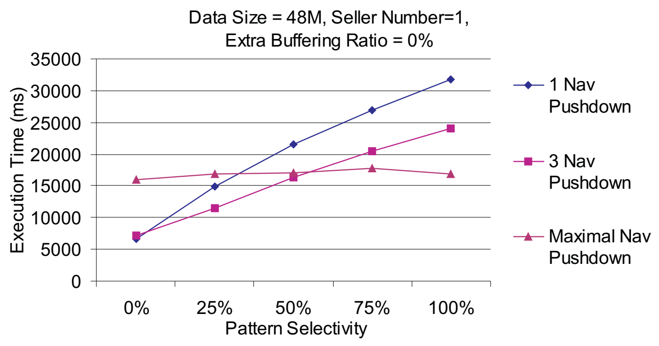


Figure 2.25: Performance on Data Set 1

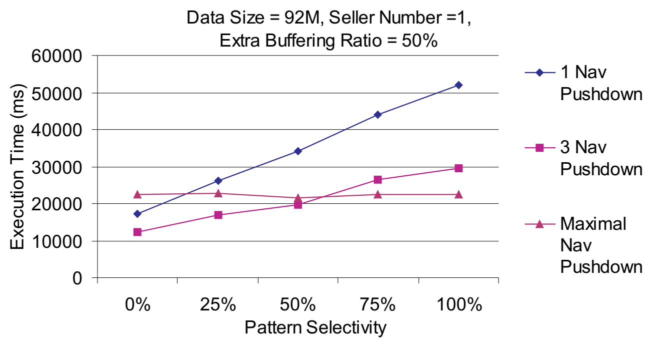


Figure 2.26: Performance on Data Set 2

pushdown plans remains similar to that in Figure 2.25. However three-navigation-pushdown performs extremely badly (its performance when the selectivity is larger than 25% is not shown due to extremely high cost). This is because in Figure 2.23, a *bidder* is paired with each *seller* by $StructuralJoin_{\$a}$. Therefore each *bidder* is duplicated 10 times since there are 10 *seller* elements within the same *open_auction*. Correspondingly, any downstream computation on a *bidder* element will be duplicated. For example, a single *bidder* element will be navigated into 10 times by $NavNest_{\$c,filter_1}\$e$ in Figure 2.23 to evaluate $filter_1$. In the other two plans, either $Join_{\$o}$ in Figure 2.22 or $StructuralJoin_{\$a}$ in Figure 2.24 is performed after locating all the patterns within *bidder* so that no navigation computation is duplicated.

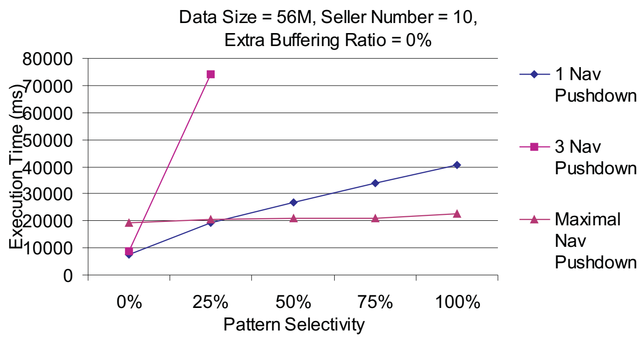


Figure 2.27: Performance on Data Set 3

Chapter 3

Runtime Plan Optimization: Switching between Automaton and Algebra Processing Styles

In the previous chapter, we have illustrated that the decisions regarding which patterns to be retrieved in the automaton or out of the automaton can have significant impact on the performance of query evaluation. In this chapter, we explore how to get a good plan taking advantage of this optimization opportunity.

3.1 Solution Space

We provide a set of rewrite rules in Raindrop. From an initial plan, by repeatedly applying the rewrite rules, we can get a batch of alternative plans that compose the search space. We now describe these rewrite rules. In this chapter, we use the query shown in Figure 3.1 as the running example. Figure 3.2 shows a plan, which retrieves all pattern in the automaton, for this query.

```
for $a in stream("open_auctions")/auctions/auction[reserve]
     $b in $a/seller, $c in $a/bidder
Where $b//profile contains  "frequent" and $c//zipcode = "01609"
return
   <auction> {$b, $c} </auction>
```

Figure 3.1: Example Query for Automaton-in-or-out Optimization

3.1.1 Token-or-Node Mode Change Rules

The **token-or-node mode change rules**, as described in Section 2.4, change the modes (i.e., on tokens or on nodes) of pattern retrieval. This is the key rewrite rule for generating alternative plans in our solution space. Since a pattern retrieval on tokens (resp. on nodes) is performed in the automaton (resp. out of the automaton), we also say this rule pulls pattern retrieval out of the automaton or pushes patterns retrieval into the automaton. For ease of reading, we recap these rules briefly.

Figures 3.3 and 3.4 show the token-or-node mode change rules in two circumstances. In Figure 3.3, no $StructuralJoin_{\$col1}$ exists in the top plan so that a $StructuralJoin_{\$col1}$ is introduced when $\$col2$ = $\$col1/path1$ is pushed down. In Figure 3.4, a $StructuralJoin_{\$col1}$ exists in the top plan so that no new $StrucutralJoin$ is introduced when $\$col3$ = $\$col1/path2$ is pushed down. Figure 3.5 further shows a rule that eliminates an unnecessary $Extract_{\$col0}\$col1$ operator when $\$col1$ is not consumed by any non-automaton operators.

An interesting feature of the mode change rules is that when we push a pattern retrieval, say $\$col2$ = $\$col1/path$, into the automaton, the resultant $TokenNav_{\$col1,path}\$col2$ can only be placed in one unique position, i.e., right on top of a $TokenNav$ that re-

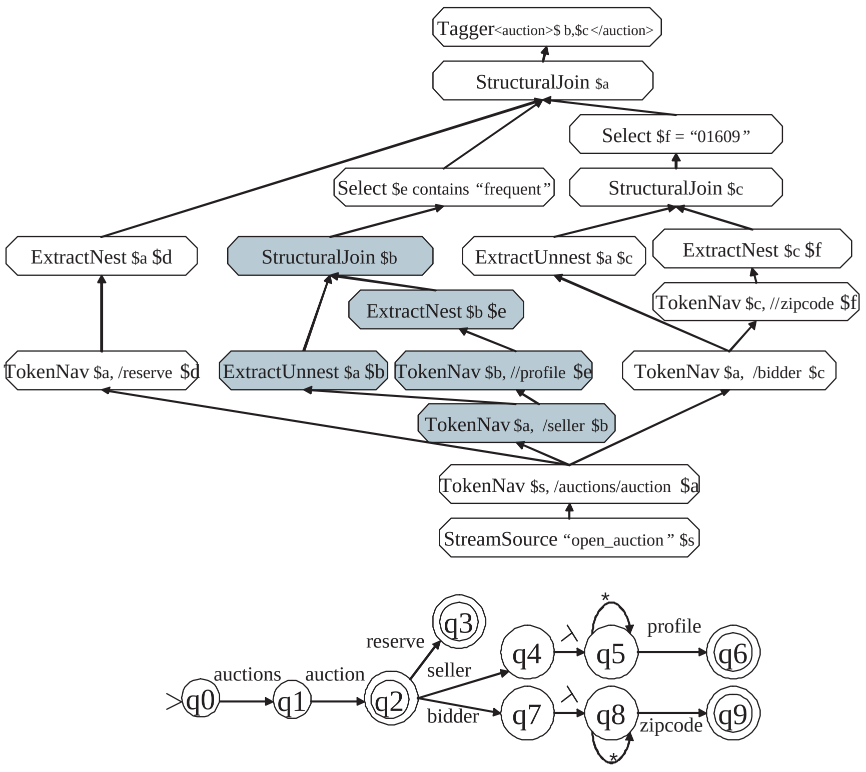


Figure 3.2: Raindrop Plan for Query in Figure 3.1

trieves $\$col1$ (e.g., in Figure 3.4, $TokenNav_{\$col1,path2}\$col3$ has to be placed above $TokenNav_{\$col0,path0}\$col1$). In other words, $TokenNav_{\$col1,path}\$col2$ cannot be commuted with any other operators. This is because of the on-the-fly access nature of stream processing. Tokens cannot be accessed twice[1], $TokenNav_{\$col1,path}\$col2$ must be immediately evaluated on the tokens that compose bindings of $\$col1$.

In contrast, when we pull $\$col2$ = $\$col1/path$ out of the automaton, the resultant $NodeNav_{\$col1,path}\$col2$ may be placed in multiple positions. For example, Figure 3.6 shows a plan after we pull $TokenNav_{\$a,/seller}\$b$ out of the plan in Fig-

[1]Tokens can however be stored as XML element nodes which can be repeatedly accessed.

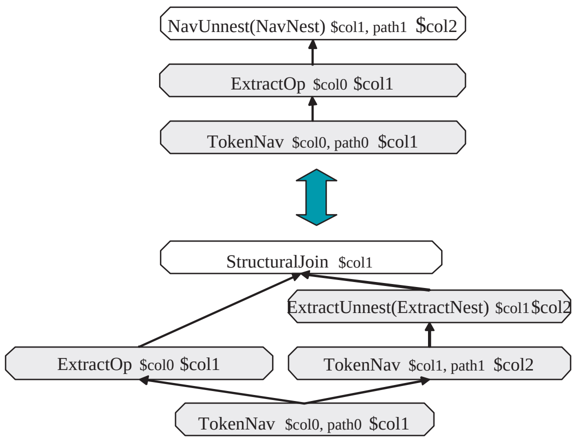

NavUnnest(NavNest) $col1, path1 $col2
ExtractOp $col0 $col1
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col0, path0 $col1

Figure 3.3: Mode Change with Introducing/Eliminating StructuralJoin

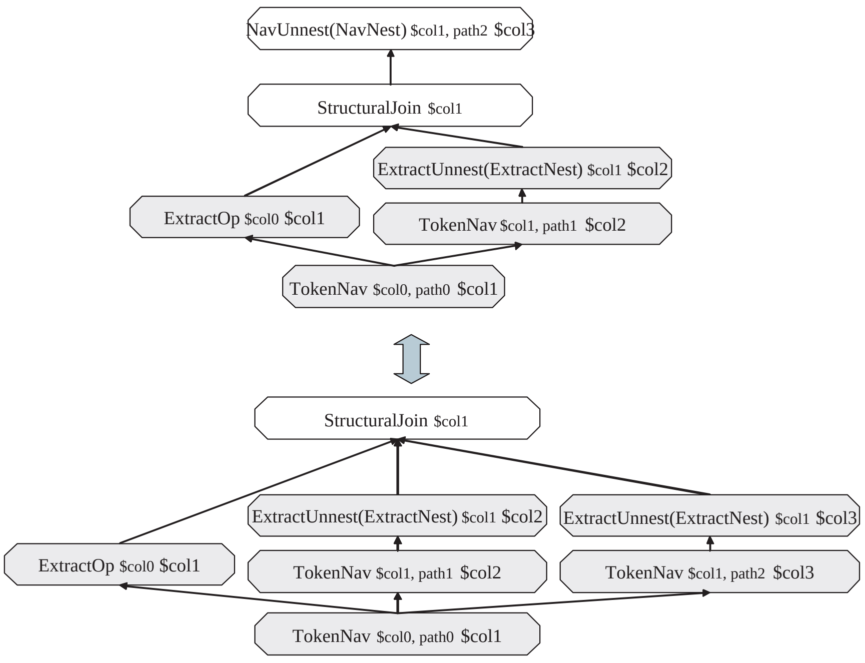

NavUnnest(NavNest) $col1, path2 $col3
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1

Figure 3.4: Mode Change without Introducing/Eliminating StructuralJoin

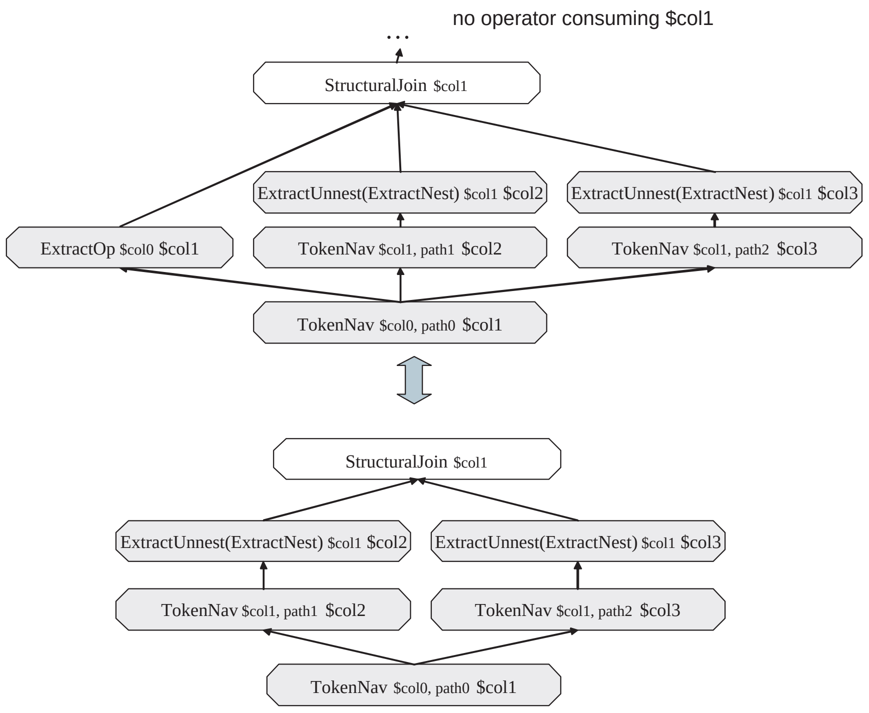

no operator consuming $col1
...
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
ExtractOp $col0 $col1
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1
StructuralJoin $col1
ExtractUnnest(ExtractNest) $col1 $col2
ExtractUnnest(ExtractNest) $col1 $col3
TokenNav $col1, path1 $col2
TokenNav $col1, path2 $col3
TokenNav $col0, path0 $col1

Figure 3.5: Eliminate $Extract_{\$col0}\$col1$ when no Regular Tuple-based Operator Consumes $\$col1$

ure 3.2. If later we pull out $\$d = \$a/reserve$, the resultant $NavNest_{\$a,/reserve}\$d$ is placed by default between $ExtractUnnest_{\$s}\$a$ and $NavUnnest_{\$a,/seller}\$b$. However it can also be placed for example between $NavUnnest_{\$a,/seller}\$b$ and $NavNest_{\$b,//profile}\$e$, because $NavNest_{\$a,/seller}\$b$ still outputs tuples carrying cells bound to $\$a$.

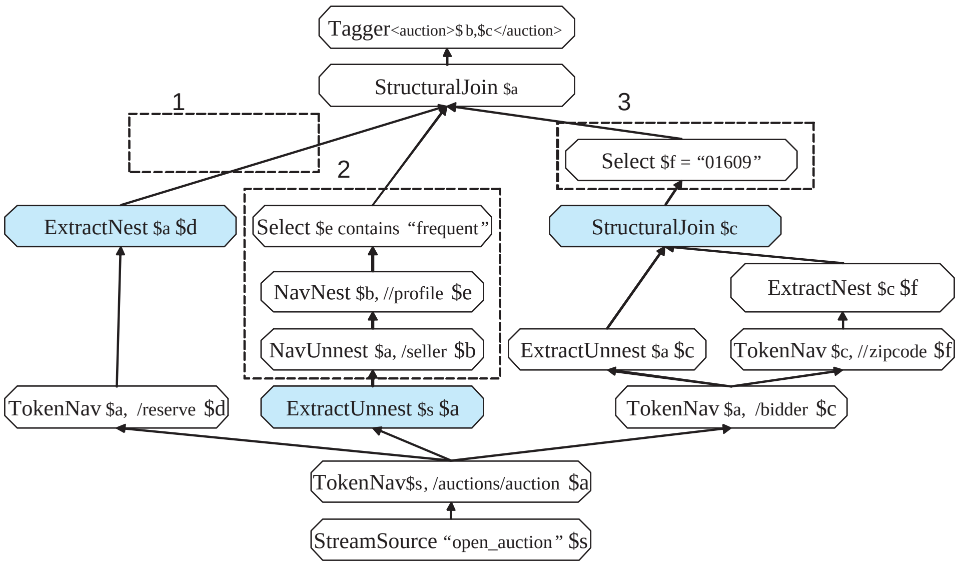


Figure 3.6: Plan Derived from the Pull-out of $TokenNav_{\$a,/seller}\$b$ from Plan in Figure 3.2

Operator commuting has been long studied as an important optimization opportunity [19, 45]. This motivates us to introduce a second kind of rewrite rules in the next section to explore this opportunity.

3.1.2 Operator Commuting Rules

We now list the commuting rules. We use Op_c to represent a $Select$ or a $NodeNav$ operator. c represents the selection predicate if Op is a $Select$ operator, or the path expression if Op is a $NodeNav$ operator. P, P_1, and P_2 in the rewrite rules represent subplans. We also use $\bowtie_{SJ}$ to represent a $StructuralJoin$ operator.

Commuting Op_{c1} with Op_{c2}:

$Op_{c1}(Op_{c2}(P))$ = $Op_{c2}(Op_{c1}(P))$ when both $c1$ and $c2$ involve only columns output generated by a subplan P.

Commuting Op_c with $StruturalJoin$:

$Op_c(P_1 \bowtie_{SJ} P_2)$ =

$(Op_c(P_1)) \bowtie_{SJ} P_2$ when c involves only columns output by P_1.

Figures 3.7 , 3.8 and 3.9 show the examples of commuting a $NodeNav$ operator with a $Select$, another $NodeNav$ and a $StructuralJoin$ operator respectively. A $NodeNav_{\$col1,path1}\$col2$ can commute with any automaton-outside operator as long as the $Extract$ operator that extracts $\$col1$ is still placed under $NodeNav_{\$col1,path1}\$col2$ after the commuting.

3.1.3 Input Subplan Reordering Rule

After we have determined where to place a *NodeNav* operator, we can have further optimization decisions to make. For example, in Figure 3.6, according the execution style of $StructuralJoin$ operators as described in Section 2.5.4, when $StructuralJoin_{\$a}$ is invoked as a </*auction*> is encountered, only the three highlighted operators can have data in their output queues (note that the output of any descendant operator of $StructuralJoin_{\$c}$ must have all been consumed when </*bidder*> was encountered).

For each of the three operators, denoted as op, the intermediate operators be-

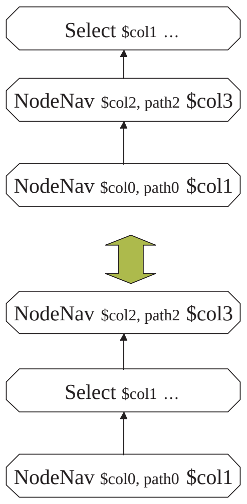

Select $col1 ...
NodeNav $col2, path2 $col3
NodeNav $col0, path0 $col1
NodeNav $col2, path2 $col3
Select $col1 ...
NodeNav $col0, path0 $col1

Figure 3.7: Commuting $NodeNav_{\$col2,path2}\$col3$ with $Select_{\$col1}$

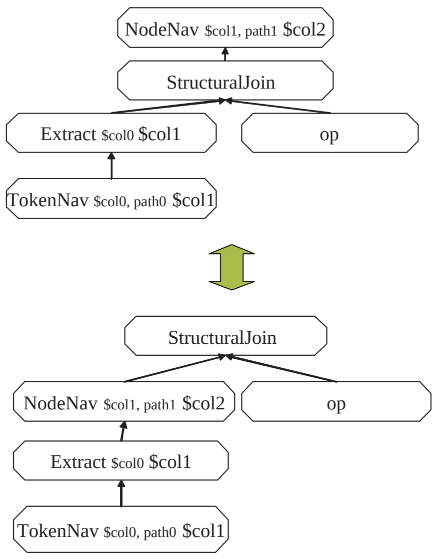

NodeNav $col1, path1 $col2
StructuralJoin
Extract $col0 $col1
op
TokenNav $col0, path0 $col1
StructuralJoin
NodeNav $col1, path1 $col2
op
Extract $col0 $col1
TokenNav $col0, path0 $col1

Figure 3.8: Commuting $NodeNav_{\$col1,path2}\$col2$ with $StructuralJoin$

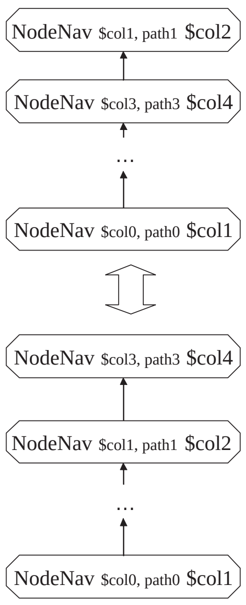

NodeNav $col1, path1 $col2
NodeNav $col3, path3 $col4
...
NodeNav $col0, path0 $col1
NodeNav $col3, path3 $col4
NodeNav $col1, path1 $col2
...
NodeNav $col0, path0 $col1

Figure 3.9: Commuting $NodeNav_{\$col1,path1}\$col2$ with $NodeNav_{\$col3,path3}\$col4$

tween *op* and $StructuralJoin_{\$a}$ must be evaluated when $StructuralJoin_{\$a}$ is invoked. We call the intermediate operators between *op* and $StructuralJoin_{\$a}$ an input subplan of $StructuralJoin_{\$a}$ and *op* the entry operator of this input subplan. For example, the three dashed boxes in Figure 3.6 contain three input subplans of $StructuralJoin_{\$a}$ with entry operators $ExtractNest_{\$a}\$d$, $ExtractUnnest_{\$s}\$a$ and $StructuralJoin_{\$c}$ respectively. Even though there is no intermediate operator between $ExtractNest_{\$a}\$d$ and $StructuralJoin_{\$a}$, for uniformity, we say $Extractnest_{\$a}\$d$ is the entry operator of an empty input subplan of $StructuralJoin_{\$a}$.

The method *ancestorUpstreamDriven* in Algorithm 3 in Section 2.6 describes the process of evaluating these input subplans. When an $</auction>$ is encountered, $StructuralJoin_{\$a}$ is invoked. It then in turn invokes its input operators (lines 7 - 9 in Algorithm 3). Each such input operator again invokes its input operator. Finally, the entry operator is invoked by its parent operator. Therefore the data in the output queue of the entry operator are consumed all the way through the input subplan. In this way, an input subplan is thoroughly evaluated. After all three input subplans have been evaluated, $StructuralJoin_{\$a}$ performs Cartesian products on the output of these input subplans (line 10 in Algorithm 3).

We now propose to further optimize to this process. Algorithm 5 improves Algorithm 3 in two ways.

Precheck of Output of Entry Operators. The first improvement is that when $StructuralJoin_{\$a}$ is invoked, it checks whether all entry operators have generated some output during the processing of the current binding of $\$a$ (lines 7 - 12 in Algorithm 5). Only if yes, $StructuralJoin_{\$a}$ goes on to evaluate the input subplans. For example, suppose $ExtractNest_{\$a}\$d$ does not have output when checked, i.e., the current $auction$ element does not have a $reserve$ child element, then we can save the evaluation of the input subplans contained in the two dashed boxes.

Immediate Stop at Empty Output of Input Subplans. The second improvement is that when we evaluate the input subplans one by one, if a subplan does not generate output, we immediately stop evaluating the rest subplans (lines 17 - 19) since it is guaranteed that the $StructuralJoin$ would not output anything. We however need to assure that all unconsumed data are cleaned up. First, for those input subplans that have already generated output before we stop the evaluation, we clean up their output (lines 28 - 30 in Algorithm 5). Second, for those input subplans that have not been evaluated yet, we clean up their input, i.e., the data generated by their entry operators (lines 31 - 33 in Algorithm 5). This assures correctness as no old data will be mixed with the new data that will be generated

Algorithm 5 Optimized In-Time Structural Join (Compared to Algorithm 3)

```
public class StructuralJoin {
public ancestorUpstreamDriven(){
boolean $allEntryHaveResults$ = TRUE;
boolean $allSubplanHaveResults$ = TRUE;
int $i$;
List $inputTuples$[];
int $n$ = number of input operators of this $StructuralJoin$;
//Precheck of Output of Entry Operators
for each entry operator $entryOp$ of input subplans do
    if $entryOp$ has no data in its output queue then
        $allEntryHaveResults$ = FALSE;
        break;
    end if
end for
if allEntryHaveResults then
    for ($i$ = 1; $i \leq n$; $i$++) do
        Let $inputOp$ denotes the $i^{th}$ input operator;
        List $curInputTuples$ = output generated when $inputOp$ is evaluated;
        //Immediate Stop at Empty Output of Input Subplans
        if $curInputTuples$ are empty then
            allSubplanHaveResults = FALSE;
            break;
        else
            $inputTuples[i]$ = $curInputTuples$;
        end if
    end for
end if
if allSubplanHaveResults then
    outputTuples = join $inputTuples[1]$, $inputTuples[1]$, ..., and $inputTuples[n]$;
else
    for (int j = 1; j $\leq$ i; j++) do
        clean up output queue of the $j^{th}$ input operator.
    end for
    for (int j = i + 1; j $\leq n$; j++) do
        clean up output queue of the entry operator of the $j^{th}$ input subplan.
    end for
end if
... //lines 13 - 21 in Algorithm 3 in Section 2.6 in Chapter 2
}
}
```

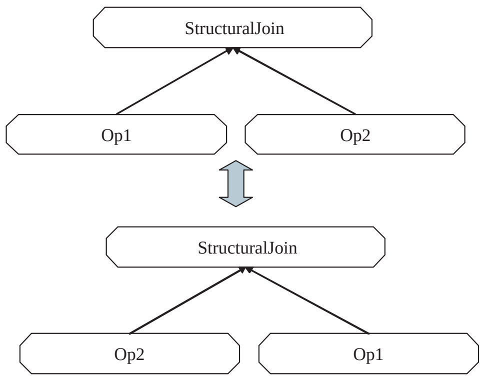


Figure 3.10: Reordering Input Subplans of StructuralJoin

within the next *auction* element.

The order in which we evaluate the input subplans is important for the efficiency. For example, in Figure 3.6, suppose a *seller* seldom has a *profile*, then the second input plan should be evaluated before the third input plan. Therefore if we find that the second input plan does not generate any output within a binding of $\$a$, we do not need to evaluate the third input plan. This can lead to significant cost savings when there is a large number of *bidder* elements in an *auction*. We therefore offer a third rewrite rule called *input subplan reordering*. This rule switches the order of the input subplans whose topmost operators are op_1 and op_2 respectively.

Reordering Input Plans:

$op_1 \bowtie_{SJ} op_2 = op_2 \bowtie_{SJ} op_1$.

This rule is graphically shown in Figure 3.10. In Figure 3.10, we assume the input subplans are evaluated from left to right. We change the order of the input subplans in the top plan and get the bottom plan.

3.1.4 Relationships among Rewrite Rules

The *operator-commuting* and *input-subplan-reordering* rules are designed to complement the *token-or-node mode change* rules. The comparison of the performance when a pattern is retrieved in or out of the automaton should be fair. That means both the automaton processing and non-automaton processing should be optimized. Given a set of patterns to be retrieved in the automaton, the automaton part of the plan is uniquely determined. There are however alternatives for the non-automaton part of the plan. The *operator-commuting* and *input-subplan-reordering* rules are then applied to optimize the non-automaton part of the plan.

3.2 Cost Model

In order to be able to compare two alternative Raindrop plans, we now propose a cost model. In traditional databases, the cost of a plan is defined as the processing time on the whole input data. Since the input stream can possibly be infinite, we need to define the cost of the plan as the processing time on a finite input unit. Because we never allow pulling out the bottommost *TokenNav* operator in order not to buffer the complete incoming stream (refer to Section 2.4.1), all alternatives have the same bottommost *TokenNav* operator. We therefore define the cost of a plan (resp. an operator) as the average processing time on processing the data that originate from one destination element located by the bottommost *TokenNav* operator. For example, the cost of the plan in Figure 3.2 is the average time spent on processing one *auction* element; the cost of $TokenNav_{\$b,//profile}\$e$ is the average time for locating all $profile$ elements within one *auction* element[2]. For simplicity, in the rest of this chapter, we refer to the destination element located by the bottommost *TokenNav* as a *bottom input element*.

We propose our cost model for a scenario with the following features: (1) the statistics are unavailable before the stream comes in; and (2) the query however is known beforehand, i.e., users preregister their queries before the stream arrives. In this scenario, we can run an initial plan of the query on the incoming stream and collect the statistics needed for this particular query. We will further discuss for other scenarios, which parts in our proposed cost model fit and which parts need to be extended in Section 3.2.5.

As described in Section 2.2, besides XML specific operators such as navigation, Raindrop also supports SQL-like operators such as *Select*, *Join*, *Groupby*, *Orderby*, *Union*, *Difference* and *Intersect*. In the first step of cost-based optimization for Raindrop plans, we consider only $Select$ operator among the SQL-like operators. We can however extend to support the other SQL-like operators in future work. Note that the cost model for the SQL-like operators is not a major challenge since it has been widely studied in relational databases. The novel aspect of Raindrop cost model lies more in costing the automaton, which is little studied before.

3.2.1 Unit Costs of Automaton-Outside Operators

In Raindrop implementation, the cost of a unary automaton-outside operator is linear in the number of its input tuples. Also, the cost of the multi-way oper-

[2] Strictly speaking, the cost of a plan in our definition excludes the *StreamSource* and the bottommost *TokenNav* operators. Since these two operators are common in all alternative plans, we are not interested in their costs when comparing two alternative plans.

ator, i.e., *StructuralJoin*, is linear in the product of the number of input tuples from each of its child operators. In other words, in current Raindrop implementation, given an automaton-outside operator op that has n child operators $childOp_1$, $childOp_2$, ..., $childOp_n$, its cost can be expressed as $|childOp_1| \times |childOp_2| \times$ ... $\times$ $|childOp_n| \times UnitCost(op)$ where $|childOp_i|$ ($1 < i < n$) denotes the cardinality of the input originated from $childOp_i$ during the processing of a bottom input element; and $UnitCost(op)$ is the processing time on each input tuple.

We further assume that the unit cost of an operator is not affected by how many number of input tuples the operator processes each time. Aurora [20] observes "intra-operator non-linearity" of tuple processing by an operator. That is, the unit cost of tuple processing may decrease as the number of tuples for processing increases. According to [20], this reduction in unit cost may arise due to two reasons. First, an operator may optimize its execution better with larger number of tuples available for processing. For example, merge joins can be used instead of nested loop joins for larger number of input tuples. Second, the total number of calls to the operator code decreases, cutting down the overhead of function calling. In Raindrop plans, operators do not have different evaluation strategies to cater to larger number or smaller number of tuples. Therefore, "intra-operator non-linearity" cannot arise because of the first reason mentioned above. Most cost models, relational [63**?**] or XML [6, 57], ignore such "non-linearity" arising because of the second reason. This is because it is hard to quantify the overhead of operator code which is very low level. We assume the same in Raindrop.

An important question to ask is, given an operator *op*, is it possible to observe its $UnitCost(op)$ during the execution of an arbitrary plan? If yes, we can directly use this $UnitCost(op)$ observed during the execution of an initial plan. If not, we then have to analyze what factors contribute to $UnitCost(op)$, i.e., cost models for such operators have to be defined at a lower granularity than $UnitCost(op)$. For different operators, the answer is analyzed below:

1). A *Select* operator, when appearing in one plan, must appear in all other equivalent alternative plans because we do not provide any rewrite rule to eliminate a *Select* operator. Therefore, no matter what the currently running plan is, *UnitCost* of a *Select* operator is always observable. Since we assume that the $UnitCost(op)$ is not affected by how many number of input tuples are processed each time the operator code is called, $UnitCost(op)$ observed in a currently running plan is the same as that in any other plans.

2). A *NodeNav* operator does not appear in all plans due to the *token-or-node mode change rule*. Also, a *StructuralJoin* operator may not necessarily appear in every plan, e.g., $StructuralJoin_{\$col1}$ appears in the bottom plan but not the top plan in Figure 2.6 in Section 2.4.2 in Chapter 2. Therefore, *UnitCost*'s of these two operators are not always observable in a currently running plan.

In summary, we may have to estimate the unit cost of a *NodeNav* or a *StructuralJoin* for costing a plan other than the currently running plan but this is not necessary for a *Select* operator. Therefore in the rest of this section, we analyze how to estimate the *UnitCost* for the *NodeNav* and *StructuralJoin* operators only. Table 3.1 gives the notations used for estimating these *UnitCost*.

***UnitCost* of NodeNav.** $UnitCost(NodeNav_{\$u,p}\$v)$ is the time $NodeNav$ spends on navigating into the tree rooted at a node which is a binding of $\$u$ to find all

Notation	Explanation
$n_{p[i]}$	for $NodeNav_{\$u,p}\$v$, we use $p[i]$ to denote the i^{th} navigation step on path p. $p[0]$ denotes the binding of $\$u$. $n_{p[i]}$ denotes average number of children of a binding of $p[i]$ within a binding of $\$u$
$w_{p[i]}$	for $NodeNav_{\$u,p}\$v$, $w_{p[i]}$ denotes average number of a binding of $p[i]$ within a binding of $\$u$
C_{visit}	time for visiting one node in an XML element tree
$C_{bicartesian}$	cost of performing a binary cartesian product, one input tuple from either side

Table 3.1: Notations Used in Defining *UnitCost*'s for $NodeNav$ and $StructureJoin$

the nodes that are bindings of $\$v$. 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 "//" (for uniformity, we also view "//" as a special navigation step). To match the i^{th} navigation step, every child of bindings of the $i - 1^{th}$ navigation step is visited. The number of these child nodes within a binding of $\$u$ is $n_{p[i-1]}w_{p[i-1]}$. Thus the time spent on finding $p[i]$ is $n_{p[i-1]}w_{p[i-1]}C_{visit}$. We then have the below equation.

Equation 1 $UnitCost(NodeNav_{\$u,p[1]/p[2]/.../p[n]}\$v) = \sum_{i=1}^{n} n_{p[i-1]}w_{p[i-1]}C_{visit}$.

***UnitCost* of StructuralJoin.** Suppose a $StructuralJoin$ has n child operators $childOp_1$, $childOp_2$, ..., $childOp_n$. The *UnitCost* of $StructuralJoin$ is defined as the time spent on cartesian producting a tuple output by $childOp_1$, a tuple output by $childOp_2$, ..., with a tuple output by $childOp_n$. This time spent on the cartesian product may differ when n differs. The values of n for a $StructuralJoin_{\$v}$ operator in different alternative plans can be different. n can increase after the mode change of a $NodeNav$ operator (see Figure 2.6 in Section 3.1.1). We ignore this difference to avoid an overcomplicated cost model. We therefore use the unit cost of performing a binary Cartesian product (i.e., n = 2) as the general unit cost of a *StructuralJoin*. We then have the below equation.

Equation 2 $UnitCost(StructuralJoin) = C_{bicartesian}$.

3.2.2 Costs of Input Subplans of StructuralJoin

We have studied how to get $UnitCost(op)$ for an automaton-outside operator op. Now we consider how to compute the cost of op, denoted as $Cost(op)$. As mentioned in Section 3.2.1, $Cost(op) = |childOp_1| \times |childOp_2| \times ... \times |childOp_n| \times UnitCost(op)$. $|childOp_1| \times |childOp_2| \times ... \times |childOp_n|$ is the amount of input to op during the processing of a bottom input element. In a traditional plan, the amount of data that needs to be processed by an operator is only affected by how much data is filtered by its descendant operators (i.e., the selectivity of its descendant operators). However, when a *StructuralJoin* is invoked, an input subplan is executed only when its left sibling subplans have all generated some output. Therefore the amount of data that needs to be processed by an input subplan is also affected by the likelihood of the left sibling subplans having generated some output.

We now define two concepts, *selectivity* and *non-empty-output probability*, of operators. We also define a third concept *entry plan* for entry operators. These concepts are used to compute the cost of an input subplan.

Selectivity: The selectivity of an operator op, denoted as $\sigma(op)$ is defined as below:

1). If op is a $TokenNav_{\$u,p}\$v$ or $Extract_{\$u}\$v$, $\sigma(op)$ is the average number of bindings of $\$v$ generated within a binding of $\$u$.

2). If op is a $Select$, $NodeNav$ or $StructuralJoin$, $\sigma(op)$ is defined as in the traditional databases. Suppose op has n child operators, $\sigma(op)$ is defined as $\frac{cardinality\ of\ op's\ output}{\prod_{i=1}^{n} cardinality\ of\ input\ from\ i^{th}\ child\ operator\ of\ op}$.

Non-empty-result Probability: The non-empty-result probability of an operator op is denoted as $P_{\not\Rightarrow\emptyset}(op)$. "$\not\Rightarrow$ $\emptyset$" in the notation means "not generating an empty result", i.e., generating some result. It is defined as below:

1). If op is a $TokenNav_{\$u,p}\$v$, $P_{\not\Rightarrow\emptyset}(op)$ is the probability of a binding of $\$u$ containing at least one binding of $\$v$.

2). If op is a $Select$ or $NodeNav$, $P_{\not\Rightarrow\emptyset}(op)$ is the probability of op generating some output during the processing of one input tuple.

Entry Plan: As described in Section 3.1.3, a $StructuralJoin_{\$v}$ has several entry operators. For example, in Figure 3.6, the three highlighted operators are the entry operators of $StructuralJoin_{\$a}$. There are intermediate operators between an entry operator and the $TokenNav$ operator that retrieves $\$v$. We call the plan consisting of these intermediate operators (including the entry operator) an *entry plan*. In Figure 3.6, there are five intermediate operators between the entry operator $StructuralJoin_{\$c}$ and $TokenNav_{\$s,/auctions/auction}\$a$, i.e., $StructuralJoin_{\$c}$, $ExtractUnnest_{\$a}\$c$, $ExtractNest_{\$c}\$f$, $TokenNav_{\$c,//zipcode}\$f$, and $TokenNav_{\$a,/bidder}\$c$. We say the plan composed of these five operators an entry plan of the entry operator $StructuralJoin_{\$c}$. We use the function $entryPlan(op)$ to denote the entry plan of an entry operator op.

Assume the input subplans of $StructuralJoin_{\$v}$ from left to right are $subplan_1$, $subplan_2$, ..., $subplan_n$ with entry operators $entry_1$, $entry_2$, ..., $entry_n$ respectively. Equation 3 computes the cost of $subplan_i$ ($1 \leq i \leq n$).

Equation 3 *$Cost$($subplan_i$ of $StructuralJoin_{\$v}$)*

= number of bindings of $\$v$ within one bottom input element (1)

$\times$ *evaluation time of* $subplan_i$ *on input generated within a binding of* $\$v$ *(2)*

= $\prod_{op\ \in\ operator\ set\ between\ bottommost\ TokenNav\ and\ TokenNav\ that\ retrieves\ \$v} \sigma(op)$ *(3)*

$\times$ *probability of* $subplan_i$ *being evaluated (4)*

$\times$ *amount of input tuples to* $subplan_i$ *within a binding of* $\$v$*(5)*

$\times$ *evaluation time of* $subplan_i$ *on one input tuple (6)*

= $\prod_{op\ \in\ operator\ set\ between\ bottommost\ TokenNav\ and\ TokenNav\ that\ retrieves\ \$v} \sigma(op)$ *(7)*

$\times$ $P_{\neq\emptyset}(entry_1)P_{\neq\emptyset}(entry_2)...P_{\neq\emptyset}(entry_n)$ *(8.a)*

$\times$ $P_{\neq\emptyset}(subplan_1)...P_{\neq\emptyset}(subplan_{i-1})$ *(8.b)*

$\times$ $\sigma(entryPlan(entry_i))$ *(9)*

$\times$ $UnitCost(subplan_i)$ *(10)*

In Equation 3, Expression (1) is expanded into Expression (3). When we say “operator set between bottommost $TokenNav$ and the $TokenNav$ that retrieves $\$v$”, the set does not include bottommost $TokenNav$ but it includes the $TokenNav$ that retrieves $\$v$. Any operator in the set is a $TokenNav$ that retrieves an ancestor pattern of $\$v$ or $\$v$ itself. For example, suppose we want to cost an input subplan of $StructuralJoin_{\$c}$ in Figure 3.6. To compute the number of bindings of $\$c$ in the bottom input element, the operators set between the bottommost $TokenNav$ (i.e., $TokenNav_{\$s,/auctions/auction}\$a$) and the $TokenNav$ that retrieves $\$b$ (i.e., $TokenNav_{\$a,/seller}\$c$) is $\{TokenNav_{\$a,/seller}\$c\}$. Expression (3) is then expanded as $\sigma(TokenNav_{\$a,/seller}\$c)$, i.e., the number of bindings of $\$c$ within a binding of $\$a$.

Expression (2) is expanded into Expressions (4) (5) and (6). Expression (4) later is expanded into Expressions (8.a) and (8.b). Expression (8.a) gives the probability of all entry operators generating output while Expression (8.b) gives the possibility of all left sibling input plans of $subplan_i$ generating output.

Finally, Expressions (5) and (6) are expanded into Expressions (9) and (10) respectively. The average number of tuples generated by $entry_i$ within a binding of $\$v$ is the selectivity of the entry plan of $entry_i$, i.e., $\sigma(entryPlan(entry_i))$ in Expression (9). The unit cost of processing one input tuple of $subplan_i$ is $UnitCost(subplan_i)$ in Expression (10).

$\sigma(entryPlan(entry_i))$ and $UnitCost(subplan_i)$ require us to compute the selectivity and the cost of a plan respectively. This can be computed exactly as in traditional databases. We compute the selectivity of a plan as below.

1). For a plan P = $P_A(P_B)$ which means subplan P_A consumes output of subplan P_B, $\sigma(P)$ = $\sigma(P_B) \times \sigma(P_A)$; and $Cost(P)$ = $n \times UnitCost(P_B)$ + $n \times \sigma(P_B) \times UnitCost(P_A)$ where n is the number of input to P_B.

2). For a plan P = P_A $JoinOp$ P_B which means subplan P_A is joined with subplan P_B by $JoinOp$, $\sigma(P)$ = $\sigma(P_A) \times \sigma(P_B) \times \sigma(JoinOp)$; and $Cost(P)$ = $n_A \times UnitCost(P_A)$ + $n_B \times UnitCost(P_B)$ + $n_A \times n_B \times \sigma(JoinOp) \times UnitCost(JoinOp)$ where n_A and n_B are the number of input tuples to P_A and P_B respectively.

By breaking a bigger plan into smaller subplans, we can eventually compute the selectivity/cost of a plan from the selectivity/cost of its operators.

3.2.3 Costs of Automaton-Inside Operators

In the previous section we have discussed how to compute costs for automaton-outside operators. We now describe how to compute the costs for the automaton-inside operators. We first briefly recap how an automaton is used to retrieve patterns while more details can be found in Section 2.5. An automaton behaves as below:

1). When an incoming token is a start tag:

   a. If the stack top is not empty, the incoming token is looked up in the transition entries of every state at the stack top. The automaton pushes the states that are transitioned to onto the stack. If no states are transitioned to, the automaton pushes an empty set (denoted as $\emptyset$) onto the stack.

   b. When the stack top contains an empty set, the automaton directly pushes another empty set onto the stack without any lookup.

2). When an incoming token is an end tag: the automaton pops the states at the stack top off the stack.

3). When an incoming token is a PCDATA token, the automaton makes no change to the stack.

4). An incoming token (start tag, end tag or PCDATA token) is stored if required by an $Extract$ operator.

When costing a pattern retrieval, we need to be careful with "amortized" computations. For example, in Figure 3.11, when a stack top contains instances of

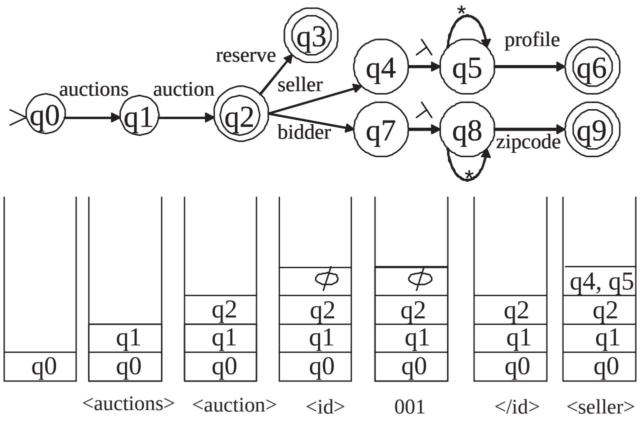


Figure 3.11: Automaton of Plan in Figure 3.2 and Stack Snapshots

$q4$ and $q5$ (see the rightmost stack), an incoming $</seller>$ will lead to a stack backtrack. However we cannot solely assign this backtracking cost to the pattern retrieval $\$a/seller$. This is for two reasons. First, even if the query does not ask for $\$a/seller$, backtracking is still needed when $</seller>$ is encountered in order to restore the stack to the status before the matching $<seller>$ has been encountered. Second, the backtracking cost is a constant, i.e., it is not affected by which states are popped or the number of states popped. For example, in Java implementation, we can simply move the reference to the stack top one level down to accomplish the state pop-off.

To avoid repeatedly costing the same amortized computations, we analyze the cost of retrieving a pattern p by comparing the cost of running a stream on an automaton A_{with} and the cost of running the same stream on another automaton $A_{without}$. A_{with} denotes an automaton that encodes $\$v/p$ and all the ancestor patterns of $\$v/p$ (e.g., the ancestor patterns of $\$b//profile$ are $\$a = auctions/auction$ and $\$b = \$a/bidder$). $A_{without}$ encodes only the ancestor patterns of $\$v/p$. Since

Notation	**Explanation**
$Q(A)$	states in an automaton A
$Q_{extract}(A)$	states associated with extraction operators, e.g., states q_4 and q_7 in Figure 3.11
C_{nonEmp}	cost of processing a start token when stack top is not empty
C_{emp}	cost of processing a start token when stack top is empty
$C_{backtrack}$	cost of popping off states at the stack top
$C_{extract}(q)$	cost of buffering elements, whose start tags activates state q, in a bottom input element
$n_{active}(q)$	the number of times that stack top contains a state q when a start tag arrives in a bottom input element. Each such tag is the start tag of a child of an element that activates q
n_{start}, n_{end}	number of start or end tags in a bottom input element. $n(start) = n(end)$.

Table 3.2: Notations Used in Cost of Automaton-Inside Operators

A_{with} and $A_{without}$ only differs in that A_{with} retrieves an additional pattern p, the cost difference of running a stream on A_{with} and $A_{without}$ is then the cost of retrieving p in the stream.

We first study how to compute the cost of running a stream on an automaton. Given an automaton A with a start state which is activated by a start tag of the bottom input element (i.e., q_2 in Figure 3.11), the cost of running a stream on the automaton A is:

Equation 4 $Cost(A)$ =

state transiting cost for processing start tags *(1)*

+ stack backtracking cost for processing end tags *(2)*

+ extracting cost for processing tokens *(3)*

Using the notations in Table 3.2, we can refine Equation 4 to Equation 5.

Equation 5 $Cost(A)$ =

$\sum_{q \in Q(A)} n_{active}(q)\ C_{nonEmp}$ *(1.a)*

$+ [n_{start}$ - $\Sigma_{q \in Q(A)} n_{active}(q)]\ C_{emp}$ *(1.b)*

$+ \, n_{end} \, C_{backtrack}$ *(2)*

$+ \sum_{q \in Q_{extract}(A)} n_{active}(q) \, C_{extract}(q)$ *(3)*

$= \sum_{q \in Q(A)} n_{active}(q) \, (C_{nonEmp} - C_{emp}) + n_{start}(C_{emp} + C_{backTrack})$ *(4)*

$+ \sum_{q \in Q_{extract}(A)} n_{active}(q) \, C_{extract}(q)$ *(5)*

Expression (1) in Equation 4 is expanded into Expressions (1.a) and (1.b) in Equation 5. A start tag activates more than one state only when "//" occurs in the query, namely, there are λ transitions and self transitions. For example, in Figure 3.11, if q_4 is at the stack top, q_5 must be at the stack top as well. Since λ transitions and self transitions usually is only a small portion of the transitions in an automaton, $\sum_{q \in Q(A)} n_{active}(q)$ is approximately equal to the number of start tokens that are processed with a non-empty stack top. Therefore Expression (1.a) is 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 (n_{start} - number of start tags that are processed with an non-empty stack top) = (n_{start} - $\Sigma_{q \in Q(A)} n_{active}(q)$). Expression (1.b) of Equation 5 thus is 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, $C_{backtrack}$. Since there are n_{end} end tags in a bottom input element, Expressions (2) in Equation 5 is the cost of processing end tags.

In Expression (3) in Equation 5, $q \in Q_{extract}(A)$ is a state associated with an $Extract$ operator. $n_{active}(q) C_{extract}(q)$ then denotes the cost of storing the elements whose start tags activate q. Therefore Expression (3) in Equation 5 is the total extraction cost.

We now know how to compute the cost of running a stream on a given automaton. We can then compute the cost of a $TokenNav_{\$u,p}\$v$ operator by computing $Cost(A_{with})$ - $Cost(A_{without})$, as shown in Equation 6. A_p in the equation denotes the sub-automaton that encodes $\$v/p$ only.

Equation 6 *Cost*$(TokenNav_{\$u,p}\$v)$

= *Cost*(A_{with}) $-$ *Cost*$(A_{without})$

= $\sum_{q \in Q(A_{with})-Q(A_{without})} n_{active}(q)\ (Cost_{nonEmp} - Cost_{emp})$

$+ \sum_{q \in Q_{extract}(A_{with})-Q_{extract}(A_{without})} n_{active}(q)\ Cost_{extract}(q)$

= $\sum_{q \in Q(A_p)} n_{active}(q)\ (Cost_{nonEmp} - Cost_{emp})$

$+ \sum_{q \in Q_{extract}(A_p)} n_{active}(q)\ Cost_{extract}(q)$

3.2.4 Cost Model Summary

The cost of a plan consists of two parts. The first part is the cost of all pattern retrieval performed in the automaton. We use Equation 6 to compute the cost of each pattern retrieval. The second part is the cost of automaton-outside operators. The automaton-outside operators can be divided into several disjunct groups, each group composed of a *StructuralJoin* and its input subplans. We can use Equation 3 to compute the cost of each such group.

3.2.5 Discussion on Extension of Cost Models

In the beginning of Section 3.2, we mentioned that we assume the user query is known beforehand. With this assumption, we can then run an initial plan of the query and collect the statistics needed for this particular query. The impact of this query specific statistics collection mechanism is that for those operators that appear in all alternative plans, we do not need to further analyze what factors contribute to their $UnitCost$ because their $UnitCost$ can be directly observed in the currently running plan.

There are two scenarios in which the above statistics collection mechanism does not fit. The first scenario is that the stream query engine has to process a large number of queries so that it cannot afford to collect specific statistics for each query. Statistics summary techniques [2, 84] are needed to achieve good scalability. The second scenario is that the user adds a new query after the stream starts to arrive. Of course we can still run an initial plan of this new query and collect statistics for it if scalability is not a concern here. Another solution is that we always summarize the statistics as the stream runs so that once a new query is added, we can immediately provide cost estimates and choose a plan for this query. This solution is essentially static optimization, i.e., getting the statistics, choosing a plan and running the chosen plan.

In summary, in both scenarios, we can estimate the cost of a plan from general statistics instead of specific statistics for this particular plan. A summary statistics collection mechanism may not observe the *UnitCost* of all $Select$ operators in the plans. For example, operators that involve a “contain” function such as $Select_{\$e\ contains\ “frequent”}$ are quite common in queries on text-centered XML document [7]. The query-specific statistics collection mechanism ensures that we can directly observe the *UnitCost* of such operators. In the summary statistics collection mechanism however, we need to enhance our cost model by analyzing what factors contribute to evaluating, for example, a “contain” function. Except such enhancement on analyzing the *UnitCost*’s of the functions used in $Select$ operators, all the other parts of our cost model still fit in the summary statistics collection mechanism.

3.3 Combining Heuristics and Costs for Operator Commuting

The operator-commuting and input-subplan-reordering rules optimize the non-automaton part of a plan. The operator-commuting rule reorders two operators that have a parent-child relationship while the input-subplan-reordering rule reorders subplans that have a sibling relationship. We sometimes refer to these two rules as parent-child operator reordering and sibling operator reordering respectively. These two rules are not independent of each other. That means, optimizing a plan using one rule first and then optimizing the plan using the second rule does not ensure the resultant plan is overall optimal. We now give an example to illustrate the dependency relationship between the two rules.

Example 12 *Without the input-subplan-reordering rule, the likelihood of an operator being executed is only decided by the selectivity of its descendant operators. For example, in Figure 3.2, if we push down* $Select_{\$f=\text{“01609”}}$ *under* $StructuralJoin_{\$c}$*,* $Select_{\$f=\text{“01609”}}$ *will be placed between* $ExtractNest_{\$c}\$f$ *and* $StructuralJoin_{\$c}$*.* $Select_{\$f=\text{“01609”}}$ *was above* $ExtractUnnest_{\$a}\$c$ *before the push-down.* $ExtractUnnest_{\$a}\$c$ *simply wraps tokens that are components of bindings of* $\$c$ *into tuples. In other words,* $ExtractUnnest_{\$a}\$c$ *does not filter the input so that whether it is executed before* $Select_{\$f=\text{“01609”}}$ *or in parallel with* $Select_{\$f=\text{“01609”}}$ *does not affect the cost of* $Select_{\$f=\text{“01609”}}$*. Therefore, pushing down* $Select_{\$f=\text{“01609”}}$ *under* $StructuralJoin_{\$c}$ *does not change the cost of* $Select_{\$f=\text{“01609”}}$*. However* *pushing down $Select_{\$f=\text{“01609”}}$ under $StructuralJoin_{\$c}$ can decrease the cost of $StructuralJoin_{\$c}$ since $Select_{\$f=\text{“01609”}}$ can filter input to $StructuralJoin_{\$c}$.*

In summary, $Select_{\$f=\text{“01609”}}$ would be pushed down under $StructuralJoin_{\$c}$ when only the operator-commuting rule is considered. However if we in addition consider the input-subplan-reordering rule, leaving $Select_{\$f=\text{“01609”}}$ above the $StructuralJoin_{\$c}$ operator as in Figure 3.2 may be better than pushing it down because we may be able to save the evaluation of $Select_{\$f=\text{“01609”}}$ when its left sibling subplans, for example, $Select_{\$e\ contains\ \text{“}frequent\text{”}}$, are very selective. Therefore, the parent-child operator relationships in a plan optimized without input-subplan-reordering rule are not necessarily the same as those in a plan optimized with the input-subplan-reordering rule.

Since the search space generated by only the token-or-node mode change rules can be already very large (a query with n patterns can have up to 2^n alternative plans), we prefer to optimize the non-automaton part of a plan in a short time. We therefore use a search strategy that basically considers the operator-commuting and input-subplan-reordering rules independently, i.e., optimize in two phases. In the first phase, we optimize using only the operator-commuting rule on the initial plan and get a new plan.. In the second phase, we then optimize the plan derived in the first phase using the input-subplan-reordering rule only. Such a strategy prevents a search space explosion compared to considering all combinations of applying both types of rules. It however is not exactly an independent search, since some operator-commuting decisions we make in the first phase target leaving opportunities for the later input-subplan-reordering optimization. For example, in Example 12, we may choose to place $Select_{\$f=\text{“01609”}}$ above $StructuralJoin_{\$c}$.

We present how to make the operator commuting decisions in this section while we present how to make the input-subplan-reordering decisions in Section 3.4.

3.3.1 Using both Heuristics and Costs for Operator Commuting

The operator commuting in Raindrop plans can be divided into two types. One is commuting *Select*-like operators with each other. Besides *Select* operators, *NodeNav* operators are special *Select* operators because a $NodeNav_{\$u,p}\$v$ has only one child operator and filters out input tuples whose bindings of $\$u$ do not contain a path p. The second type is commuting *Select*-like operators with *StructuralJoin* operators. Note that in a Raindrop plan, *StructuralJoin* operators cannot be commuted with each other. Suppose we have $StructuralJoin_{\$u}$ and $StructuralJoin_{\$v}$ where $\$v$ is a descendant element within a $\$u$ (i.e., $\$v$ can be expressed as $\$v$ = $\$u/p$). Because of the sequential manner of accessing token streams, a binding of $\$v$ must be completely accessed before the corresponding binding of $\$u$ has been completely accessed. That dictates that $StructuralJoin_{\$v}$ is always performed before $StructuralJoin_{\$u}$ on the data that are located within the same binding of $\$u$. That is to say, the order among ancestor and descendant $StructuralJoin$ operators is fixed by the query semantics. Therefore $StructuralJoin$ can only be commuted with *Select* and *NodeNav* operators.

For the first type of commuting, i.e., commuting *Select*-like operators with each other, we can utilize some existing techniques. [19] proposed a cost-based technique for determining the order of *Select*-like operators. The basic idea is to define a rank function on the operators. The operators are then evaluated in the ascending order of their rank functions. This order is guaranteed to be optimal. The rank function on a *Select*-like operator op is defined as $rank(op)$ = $\frac{\sigma(op)-1}{UnitCost(op)}$ where

$\sigma(op)$ is the selectivity of op (i.e., $\frac{number\ of\ output\ tuples}{number\ of\ input\ tuples}$) and $UnitCost(op)$ is the cost of processing one input tuple in op. Intuitively, this rank function indicates that if the operator has a low unit cost (i.e., processes one input tuple quickly) and a low selectivity (i.e., filter many of its input tuples), it should be executed early. Such a rank function based technique can also be used to commute *Select* or *NodeNav* operators in Raindrop.

For the second type of commuting, i.e., commuting *NodeNav* or *Select* with *StructuralJoin*, the above cost-based technique no longer applies. [45] extends the rank function based technique in [19] to reorder *Select* and *Join* operators. [45] assumes certain properties of the *Select* and *Join* operators. Suppose a *Join* operator has two child operators Sel_1 and Sel_2. [45] assumes that commuting either $Select$ operator with $Join$ only affects the costs of these two operators commuted. For example, commuting Sel_1 with $Join$ does not change the cost of Sel_2. Suppose a *StructuralJoin* also has two child *Select* operators Sel_1 and Sel_2 from left to right. Since we only evaluate Sel_2 after Sel_1 has generated output, the cost of Sel_2 is affected by the non-empty-probability of Sel_1. Commuting Sel_1 with $Join$ can increase the cost of Sel_2. The assumption that only the costs of the operators involved in the commuting are changed is violated. Therefore, the rank function based technique does not work for commuting *Select* or *NodeNav* with *StructuralJoin*. We instead propose heuristics for making decisions for such type of commuting.

In summary, we use the existing rank function based technique [19, 45] to commute *Select* or *NodeNav* operators in Raindrop while we propose heuristics to commute *NodeNav* or *Select* with *StructuralJoin*. We have discussed how to compute $\sigma(op)$ and $UnitCost(op)$ in Section 3.2. We now describe our heuristics for commuting *NodeNav* or *Select* with *StructuralJoin*.

3.3.2 Heuristics for Commuting Select/NodeNav with StructuralJoin

We categorize $StructuralJoin_{\$v}$ into three cases and propose heuristics for each case.

Case 1: $StructuralJoin_{\$v}$ with Duplicate $\$v$ Output

Heuristic 1: Given a *Select* or *NodeNav* operator *op* and a *StructuralJoin*, we place *op* beneath the *StructuralJoin* if an $ExtractUnnest_{\$v}\$w$ or $NavUnnest_{\$v,p}\$w$ ($NavUnnest_{\$v,p}\$w \neq$ *op*) exists in the plan.

For example, in Figure 3.6, $NavUnnest_{\$a,/seller}\$b$ would be placed beneath $StructuralJoin_{\$a}$ since there exists a $TokenNav_{\$a,/bidder}\$c$ in the plan. Suppose we instead place $NavUnnest_{\$a,/seller}\$b$ above $StructuralJoin_{\$a}$, if one *auction* has 10 *bidder*'s, $StructuralJoin_{\$a}$ will output 10 tuples for this *auction*, one tuple for one different *bidder*. Then the same *auction* will be navigated into 10 times by $NavUnnest_{\$a,/seller}\$b$ to locate the *seller*.

Our experiment in Figure 2.27 in Section 2.7.2 in Chapter 2 has illustrated that such duplicate computations seriously degrade the plan performance. We therefore propose such a heuristic to avoid any duplicate computations.

Case 2: Intermediate $StructuralJoin_{\$v}$ without Duplicate $\$v$ Output

Heuristic 2: We place a *Select* or *NodeNav* operator *op* above a $StructuralJoin_{\$v}$ if (1) except *op*, no $ExtractUnnest_{\$v,p}\$w$ nor $NavUnnest_{\$v,p}\$w$ exists in the plan and (2) other $StructuralJoin$ operators exist above $StructuralJoin_{\$v}$.

This heuristic is designed to provide more opportunities for structural join related optimization. Two operators that belong to the input plans of different *StructuralJoins* have no impact on each other's execution. For example, in Figure 3.2, suppose we place $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ and $Select_{\$f=\text{“01609”}}$ beneath $StructuralJoin_{\$b}$ and $StructuralJoin_{\$c}$ respectively. Each time when a $</seller>$ is encountered and $profile$ elements are located within this $seller$ (resp. When a $</bidder>$ is encountered and $zicode$ elements are located), $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ (resp. $Select_{\$f=\text{“01609”}}$) would have to be performed. Instead, consider the case in which we place $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ and $Select_{\$f=\text{“01609”}}$ above $StructuralJoin_{\$b}$ and $StructuralJoin_{\$c}$ as in Figure 3.2. When a $</auction>$ is encountered, $bidder$ elements would have to be found within this $auction$ before $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ could possibly be performed. This is because *StrucutralJoin* only evaluates its input subplans when all entry operators have generated output (see “precheck of output of entry operators” in Algorithm 3). Moreover, The evaluation of $Select_{\$f=\text{“01609”}}$ will not be performed if $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ does not generate any output (see ‘immediate stop at empty output of input subplans” in Algorithm 3). Therefore, both $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ and $Select_{\$f=\text{“01609”}}$ are more likely to be avoided after the commuting.

Generally speaking, before this commuting rewriting, the *Select* and *NodeNav* operators were scattered in the input plans of different *StructuralJoins*. After the rewriting, these operators are “concentrated” into the subplans of less *StructuralJoin* operators. For example, all *Select* operators occur in the input subplans of $StructuralJoin_{\$a}$ in Figure 3.2. Such operators are then less likely to be evaluated because of the optimization techniques in *StructuralJoin*.

Case 3: Topmost $StructuralJoin_{\$v}$

Heuristic 3: We place a *Select* or *NodeNav* operator op underneath a $StructuralJoin$ if no other $StructuralJoin$ operators exist above $StructuralJoin_{\$v}$.

Placing op above $StructuralJoin_{\$v}$ does not open up any optimization opportunity for input subplan reordering as in the second case. It may even increase the cost of $StructuralJoin_{\$v}$ because more data that could otherwise be filtered out by $Select$ or $NodeNav$ are now input to $StructuralJoin_{\$v}$. Therefore for such a topmost $StructuralJoin$, $Select$ or $NodeNav$ operators should be placed underneath it. For example, in Figure 3.2, we keep both $Select_{\$e\ contains\ \text{“}frequent\text{”}}$ and $Select_{\$f\ =\ \text{“01609”}}$ underneath $StructuralJoin_{\$a}$, since it is the topmost $StructuralJoin$ in the plan.

3.3.3 Operator Commuting Algorithm

We now describe the algorithm that commutes the operators in a Raindrop plan. Lemma 1 shows an important property that is utilized in the algorithm.

Lemma 1 ***Order of Applying Commuting Rules Being Insensitive:*** *Given two operators $op1$ and $op2$ ($op1$ or $op2$ can be either a* Select *or a* NodeNav*), whether we commute $op1$ with $StructuralJoin$ or commute $op2$ with $StructuralJoin$ first does not affect the final outcome.*

Proof 1 *We decide which category a* StructuralJoin *belongs to by checking whether this* StructuralJoin *can output duplicates and whether it is the topmost* StructuralJoin. *Commuting two operators would not eliminate an $ExtractUnnest_{\$v}\$w$ or $NavUnnest_{\$v,p}\$w$. It therefore does not change the property of a $StructuralJoin_{\$v}$ outputting dupli-*

cates or not. Also, the commuting would not eliminate any StructuralJoin *so that it does not change the property of a* $StructuralJoin_{\$v}$ *being topmost or not. That is to say, no matter how we commute operators, the category that a* $StructuralJoin$ *belongs to does not change. Since whether an operator should be commuted with a* $StrucutralJoin$ *is completely decided by the category of the* StrucutralJoin, *which is unchanged, the order in which we commute operators with a* StrucutralJoin *does not affect the final outcome.*

Algorithm 6 Commuting Operators Using both Heuristics and Costs

```
1:  for each $StructuralJoin$ in the plan do
2:    if $StructuralJoin$ falls in the first or third category then
3:      while its parent is a Select or NodeNav operator do
4:        commute this $StructuralJoin$ with its parent
5:      end while
6:    else
7:      while it has Select or NodeNav child operators do
8:        commute $StructuralJoin_{\$v}$ with each child operator
9:      end while
10:   end if
11: end for
12: for each $StructuralJoin$ in the plan do
13:   for each input subplan of $StructuralJoin$ do
14:     commute operators within the input subplan according to their rank functions
15:   end for
16: end for
```

Algorithm 6 shows the optimization using the operator-commuting rules. We perform the commuting in two steps. In the first step, we use the heuristics to commute $StructuralJoin$ with $Select$ or $NodeNav$ operators (lines 1 -11). Lemma 1 shows that the order in which we commute a *Select* or a *NodeNav* with *StructuralJoin* does not matter. We therefore traverse each *StructuralJoin* and commute the *Select* or *NodeNav* operators with it. Since In the second step, we visit each input subplan of $StructuralJoin$ operators. For each input subplan, we use the rank functions [19, 45] to commute between the *Select* and *NodeNav* operators (lines 12 - 16).

3.4 Using Rank Functions for Input Subplan Reordering

The problem of reordering input subplans of $StructuralJoin$ bears some resemblance to the problem of ordering select and join operators [19, 45]. However, the operators [19, 45] considered to be reordered must have a consuming-producing relationship, i.e., the output of one select operator will be the input to another select operator. In contrast, the subplans in our scenario do not have such relationships. For example, in Figure 3.2, the output of the subplan containing $Select_{\$f=\text{“01609”}}$ is not sent to the subplan containing $Select_{\$e\ contains\ \text{“}frequent\text{”}}$. We therefore extend the techniques in [19, 45] and derive a criterion, shown in Theorem 2, for deciding the optimal evaluation order of input subplans.

Theorem 2 *The cost of input subplans is minimal when they are evaluated in the ascending order of their rank functions as defined below:*

$$rank(subplan) = \frac{\sigma(entryPlan(entryOp))UnitCost(subplan)}{1-P_{\not\Rightarrow\emptyset}(subplan)}.$$

The proof can be found in Appendix C. Intuitively, this criterion says, a subplan should be evaluated early if (1) its entry operator filters many of its input tuples, i.e., small $\sigma(entryPlan(entryOp))$, (2) it costs little, i.e., small $UnitCost(subplan)$, and (3) it often does not generate any output each time when the *StructuralJoin* is invoked, i.e., small $P_{\not\Rightarrow\emptyset}(subplan)$.

3.5 Enumerative Search for One-time Optimization

In the previous two sections, we studied how to optimize the non-automaton part of a plan. We now address whether a pattern should be retrieved in the automaton or out of the automaton. In this section, we present an enumerative search algorithm which ensures: (1) all possible alternative plans are explored so that it guarantees to find the optimal plan and (2) an alternative is never explored twice.

Suppose the initial plan has n pattern retrieval operators. Our exhaustive search algorithm enumerates the combinations (i.e., subsets) of the n pattern retrieval operators. For each combination of operators, we change the modes of the operators in the initial plan and get an alternative plan. However, as stated in Lemma 2, certain combinations can lead to the generation of plans that are redundant. Such combinations are not explored by our exhaustive search algorithm.

Lemma 2 ***Combinations Containing Operators with Pattern Dependency Relationship being Redundant:*** *Suppose $navOp_1$ and $navOp_2$ have pattern dependency relationship. They retrieve two patterns $\$v = \$u/p1$ and $\$y = \$x/p2$ where $\$x$ is an element within $\$v$. $navOp_1$ and $navOp_2$ can be either a $TokenNav$ or a $NodeNav$ type. They are not necessarily the same types. A combination containing both $navOp_1$ and $navOp_2$ produces the same plan as another combination that contains no operators with pattern dependency relationship.*

Proof 2 *We distinguish between three cases: first, $\$u/p1$ and $\$x/p2$ are both retrieved in the automaton; second, $\$u/p1$ and $\$x/p2$ are both retrieved out of the automaton; third, $\$u/p1$ is retrieved in the automaton while $\$x/p2$ is retrieved out of the automaton. The fourth case, i.e., $\$u/p1$ is retrieved out of the automaton* *while $\$x/p2$ is retrieved in the automaton, is not supported in Raindrop. Because as mentioned in Section 2.4.3, if $\$u/p1$ is retrieved out of the automaton, then its descendant pattern $\$u/p1$ must be retrieved out of the automaton as well. We now prove that the combination in the first case is redundant. The proofs of the two other cases are similar and can be found in Appendix D.*

***Case 1:** Suppose a plan contains a $TokenNav_{\$u,p1}\$v$ and a $TokenNav_{\$x,p2}\$y$ where $\$u/p1$ is the ancestor pattern of $\$x/p2$. Changing the modes of both means we pull out both $\$u/p1$ and $\$x/p2$. However, pulling out $\$u/p1$ alone will make $\$x/p2$ to be pulled out as a second effect. For example, in Figure 3.2, pulling out $\$a/seller$ requires $\$b//profile$ to be also pulled out. Therefore, this combination generates the same plan as the combination that contains $TokenNav_{\$u,p1}\$v$ but not $TokenNav_{\$x,p2}\$y$.*

If a combination contains no pattern retrieval operators that have pattern dependency relationship with each other, we say this is a *valid* combination. Changing the modes in a valid combination must uniquely lead to a plan, regardless of the order in which we change the modes of the operators in it. Lemma 3 states this *order insensitive* property of a valid combination.

Lemma 3 ***Combinations being Order Insensitive:*** *If two pattern retrieval operators $navOp_1$ and $navOp_2$ have no pattern-dependency relationship, then regardless of the order in which we change the modes of $navOp_1$ and $navOp_2$, the two plans derived contain the same operators.*

Proof 3 *We distinguish between three cases the same as those used for proving Lemma 2. We prove the first case, i.e., given two operators $TokenNav_1$ and*

$TokenNav_2$, no matter in what order we change their modes, we get the same plans. The proof for the other two cases can be found in Appendix E.

Case 1: *Suppose $TokenNav_1$ = $TokenNav_{\$u,p}\$v$. Pulling out $\$u/p$ can eliminate the operators or introduce new operators into the plan in four ways. First, $TokenNav_{\$u,p}\$v$ and $Extract_{\$u}\$v$ are rewritten into $NodeNav_{\$u,p}\$v$. Second, if before the rewriting there exists no $Extract$ operator that extracts $\$u$, then an $Extract$ operator that extracts $\$u$ will be introduced to the plan after the rewriting. Third, the descendant patters of $\$u/p$ that are retrieved in the automaton will be pulled out. Fourth, if there exists no other operator in the format of $TokenNav_1$. $TokenNav_{\$u,p'}\$v'$ but there exists a $StructuralJoin_{\$u}$ before the rewriting, this $StructuralJoin_{\$u}$ is eliminated after the rewriting.*

Later, if we change the mode of $TokenNav_2$, we have the below observations:

1). *Mode change of $TokenNav_2$ will introduce neither a $TokenNav_{\$u,p}\$v$ nor a $Extract_{\$u}\$v$. It will not elimiate an $Extract$. Neither will it introduce a $StructuralJoin$ operator. Therefore it will not cancel out the first, second and fourth changes resulted from the mode change of $TokenNav_1$.*

2). *Since $TokenNav_2$ does not have a pattern dependency relationship with $TokenNav_1$, mode change of $TokenNav_2$ will not affect those operators whose modes have been changed as a secondary effect of the pull-out of $\$u/p$. That is to say, it will not cancel out the third change resulted from the mode change of $TokenNav_1$.*

In summary, a mode change of $TokenNav_2$ that occurs after the mode change of $TokenNav_1$ does not cancel any change that has been made. Therefore the *order in which we change the modes of* $TokenNav_1$ *and* $TokenNav_2$ *does not matter.*

Algorithm 7 Exhaustive Search

ExhaustOpt($curPlan$, $navsToBeTried$)
Input: $curPlan$ - a current plan, will be set to the initial plan when the algorithm is first called;
 $navsToBeTried$ - a list of pattern retrieval operators eligible for mode changes;
Output: the best plan in the search space

```
1: Plan $bestPlan$ = $curPlan$.
2: int $n$ = number of pattern retrieval operators in $navsToBeTried$;
3: **for** (int $i$ = 1; $i \leq n$; $i$++) **do**
4:    NavOp $curNavOp$ = $i^{th}$ operator in $navsToBeTried$;
5:    Plan $newPlan$ = copy of $curPlan$;
6:    Change mode of $curNavOp$ in $newPlan$;
7:    Optimize $newPlan$ using operator-commuting rules (see Algorithm 6);
8:    Optimize $newPlan$ using input-subplan-reordering rules;
9:    List $newNavsToBeTried$;
10:   **for** (int $j$ = $i + 1$; $j \leq n$; $j$++) **do**
11:      NavOp $newNavOp$ = $j^{th}$ operator in $navsToBeTried$;
12:      **if** $newNavOp$ and $curNavOp$ have no pattern dependency relationship **then**
13:         add $newNavOp$ into $newNavsToBeTried$;
14:      **end if**
15:   **end for**
16:   $curBestPlan$ = ExhaustOpt($newPlan$, $newNavsToBeTried$);
17:   **if** $curBestPlan$ costs less than $bestPlan$ **then**
18:      $bestPlan$ = $curBestPlan$.
19:   **end if**
20: **end for**
21: return $bestPlan$.
```

Algorithm 7 utilizes Lemmas 2 and 3 to search through the solution space. The algorithm *ExhaustOpt* takes two input parameters, namely, a plan and a list of pattern retrieval operators. The first time $ExhaustOpt$ is called, $curPlan$ is the initial plan and $navsToBeTried$ contains all the pattern retrieval operators in the initial plan. Suppose there are n operators in $navsToBeTried$. For each operator $navOp_i$ ($1 \leq i \leq n$) in the $navsToBeTried$ list, we make a copy of $curPlan$, change the mode of $navOp$ in the plan copy and then get a new plan (lines 4 - 6). We will get n new plans. We recursively apply $exhaustiveSearch$ with the input parameters $newPlan$ and $newNavsToBeTried$ (line 16). For a plan $newPlan$ that results from the mode change of $navOp_i$, we make sure that $newNavsToBeTried$ does not contain any operators that have dependency relationship with $navOp_i$ (lines 12 - 13), because changing the modes of such an operator and $navOp_i$ is forbidden by the *invalid combination lemma*.

We now illustrate this algorithm ensures that no alternative plan is missed. Given an arbitrary plan P_{any} that can be derive by changing modes of a sublist of the $navsToBeTried$ in the initial plan, it must be explored by $ExhaustOpt$. We denoted the sublist as S. Assume S = $\{navOp_k, navOp_{k+1}, ...\}$ ($1 \leq k <$ number of pattern retrieval operators in $navsToBeTried$). When $ExhaustOpt$ is called the first time, among n new plans, one results from the mode change of $navOp_k$. When $ExhaustOpt$ is recursively called on this new plan, it will change the mode of $navOp_{k+1}$. The process continues. When $ExhaustOpt$ is called the $|S|^{th}$ time ($|S|$ denotes the number of operators in S), P_{any} must be generated.

Also, the process mentioned above is the only way that $ExhaustOpt$ can generate P_{any}. Whenever we change the mode of a $navOp_i$ in the current plan and get a new plan, only operators occurring after $navOp_i$ in the $navsToBeTried$ list are added into the $newNavsToBeTried$ list (see line 10 where j starts from $i + 1$). P_{any} can be only generated when the exhaustive applies token-or-node rewrite rule in the order of $navOp_k$, $navOp_{k+1}$ and so on. Therefore P_{any} is never explored twice.

3.6 Greedy Search for One-time Optimization

For a query with n patterns, the search space can have up to $\Sigma_{i=0}^{n} C_n^i = 2^n$ alternative plans. We say “up to” because some combinations are invalid and thus excluded. Finding an optimal plan obviously will be time-consuming. In this section, we present a greedy search algorithm that aims to quickly find a good but not necessarily optimal plan.

3.6.1 Baseline Greedy Search

Figure 3.12 intuitively depicts how Greedy search works. The initial plan shown as P in Figure 3.12 has a set of pattern retrieval operators denoted as S_0. For each pattern retrieval operator in S_0, we change its mode and get a new plan, denoted as P_1, ..., P_n respectively. We use operator-commuting and input-subplan-reordering rules to further optimize the new plans (the circles on P_1, ..., P_n in Figure 3.12 denote such optimization). If the cheapest plan among the n new plans is also cheaper than the initial plan, we then select this cheapest plan as the new current plan. For example, in Figure 3.12, P_1, which results from the mode change of $navOp_1$, is selected after the first iteration (the highlighted node represents a new current plan).

With the newly selected plan, we begin the next iteration. In this iteration, we again change the mode of a set of operators, denoted as S_1, where S_1 = S_0 $-$ $navOp_1$ $-$ operators that have pattern dependency relationship with $navOp_1$. Similar to the first iteration, we optimize, cost and compare the new plans. We then get a new current plan P_{12} in Figure 3.12. The iterations continue until no new plan is found to be cheaper than the current plan, i.e., the best plan found so far. Algorithm 8 shows the pseudocode for this search process. We call this algorithm *GreedyOpt*.

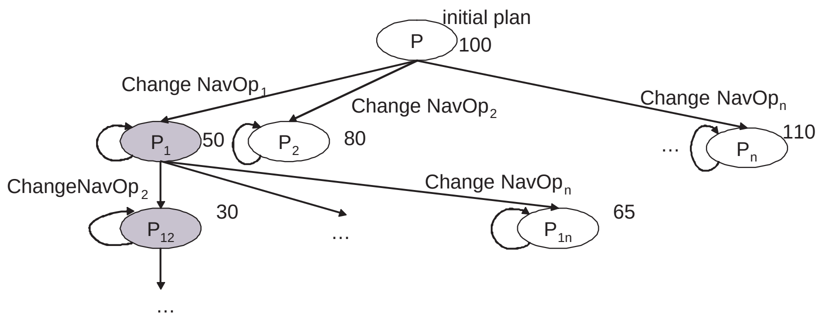


Figure 3.12: Greedy-based Search

We now compute the upper bound on the number of alternative plans explored by the GreedyOpt algorithm. In the first iteration, GreedyOpt explore n alternatives plans. In the second iteration, GreedyOpt explore at most $n - 1$ alternative plans. After at most n iterations, the process terminates. Therefore GreedyOpt explore at most $\sum_{i=1}^{n} i$ = $n(n + 1)$/2 alternative plans.

3.6.2 Expediting Cost Estimate

In the section, we propose techniques to expedite the GreedyOpt algorithm. These techniques reduce the time spent on processing an alternative plan, more specifically, costing an alternative plan. When we apply a mode change and get a new plan, we need to cost this new plan. In a naive approach, we recompute the cost from scratch. In contrast, we can analyze what parts of plan are affected by the

Algorithm 8 Greedy Search in an One-time Optimization Scenario

GreedyOpt($curPlan$, $navsToBeTried$)

Input: $curPlan$ - a current plan, will be set to the initial plan when the algorithm is first called;

$navsToBeTried$ - a list of pattern retrieval operators eligible for mode changes;

Output: the best plan among the plans explored

```
1:  Plan $bestPlan$ = $curPlan$;
2:  for each operator $navOp$ in $navsToBeTried$ do
3:      Plan $newPlan$ = copy of $curPlan$;
4:      Change mode of $navOp$ in $newPlan$;
5:      Optimize $newPlan$ using operator-commuting rules;
6:      Optimize $newPlan$ using input-subplan-reordering rules;
7:      if $newPlan$ costs less than $bestPlan$ then
8:          $bestPlan$ = $newPlan$
9:      end if
10: end for
11: if $bestPlan$ != $curPlan$ then
12:     let $navOp_i$ denotes the operator whose mode change leads to $bestPlan$;
13:     $navsToBeTried$ = $navsToBeTried - navOp_i -$ all operators that have
        pattern dependency relationship with $navOp_i$;
14:     return $GreedyOpt$($bestPlan$, $navsToBeTried$);
15: else
16:     return $curPlan$.
17: end if
```

mode change and avoid recomputing. We propose two techniques, i.e., *incremental cost estimate* and *detection of same cost change*.

Incremental Cost Estimate.

We first define several concepts needed in our analysis.

Definition 4 *For a* $NavOp_{\$u,p}\$v$*, we call* $StructuralJoin_{\$u}$ *its* context StructuralJoin *because* $StructuralJoin_{\$u}$ *joins on the context element* $\$u$ *of* $NavOp_{\$u,p}\$v$*.*

Definition 5 *The heuristics in Section 3.1 impose that* $NodeNav_{\$u,p}\$v$ *cannot be moved above a* $StructuralJoin_{\$v}$ *that can output duplicates of bindings of* $\$v$ *or is the topmost* $StructuralJoin$*. We call this* $StructuralJoin$ *a* confining StructuralJoin *of* $NavOp_{\$u,p}\$v$*.*

The confining $StructuralJoin$ confines how far the $NavOp_{\$u,p}\$v$ operator itself (when $NavOp_{\$u,p}\$v$ is a $NodeNav$) or the $TokenNav$ operator rewritten from $NavOp_{\$u,p}\$v$ (when $NavOp_{\$u,p}\$v$ is a $TokenNav$) can be moved up.

Definition 6 *We define a function* $moveScope(navOp)$ *to denote a set of* $StructuralJoins$*. The set consists of all the* $StructuralJoins$ *between the context and confining* $StructuralJoins$ *of* $navOp$*, including the context and confining* $StructuralJoins$*.*

Example 13 *In Figure 3.2, for* $TokenNav_{\$a,/seller}\$b$*,* $StructuralJoin_{\$a}$ *is its context* $StructuralJoin$*. If we change the mode of* $TokenNav_{\$a,/seller}\$b$*, the resulting* $NodeNav_{\$a,/seller}\$b$ *cannot be moved above* $StructuralJoin_{\$a}$ *because there exists an* $ExtractUnnest_{\$a}\$c$ *in the plan.* $StructuralJoin_{\$a}$ *is then also* $TokenNav_{\$a,/seller}\$b$*'s confining* $StructuralJoin$*.* $moveScope(TokenNav_{\$a,/seller}\$b)$ *thus is* $\{StructuralJoin_{\$a}\}$*.*

Suppose we change the mode of a $TokenNav_{\$u,p}\$v$ operator in the current plan $P_{current}$ and get P_{new}. We use a boolean value $isIntroduced$ to denote whether an operator in the form of $Extract_{\$t}\$u$ is introduced into P_{new} because of this change. We then have Equation 7.

Equation 7 *Cost change from a pattern pull-out*

= $Cost(P_{new}) - Cost(P_{current})$

= *automaton cost in* P_{new} $-$ *automaton cost in* $P_{current}$ *(1)*

$+$ *automaton-outside cost in* P_{new} $-$ *automaton-outside cost in* $P_{current}$ *(2)*

= $Cost(Extract_{\$t}\$u) * isIntroduced - Cost(TokenNav_{\$u,p}\$v)$ *(3)*

$+ \sum_{sj \in moveScope(TokenNav_{\$u,p}\$v)} cost\ of\ input\ subplans\ of\ sj\ in\ P_{new}$

$- \sum_{sj \in moveScope(TokenNav_{\$u,p}\$v)} cost\ of\ input\ subplans\ of\ sj\ in\ P_{current}$ *(4)*

According to Equation 6 in Section 3.2.3, a $TokenNav_{\$u,p}\$v$ operator costs the same in every alternative plan where it appears. However, changing $TokenNav_{\$u,p}\$v$ to $NodeNav_{\$u,p}\$v$ can introduce a new $Extract_{\$t}\$u$ operator if $\$u$ was not extracted in the current plan. Therefore, we expand Expression (1) into Expression (3) in Equation 6. Also, since $NodeNav_{\$u,p}\$v$ cannot be placed above the confining *StructuralJoin*, the mode change does not affect the operators beneath the confining *StructuralJoin*. We thus expand Expression (2) into Expression (4) in Equation 7. We can use Equation 7 to compute the cost change from $P_{current}$ to P_{new} when we pull out a pattern from the automaton.

Equation 8 gives the cost change that would result from a push-in of a pattern into the automaton. Equation 8 is the reverse of Equation 7. $isElimiated$ is a boolean value that indicates whether an operator in the form of $Extract_{\$t}\$u$ is eliminated from $P_{current}$ because of this change.

Equation 8 *Cost change from a pattern push-in*

$= Cost(P_{new}) - Cost(P_{current})$

$= Cost(TokenNav_{\$u,p}\$v) - Cost(Extract_{\$t}\$u) * isEliminated$

$+ \sum_{sj \in moveScope(TokenNav_{\$u,p}\$v)} cost\ of\ input\ subplans\ of\ sj\ in\ P_{new}$

$- \sum_{sj \in moveScope(TokenNav_{\$u,p}\$v)} cost\ of\ input\ subplans\ of\ sj\ in\ P_{current}$

Detection of Same Cost Change.

From Equations 7 and 8, we can derive Theorem 3.

Theorem 3 *Given two pattern retrieval operators $navOp_1$ and $navOp_2$, if $moveScope(navOp_1) \cap moveScope(navOp_2) = \emptyset$, then the cost change resulted from the mode change of $navOp_1$ in a plan is independent from the mode change of $navOp_2$ in this plan.*

The proof of Theorem 3 can be found in Appendix F. Figure 3.13 shows how to utilize Theorem 3. Given a plan P_1, we get two plans P_2 and P_3 by changing the modes of $navOp_1$ and $navOp_2$ in P_1 respectively. Suppose we now change the mode of $navOp_1$ in P_3 and get a new plan P_4. If $moveScope(navOp_1) \cap moveScope(navOp_2) = \emptyset$, we then know $Cost(P_4) - Cost(P_3)$ is the same as $Cost(P_2) - Cost(P_1)$. We can simply compute $Cost(P_4)$ as $Cost(P_3) + Cost(P_2) - Cost(P_1)$.

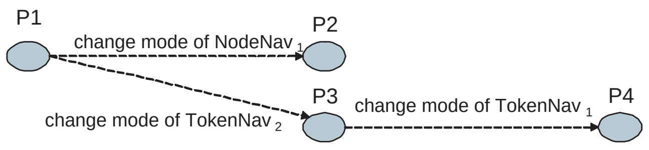


Figure 3.13: Detection of Same Cost Change: Is $Cost(P_4) - Cost(P_3) = Cost(P_2) - Cost(P_1)$?

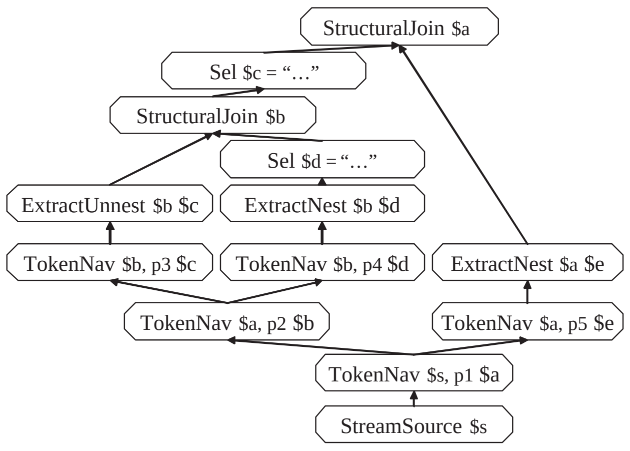
Example 14 *Suppose we have a plan in Figure 3.14 (a). This plan corresponds to P_1 in Figure 3.13. We pull out each of the four $TokenNav$ operators respectively and get four new plans. Assume we chose the plan after the pull-out of $TokenNav_{\$b,p4}\$d$, shown in Figure 3.14 (b), as the new current plan. This new current plan corresponds to P_3 in Figure 3.13. The part in Figure 3.14 (b) that is different from Figure 3.14 (a) is highlighted. To make the next move, we now need to pull out $TokenNav_{\$a,p5}\$e$ and $TokenNav_{\$b,p3}\$c$ (we do not consider the pull-out of $TokenNav_{\$a,p2}\$b$ because it has pattern dependency relationship with $TokenNav_{\$b,p4}\$d$). Thereafter, we need to estimate the costs of the two new plans.*

1). *For $TokenNav_{\$a,p5}\$e$, $moveScope(TokenNav_{\$b,p4}\$d) \cap moveScope(TokenNav_{\$a,p5}\$e)$ = $\{StructuralJoin_b\} \cap \{StructuralJoin_{\$a}\}$ = $\emptyset$. Therefore the two cost changes that the pull-out of $TokenNav_{\$a,p5}\$e$ in Figures 3.14 (a) and (b) cause respectively are the same. We can reuse the estimate of the cost change from the last time.*

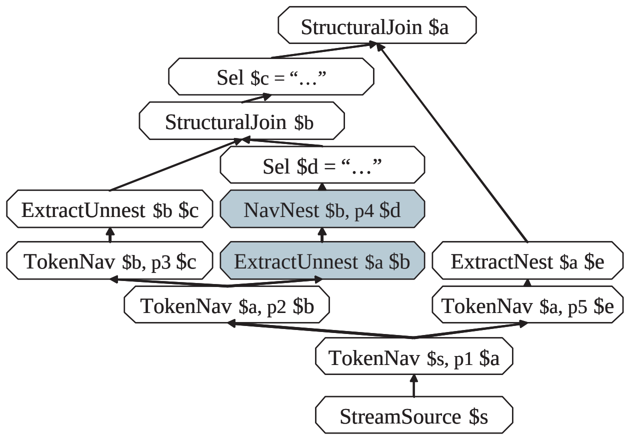
2). *In contrast, for $TokenNav_{\$b,p3}\$c$, $moveScope(TokenNav_{\$b,p4}\$d) \cap moveScope(TokenNav_{\$b,p3}\$c)$ = $\{StructuralJoin_b\} \cap \{StructuralJoin_{\$b}, StructuralJoin_{\$a}\} = \{StructuralJoin_{\$b}\} \neq \emptyset$. The two cost changes that the pull-out of $TokenNav_{\$b,p3}\$c$ in Figures 3.14 (a) and (b) cause respectively are different. We cannot reuse the estimate of cost change from last time.*

Summary

When we get a new plan, we first apply the technique of "detection of same cost change". If we find out that the cost change is not the same as estimated last time,



(a) Original Plan



(b) Moving out TokenNav$_{\$b,\ p4}$ $d from (a)

Figure 3.14: Reuse Cost Estimate for Mode Changes of Patterns in Figure 3.14 (a)

we then apply the technique of "incremental cost estimate".

3.7 Greedy Search with Pruning for Continuous Optimization

If the environment fluctuates, we have to optimize more frequently than in the one-time optimization scenario (see Section 1.3.2 in Section 1). Correspondingly, we need to find a good plan even more quickly. The plan search time is decided by two factors, i.e., number of alternative plans explored and the time spent on each alternative plan. The GreedyOpt algorithm has reduced the plan search time of the ExhaustOpt algorithm by reducing the two factors, i.e., using a Greedy search strategy and expediting costing of a plan respectively. Within the current search space that is delimited by the three rewrite rules, it is hard to further reduce the two factors in the GreedyOpt. We therefore consider dropping some rewrite rules to shrink the search space.

Among the three types of rewrite rules introduced in Section 3.1, token-or-node mode change and operator-commuting rules are more likely to affect the plan performance than the input-subplan-reordering rule. Token-or-node mode change rules enable the plans to benefit from pulling out (resp. pushing in) pattern retrieval with high (resp. low) selectivity. Operator-commuting rules enable the plans to benefit from executing operators with high selectivity after others. In contrast, plans benefit from input subplan reordering only if an input subplan of a $StructuralJoin_{\$v}$ does not generate output within a binding of $\$v$, i.e., $P_{\not\Rightarrow\emptyset}(subplan)$ = 0. This is a rather strict requirement. Therefore, we drop the input-subplan-reordering rule.

Dropping the input-subplan-reordering rule simply means we do not change the left to right order of the input subplans of a *StructuralJoin*. The execution manner of $StrucutralJoin$ given in Algorithm 5 remains unchanged. $StructuralJoin$ still first checks whether all entry operators have generated output; if yes, it then evaluates the input subplan from left to right and terminates if any input subplan does not generate output. Therefore, the cost model of input subplans does not change.

Among the three heuristics for operator commuting, one heuristic is proposed in order to provide optimization opportunities for input subplan reordering. The heuristic says that we should place *Select* or *NodeNav* operators above a $StructuralJoin_{\$v}$ that is not a topmost *StructuralJoin* and does not output tuples with duplicate bindings of $\$v$. Even though we drop the input-subplan-reordering optimization, we still keep the heuristics for two reasons.

First, a *Select* or *NodeNav* operator is still less likely to be evaluated after being commuted with its parent *StructuralJoin* operator. Dropping the input-subplan-reordering optimization only means the benefit we get from this commuting is not maximal. Second, the side effect of placing *Select* or *NodeNav* above *StructuralJoin* is small so that it would not offset the benefit of an even sub-optimal input subplan order. When we commute a *Select* or *NodeNav* with its parent $StructuralJoin_{\$v}$, the cost of $StructuralJoin_{\$v}$ increases since the *Select* or *NodeNav* could otherwise filter out the input to $StructuralJoin_{\$v}$ without the commuting. However, we only place a *Select* or *NodeNav* above a $StructuralJoin_{\$v}$ that does not output duplicate bindings of $\$v$. This means, no operator in the plan is in the format of $ExtractUnnest_{\$v}\$w$ or $NavNest_{\$v,p}\$w$. As a result, each time when an end tag of a binding of $\$v$ is encountered, $StructuralJoin_{\$v}$ has at most one tuple from each input operator (e.g., $StructuralJoin_{\$b}$ consumes at most one tuple from $ExtractNest_{\$b,//profile}$ for each binding of $\$b$). Therefore there is not much space to further reduce the cost of this $StructuralJoin_{\$v}$. In summary, dropping the input-subplan-reordering optimization does not affect the operator-commuting optimization.

For the greedy algorithm, we only need to make a slight change in order for it to apply to our new scenario. We remove the input-subplan-ordering optimization on a plan, i.e., remove line 5 in Algorithm 8. For the new greedy algorithm, we further propose a technique for pruning the alternative plans, i.e., reducing the number of alternative plans to explore. The greedy algorithm with the pruning technique guarantees to find the same plan as the greedy algorithm without pruning.

3.7.1 Basic Ideas of Pruning

Suppose we can estimate a lower bound of the cost changes from the mode changes of $navOp$ in any plans that contain $navOp$ and have been optimized by operator-commuting rule, where cost change is defined as (cost of the plan after mode change - cost of the plan). If this lower bound is larger than 0, it means that for any plan, (cost of the plan after the mode change of $navOp$ - cost of the plan) $>$ 0. In other words, mode change of this $navOp$ in any plans always leads to a worse plan. We can then safely exclude the mode change of $navOp$ in any plans.

The challenge is then how to compute the lower bound. We want the computation to satisfy two properties. First, it should be quick; otherwise the computation overhead may offset the benefits of saving time in exploring the alternative plans. Second, the lower bound computed should be "tight". For example, an extremely relaxed lower bound "negative infinite" (cost of new plan - cost of current plan must always be greater than negative infinite) will not exclude anything. Only a lower bound that is greater than 0 can help pruning alternative plans. These two properties usually have a negative correlation, i.e., we usually need to spend more time to compute a tighter lower bound. We have to strike a balance between the time spent on computing the lower bound and the quality of the lower bound.

3.7.2 Pruning Plans Derived from Mode Change of TokenNav Operators

We first consider the case in which $navOp$ is a $TokenNav_{\$u,p}\$v$ whose $\$v$ is not selected by any $Select$ or navigated into by any $NodeNav$. When we pull out such a $TokenNav_{\$u,p}\$v$ in a current plan $P_{current}$ and get a new plan P_{new}, then no other operator would have to be moved so that they are still placed above $TokenNav_{\$u,p}\$v$. This leads to Equation 9.

Equation 9 ***Cost change of changing mode of*** $TokenNav_{\$u,p}\$v$ ***with*** $\$v$ ***not being consumed by other operators:***

Cost(P_{new}) − *Cost*$(P_{current})$*:*

= automaton cost in P_{new} − *automaton cost in* $P_{current}$ *(1)*

$+$ *automaton-outside cost in* P_{new} − *automaton-outside cost in* $P_{current}$ *(2)*

= $Cost(Extract_{\$t}\$u)$ * $isIntroduced$ − $Cost(TokenNav_{\$u,p}\$v)$ *(3)*

$+$ $Cost(NodeNav_{\$u,p}\$v)$ *(4)*

− $Cost(StructuralJoin_{\$u}$ *in* $P_{current})$ * $isEliminated$ *(5)*

$+$ *cost of rest automaton-outside operators in* P_{new} − *cost of rest automaton-outside operators in* $P_{current}$ *(6)*

Expressions (1), (2) and (3) are the same as Equation 7. Expression (1) is expanded into Expression (3). Expression (2) is expanded into Expressions (4) and (5). For Expression (4), depending on $NodeNav_{\$u,p}\$v$'s descendent operators,

$Cost(NodeNav_{\$u,p}\$v)$ can vary in different current plans. $Cost(NodeNav_{\$u,p}\$v)$ is minimal when $NodeNav_{\$u,p}\$v$ is executed as late as possible. In such cases it consumes the least input and thus costs the least. We denote this minimal cost as $min(Cost(NodeNav_{\$u,p}\$v))$. Therefore Expression (4) $> min(Cost(NodeNav_{\$u,p}\$v))$.

We now analyze the lower bound of Expression (5) in Equation 9. Changing the mode of $TokenNav_{\$u,p}\$v$ can lead to the elimination of $StructuralJoin_{\$u}$. This can happen in only one case. That is, when $StructuralJoin_{\$u}$ in the current plan has only two input subplans according to the *mode change with introducing/eliminating StructuralJoin* rewrite rule in Figure 2.6. Therefore Expression (5) $> -Cost(StructuralJoin_{\$u})$.

Except the possibly eliminated $StructuralJoin_{\$u}$, all the other automaton-outside operators in $P_{current}$ remain in P_{new}. Also, the rank of each such automaton-outside operator op, i.e., $\frac{\sigma(op)-1}{UnitCost(op)}$, is completely decided by the op itself. It is not changed by the newly created $NodeNav_{\$u,p}\$v$. Therefore commuting these automaton-outside operators with each other is not needed. However, rewriting $TokenNav_{\$u,p}\$v$ to $NodeNav_{\$u,p}\$v$ can increase the cost of those automaton-outside operators which are executed after $TokenNav_{\$u,p}\$v$ in $P_{current}$ but are executed before $NodeNav_{\$u,p}\$v$ in P_{new}. Therefore, Expression (6) $\geq$ 0.

Based on the above discussion, we have Equation 7 $> -Cost(TokenNav_{\$u,p}\$v)$ $+ min(Cost(NodeNav_{\$u,p}\$v)) - Cost(StructuralJoin_{\$u})$. Correspondingly, we have Pruning Rule 1.

Pruning Rule 1 *Given a pattern $\$v = \$u/p$ where $\$v$ is not further selected by $Select$ operators or navigated into by $NodeNav$ operators, if $min(Cost(NodeNav_{\$u,p}\$v))$ $- Cost(TokenNav_{\$u,p}\$v) - Cost(StructuralJoin_{\$u}) >$ 0, we do not consider* *to change the mode of $TokenNav_{\$u,p}$ in any alternative plan.*

3.7.3 Discussion on Pruning Other Pattern Retrieval Operators

We now discuss $TokenNav_{\$u,p}\$v$ operators whose bindings of $\$v$ are further selected on or navigated into. To get the lower bound of (cost of P_{new} - cost of $P_{current}$) for $TokenNav_{\$u,p}\$v$, we have to estimate the lower bound of those operators that consume $\$v$ in P_{new} by assuming they are executed as late as possible; and the upper bound of these operators in $P_{current}$ by assuming they are executed as early as possible. Doing this can be quite time-consuming. We therefore do not consider pruning by bounding the cost of the mode change of $TokenNav_{\$u,p}\$v$ whose $\$v$ is further consumed.

The cost change that results from the mode change of $NodeNav_{\$u,p}\$v$ whose bindings of $\$v$ are not consumed by other operators is the reverse of Equation 9.

Equation 10 ***Cost change of changing mode of*** $NodeNav_{\$u,p}\$v$ ***with*** $\$v$ ***not being consumed by other operators:***

Cost$(P_{new}) -$ *Cost*$(P_{current})$*:*

= $Cost(TokenNav_{\$u,p}\$v)$ *-* $Cost(Extract_{\$t}\$u)$ *** $isIntroduced$ *(1)*

$- Cost(NodeNav_{\$u,p}\$v)$ *(2)*

$+ Cost(StructuralJoin_{\$u})$ *in* $P_{current}$ *** $isEliminated$ *(3)*

$-$ *cost of rest automaton-outside operators in* P_{new} $+$ *cost of rest automaton-outside operators in* $P_{current}$ *(4)*

Since Expressions (3) and (6) in Equation 9 both are greater than some constants, Expressions (1) and (4) in Equation 10 are then less than these constants. It is difficult to get a lower bound for this cost change. We therefore do not develop a pruning rule for bounding the cost change caused by pushing in a node pattern retrieval.

3.7.4 Summary

Algorithm 9, which is called $greedyPruneOpt$, shows the greedy search with pruning for the continuous optimization scenario. Each time when we start the optimization, we call $greedyPruneOpt$ with three parameters, $curPlan$ which is the currently running plan, $navsToBeTried$ which is a list of pattern retrieval operators in $curPlan$, and a boolean value $TRUE$ to indicate that this is the first iteration of the optimization on $curPlan$. During the first iteration of the optimization, we apply the technique of "pruning by bounding cost change" (lines 1 - 8 in Algorithm 9). We bound the cost change for each $TokenNav$ whose pattern is not further consumed in the plan. Those operators whose lower bound is greater than 0 are excluded from mode changes. With the rest pattern retrieval operators, we then apply greedy search as before (lines 9 - 16 in Algorithm 9).

3.8 Embedding Statistics Collection into Plan Execution

We now analyze what statistics need to be collected to estimate the costs of the plans. Tables 3.1 and 3.2 in Section 3.2.3 contain the parameters needed for costing the automaton. Some of the parameters, such as C_{nonEmp}, C_{emp}, C_{visit} and $C_{bicartesian}$[3] are constants. We can determine their values off-line, i.e., before the data comes in. The other parameters, namely, $C_{extract}(q)$, $n_{active}(q)$, n_{start}, $n_{p[i]}$ and $w_{p[i]}$ vary in different data and need to be collected on-line, i.e., as the data

[3]The parameter $C_{backtrack}$ in Table 3.2 is only used for analysis, but not needed in Equation 6 which computes the cost of a $TokenNav$ operator. We therefore do not collect it.

Algorithm 9 Greedy Search with Pruning in a Continuous Optimization Scenario

GreedyPruneOpt($curPlan$, $navsToBeTried$, $isInitial$)

Input: $curPlan$ - a current plan, will be set to the initial plan when the algorithm is first called;

$navsToBeTried$ - a list of pattern retrieval operators eligible for mode changes;

$isInitial$ - a boolean indicating whether $curPlan$ is the initial plan;

Output: the best plan among the plans explored

```
1: if isInitial then
2:   for each operator $navOp$ in $navsToBeTried$ that satisfies: (1) $navOp$ is a
     $TokenNav$ and (2) the pattern retrieved by $navOp$ is not further consumed
     do
3:     double $lowerBound$ = estimate lower bound of the cost cut of mode
       change on $tokenNavOp$;
4:     if $lowerBound > 0$ then
5:       remove $tokenNavOp$ from $navsToBeTried$
6:     end if
7:   end for
8: end if
9: ... (same as lines 1 - 7 in Algorithm 8)
10: if $bestNewPlan$ costs less than $curPlan$ then
11:   let $navOp_i$ denote the operator whose mode change leads to $bestNewPlan$;
12:   $navsToBeTried$ = $navsToBeTried - navOp_i -$ all operators that have
      pattern dependency relationship with $NavOp_i$;
13:   return $GreedyPruneOpt$($bestNewPlan$, $navsToBeTried$, $FALSE$);
14: else
15:   return $curPlan$.
16: end if
```

comes in. Also, $\sigma(op)$, $P_{op \not\Rightarrow \emptyset}$ and $UnitCost(op)$ are required in Equation 3 in Section 3.2.1 for costing the automaton-outside operators.

Some parameters can be derived from the others. For example, $n_{p[i]}$ and $w_{p[i]}$ are used to estimate the cost of a $NodeNav_{\$u,p}\$v$ while $n_{active}(q)$ is used to estimate the cost of a $TokenNav_{\$u,p}\$v$. $n_{p[i]} \times w_{p[i]}$ gives the number of children of the bindings of $p[i]$ (i.e., the i^{th} navigation step on p) in a binding of $\$u$. Suppose states q and q' in the automaton are activated by bindings of $p[i]$ and binding of $\$u$ respectively. $n_{active}(q)$ is the number of children of bindings of $p_[i]$ in a bottom input element (see Table 3.2) . We then have, $\frac{n_{active}(q)}{number\ of\ bindings\ of\ \$u\ in\ a\ bottom\ input\ element}$ = $\sigma_{p[i]} \times w_{p[i]}$. Therefore we need only collect either $n_{active}(q)$ when $\$u/p$ is retrieved in the automaton; or $n_{p[i]}$ and $w_{p[i]}$ when $\$u/p$ is retrieved out of the automaton.

We now briefly introduce how we collect each required parameter:

1. $n_{active}(q)$: $n_{active}(q)$ is the number of times that stack top contains a state q when a start tag arrives. For each state q in the automaton, we maintain a counter denoted as $activeCounter(q)$. Each time when a start tag arrives, this counter of each state at the top of the stack is incremented by 1. Also, for a state that corresponds to the start of a path (e.g., q_2 in the automaton in Figure 3.2), we associate it with a second counter denoted as $reachCounter(q)$. $reachCounter(q)$ is incremented by 1 each time when q is pushed into the stack. For example, in Figure 3.2, when a start tag of a descendant of ***bidder*** elements arrives, the stack top always contains a q_8 so that $activeCount(q_8)$ is incremented. When a $<auction>$ arrives, it activates q_2 and $reachCount(q_2)$ is incremented. $n_{active}(q_8)$, i.e., the number of descendants of $bidder$ in an $auction$, is then equivalent to $\frac{activeCount(q_8)}{reachCount(q_2)}$.

2. $C_{extract}(q)$: To find out the cost of storing elements whose start tags activate

state q, we maintain a storing cost counter denoted as $storeCount(q)$. Also, the storage manager maintains a list. We can add a storing cost counter into the list or remove one from the list. Each time when q is activated, we add $storeCount(q)$ into the list. Whenever the storage manager stores a token, it traverses this list. For each storing cost counter in the list, the storage manager increments it by the length of the token. Later, when q is popped off the stack, we remove its storing cost counter from the list. At this time, the value of $storeCount(q)$ is the length of the element that activates q.

3. $P_{op \not\Rightarrow \emptyset}$: Assume $StructuralJoin_{\$v}$ is *op*'s nearest ancestor $StructuralJoin$. $notEmptyCount(op)$ is the probability of *op* generating some output within a binding of $\$v$. We associate *op* with a counter, denoted as $notEmptyCount(op)$. Each time when $StructuralJoin_{\$v}$ invokes *op* as an end tag of a binding of $\$v$ arrives, $notEmptyCount(op)$ is incremented by 1 if *op* generates some output. Suppose bindings of $\$v$ activate automaton state q, then at any time when a binding of $\$v$ has been finished processing, $\frac{notEmptyCount(op)}{activateCount(q)}$ gives $P_{op \not\Rightarrow \emptyset}$.

The collection of $\sigma(op)$ (selectivity of an operator), $UnitCost(op)$ (cost of processing one input tuple) and n_{start} (number of start tags in a bottom input element) is rather straightforward. We skip the discussion here.

3.9 Run-time Plan Migration

In the compile time optimization, *plan migration* is not needed. We optimize, get a plan and simply run it. In the run time optimization in our scenario, we optimize a currently running plan, get a new plan (if any), and then have to migrate the current plan to this new plan. Two problems arise. First, how to change the current place to the new plan. This process must be efficient, especially in the continuous optimization scenario since plan change happens from time to time. Second, we need to determine when to change the current plan to the new plan. The migration should take place as soon as possible so that we can benefit from the new plan as early as possible. We now address these two aspects in Sections 3.9.1 and 3.9.2 respectively.

3.9.1 Incremental Change of Automaton

The search algorithm returns a new query plan. However this plan is not ready for execution. We must traverse the *TokenNav* operators in the new plan and construct an automaton out of it. For example, the plan search algorithm may return the top query plan in Figure 3.2 as the new plan found. We then need to construct the bottom automaton in Figure 3.2 before the plan can be executed.

We actually do not have to reconstruct the automaton from scratch. We can modify the automaton for the currently running plan and reuse it for the new plan. Besides a new plan, the search algorithm returns a list of *NodeNav* and *TokenNav* operators in the current plan whose modes have been changed. For each operator in the list, if the operator is a $TokenNav_{\$u,p}\$v$, we remove the states that encode p in the current automaton; if a $NodeNav_{\$u,p}\$v$ has been pushed in, we add states to the current automaton to encode p.

For example, suppose we want to migrate the currently running plan in Figure 3.2 to the new plan in Figure 3.6. The mode of $TokenNav_{\$a,/bidder}\$c$ in the current plan is changed. Correspondingly, as shown in Figure 3.15, we remove the transition from q_2 to q_4 in the automaton in Figure 3.2. We still maintain the disconnected sub-automaton composed of states q_4, q_5 and q_6 which encodes

$TokenNav_{\$a,/seller}\$b$. Next time, if the mode of $NodeNav_{\$a,/seller}\$b$ in Figure 3.6 is changed, we can simply add the sub-automaton encoding $TokenNav_{\$a,/seller}\$b$ in, namely, we add the transition from q_2 to q_4 without creating any new states.

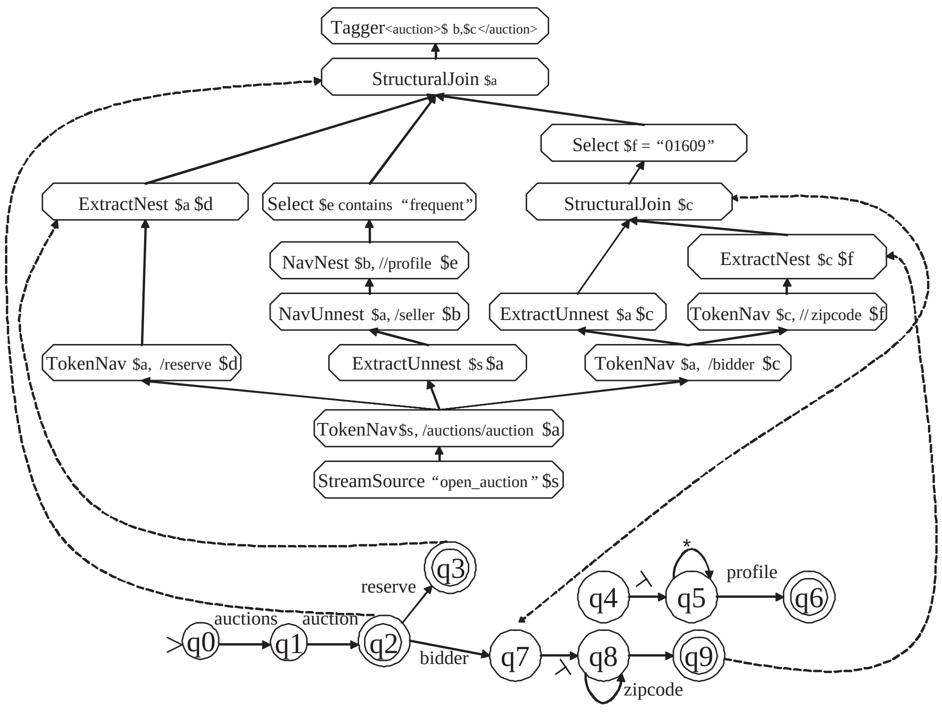


Figure 3.15: Incremental Change of Automaton for Migrating from Plan in Figure 3.2 to Plan in Figure 3.6

We now have the automaton for the new plan. The next thing to do is then to associate the automaton with the operators in the new plan. Otherwise, after the migration, when a state is activated, the operators in the current plan, instead of the operators in the new plan, will be executed. Therefore, for an automaton state that is associated with operators in the current plan, we redirect it to be associated with the matching operators in the new plan. An operator *op* in the current plan is matched with another operator op' in the new plan if op' is a copy of op. In Figure 3.15, four states in the automaton, i.e., q_2, q_3, q_7 and q_9, are redirected to be associated with the operators in the new plan. For example, the association between q_2 and $StructuralJoin_{\$a}$ means once q_2 is popped off the stack, $StructuralJoin_{\$a}$ will be invoked.

Note that recording the matching relationship, i.e., remembering an operator in the new plan is copied from a certain operator in the current plan, is not an extra burden in the plan search algorithm. Even if we do not incrementally change the automaton, the plan search algorithm still has to record the matching relationship. Otherwise, after we copy the current plan and rewrite the copy, we have no way to cost the rewritten plan since the statistics are collected for the operators in the current plan.

3.9.2 Choosing Right Moment to Migrate

A challenge in plan migration is that the migration cannot just start at a random time, as this may corrupt the running system. The example below illustrates how corruption may arise.

Example 15 *Suppose we are running a plan in Figure 3.2. Figure 3.11 in Section 3.2.3 shows the snapshots of the stack content as the tokens are processed. Assume we now pause this plan right after we have processed a $<seller>$ token and start to migrate to the new plan in Figure 3.6. The last stack in Figure 3.11 is the current stack at this moment. Since in the new plan, $\$b = \$a/seller$ is retrieved out of the automaton, the corresponding automaton of the new plan will not have states q_4, q_5 and q_6 as the current automaton in Figure 3.2 does. After the migration, for the* *next incoming start tag, the transition entry of the state at the stack top, i.e.,* q_4 *and* q_5*, would be looked up. However* q_4 *and* q_5 *are no longer in the automaton. This makes the system corrupt.*

We now characterize the safe moment to start the migration. Suppose a new plan is derived from the current plan by mode changes of a set of pattern retrieval operators denoted as S. We define a set T as: T = {Confining $StructuralJoin$ of $navOp$ | $navOp \in S$}, where confining $StructuralJoin$ of $navOp$ is the $StrucutralJoin$ beyond which $navOp$ cannot be moved as defined in Section 3.6.2. We call T *boundary StructuralJoins* because only the subplans underneath these *StructuralJoins* are changed. We call the time that the tokens under processing are not components of any binding of $\$v$ that is joined on by any boundary $StructuralJoin$ (i.e., $\$v$ satisfies: there exists a $StructuralJoin_{\$v}$ in T) the *migration window*. The migration can start within the *migration window*.

Example 16 *In Example 15, the plan in Figure 3.6 is rewritten from the plan in Figure 3.2 with* S = $\{TokenNav_{\$a,/seller}\}.Correspondinly$,$T$=$\{StructuralJoin_{\$a}\}$. *The migration can start whenever the current query plan is not in the middle of processing any component tokens of a binding of* $\$a$ *(i.e., an* auction *element). For example, the migration can start right after a* $</auction>$ *has been processed.*

We cannot start the migration any time earlier than the migration window. Otherwise we can lose data. For example, suppose we start the migration in the middle of processing component tokens of an *auction* element, say, right after we have processed a $</seller>$. At this moment, the output buffer of $StructuralJoin_{\$b}$ in Figure 3.2 contains tuples each of which has two cells, one for the binding of $\$b$ and one for the binding of $\$e$. However after the migration, $StructuralJoin_{\$b}$ is gone.

Note that we cannot move the tuples in the output buffer of $StructuralJoin_{\$b}$ to the output buffer of $NavNest_{\$b,//profile}\$e$ in Figure 3.6, because semantically, each output tuple of $NavNest_{\$b,//profile}\$e$ should contain three cells, for bindings of $\$a$, $\$b$ and $\$e$ respectively. If we simply discard the tuples in the output buffer of $StructuralJoin_{\$b}$, we then lose data.

Allowing the migration to start anytime in the migration window has impact on our migration strategy. Because the subplans that are not underneath any boundary confining *StructuralJoin* operators may have tuples in their output buffers, during the migration, we must redirect these output buffers to be associated with the operators in the new plan. This redirecting process is cheap. We simply set the output buffers of these operators in the current plan to be the output buffers of the matching operators in the new plan.

Why migrating within the migration window ensures the correctness is twofold. First, no intermediate result that is not consumed yet when the migration starts will be consumed by a different set of ancestor operators after the migration compared to before the migration. Within the migration window, the query plan is not processing any bindings of $\$v$ that a boundary $StructuralJoin$ joins on. The subplans underneath a boundary $StructuralJoin$ in the format of $StructuralJoin_{\$v}$ can generate output only when the token under processing is a component of a binding of $\$v$. Since the migration window excludes the time whenever the tokens under processing are components of bindings of such $\$v$, there must be no unconsumed result in the subplans underneath these boundary $StructuralJoins$ when the migration starts. In other words, any intermediate results unconsumed when the plan migration starts must only stay in the output buffers of those subplans which remain unchanged in the new plan. All unconsumed result generated before the plan migration will be consumed in the same manner as it is before the migration.

Second, suppose the mode of a $TokenNav_{\$v1,p}\$v2$, whose confining *StructuralJoin* is $StructuralJoin_{\$v}$, is changed. We should only remove the sub-automaton encoding the path p when the states in the sub-automaton are not in the stack. These states can only be in the stack when a binding of $\$v1$ is being processed. A binding of $\$v1$ must be part of a binding of $\$v$. For example, in Example 16, $StructuralJoin_{\$a}$ is the confining *StructuralJoin* of $TokenNav_{\$b,//profile}\$e$ and $TokenNav_{\$c,//zipcode}\$f$. Bindings of $\$b$ and $\$c$ are both child elements of a binding of $\$a$. If we pause the automaton when the element under processing is not a binding of $\$v$, the element under processing cannot be a binding of $\$v1$ either. Therefore we can safely modify the automaton without worrying about whether some states have been removed from the automaton during the migration would still remain in the stack after the migration. The situation described in Example 15 thus will not arise.

3.10 Experimental Evaluation

We have incorporated the run-time optimization techniques into the *Raindrop* framework. We run the 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 would then process the received data. We count the processing time of a token from the arrival time of the token on the second machine to the time the processing on the token has been finished. The execution time of a plan on the stream is the summation of the time spent on each token in the stream.

3.10.1 Getting Constant Values

As we have mentioned in Section 3.8, we need to get the values of the four constants C_{nonEmp}, C_{emp}, $C_{backtrack}$ and $C_{bicartesian}$. C_{nonEmp} and C_{emp} are used to evaluate the cost of a $TokenNav$ operator (see Equation 6). C_{visit} and $C_{bicartesian}$ are used to evaluate the cost of a $NodeNav$ and a $StructuralJoin$ operator respectively (see Equations 1 and 2 Section 3.2.1).

In the first experiment, we design an XML document whose root element has a tag name "root". The root element contains n children with tag name "a". Each element a does not have any child elements. This stream thus contains n + 1 start tokens, i..e, one $<root>$ and n $<a>$'s. We also design a query "/root/a". We construct a plan for this query which retrieves the pattern "/root/a" on the tokens. During the processing of the stream, when the $<root>$ is encountered, the stack top must contain an initial state of the automaton. $<root>$ matches the first navigation step "/root" and pushes a state into the stack. Next, whenever a $<a>$ is encountered, the stack top must be non-empty. Therefore each time when a start token is encountered, the stack top is always not empty. Later, we divide the execution time spent on start tokens in the stream by n + 1 and get C_{nonEmp}.

In the second experiment, we use the same XML stream and same query. However, we construct a different plan which first extracts the stream into an XML element tree and then evaluates a $NodeNav$ operator on the tree. This $NodeNav$ operator visits every node in the tree to retrieve the pattern "/root/a". We divide the execution time by n + 1 and get C_{visit}.

We also issue a query "/b" on the XML stream used in the first two experiments. During the processing of the stream, $<root>$ does not match "/b" and

Notation	Explanation	Value
C_{nonEmp}	average time of processing a start token when stack top is not empty	1.361 * 10^{-3} ms
C_{emp}	average time of processing a start token when stack top is empty	0.779 * 10^{-3} ms
C_{visit}	average time of visiting one node in an XML element tree	1.622 * 10^{-3} ms
$C_{bicartesian}$	average time of performing a binary cartesian product on one input tuple from either side to generate an output tuple	3.012 * 10^{-3} ms

Table 3.3: Values of Constant Parameters in Cost Model

correspondingly an empty set is pushed onto the stack. Next, whenever any of the n $<a>$ tokens is encountered, the stack top is empty. We divide the execution time spent on the start tokens in the stream by n and get C_{emp}.

To evaluate $C_{bicartesian}$, we simply run a query that involves a binary *StructuralJoin* operator. We divide the time spent on this *StructuralJoin* by the number of the cartesian product of its input tuples to get $C_{bicartesian}$. Table 3.3 gives these constant values.

3.10.2 Experiment Design for Comparing ExhaustOpt and GreedyOpt Search Strategies

Sections 3.5 and 3.6 propose an exhaustive and a greedy search algorithm, namely, *ExhaustOpt* and *GreedyOpt*. We now compare them in two aspects. First, we compare the optimization time, i.e., the time the algorithms spend on finding plans. Second, we compare the quality of the plans found by the algorithms, i.e., the execution time of the plans.

We test various queries conforming to four classes of pattern trees shown in Figure 3.16. Previous work on XQuery optimization has experimented with queries of similar structures [36, 58, 82]. In our pattern tree, a node represents an XML

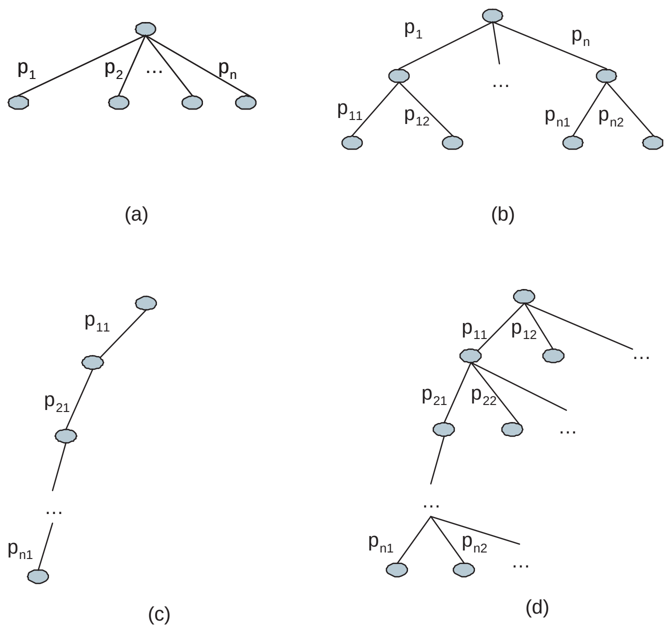


Figure 3.16: Pattern Tree Templates: (a) wide and simple; (b) wide and complex; (c) deep and simple; (d) deep and complex

element. The top node in the pattern tree represents the bottom input element. The label on the edge between a parent node u and a child node v denotes an XPath p, indicating there must exist descendent elements that are accessible via p from the element represented by u. We now describe the characteristics of each pattern tree.

1). Figure 3.16 (a) depicts a wide pattern tree. The bottom input element in the pattern tree contains paths p_1, p_2, ..., p_n. Each path is in the format of n_{11}/n_{12}/.../$n_{1i}[filter?]$ where n_{11}, n_{12}, ..., n_{1i} are element node tests and $[filter?]$ denotes an optional filter such as "/text() $>$ 100". We also say this tree is simple because only one node has more than one child node. In a plan that retrieves all patterns in the automaton, to retrieve an element node that has multiple child nodes, a $StructuralJoin$ will be performed to check whether an element contains all the required child elements. Therefore, a plan for the query in Figure 3.16 (a) contains at most one $StructuralJoin$.

2). Retrieving an XML element that has more than one child in the automaton requires one $StructuralJoin$. In contrast the wide pattern tree in Figure 3.16 (a) that requires only one $StructuralJoin$, the wide pattern tree in Figure 3.16 (b) is more complex in the sense that it involves more $StructuralJoin$ operators.

3). Figure 3.16 (c) depicts a deep tree. Small linear patterns are chained together into one larger linear pattern. $StructuralJoin$, which glues linear patterns into tree patterns, is not needed here. We therefore say this tree is simple.

4). In contrast to Figure 3.16 (c), Figure 3.16 (d) depicts a deep and complex pattern tree. n nodes in the tree have multiple children and thus there can be at most n $StructuralJoins$ in a Raindrop plan.

3.10.3 Comparing ExhaustOpt and GreedyOpt on Wide-and-Simple Pattern Trees

A pattern tree represents a class of queries. These queries locate the same patterns but return different subsets of retrieved patterns as the query results. For example, Figure 3.17 shows two query templates that both conform to the wide and simple pattern tree in Figure 3.16 (a). Query template (1) asks to return the bottom input element, i.e., $\$v$. All alternative plans of this query, no matter what patterns are retrieved in or out of the automaton, have to extract the same amount of data, i.e., bindings of $\$v$. Query template (2) asks to return $\$v1$ ($\$v1$ = $\$v/p_0$). Different plans can extract different amount of data. For example, a plan that retrieves p_1 out of the automaton still has to extract the bindings of $\$v$ into element nodes. In contrast, a plan that retrieves all the patterns in the automaton only needs to extract the bindings of $\$v1$. For easy reference, we call Figures 3.17 (1) and (2) the *extract-same* and *extract-different* queries respectively.

When comparing the alternative plans for extract-same queries, the accuracy of costing of $Extract$ operators is not important. This is because all alternative plans extract the same amount of data and thus cost the same on the $Extract$ operators no matter how inaccurately $Extract$ operators are costed. In contrast, the accuracy of costing of $Extract$ operators is important for comparing the alternative plans for extract-different queries. In order to test the accuracy of costing of every kind of operator, we study $ExhasutOpt$ and $GreedyOpt$ on both extract-same and extract-different queries.

for \$v in $p_0[p_1][p_2]\ldots[p_n]$
return \$v

(1)

for \$v in $p_0[p_1][p_2]\ldots[p_n]$
let \$v1 := \$v/p_1
return \$v1

(2)

Figure 3.17: Extract-Same and Extract-Different Queries Sharing Wide-and-Simple Pattern Tree in Figure 3.16 (a)

Testing Extract-Same Queries

Query Sets: We generate three queries that conform to the template in Figure 3.17 (1). These three queries differ in the number of patterns in the query, i.e., the value of n in Figure 3.17 (a). The values of n in the three queries are 5, 10 and 20 respectively.

Data Sets: We modify the DTD provided by the XML benchmark XMark [7]. We add more child elements to some elements in the XMark DTD so that we are able to issue queries that contain 20 patterns. We use ToXGene [24] to generate two XML streams conforming to the modified DTD. The size of each stream is around 52M. In XML stream 1, for any of the three queries, 4/5 of the patterns have a selectivity of 10% while 1/5 of the patterns have a selectivity of 90%. In XML stream 2, 1/5 of the patterns have a selectivity of 10% while 4/5 of the patterns have a selectivity of 90%.

The purpose of designing these two streams is to test the *ExhaustOpt* and *GreedyOpt* in the extreme cases. In XML stream 1, most pattern retrieval operators have a low selectivity. Pattern retrieval operators in the automaton are executed before those out of the automaton. The pattern retrieval operators that have low selectivities 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 XML 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. A lot of changes need to be made to the initial plan to get the final plan.

We now use *ExhaustOpt* and *GreedyOpt* to generate plans for the three queries on both streams 1 and 2. We run an initial plan that retrieves all patterns in the automaton on the stream, 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 3.4 reports the result.

The patterns p_1, ..., p_n in Figure 3.17 (1) are all siblings. Therefore any combinations among p_1, ..., p_n are valid (combinations of ancestor-descendant patterns are invalid according to Lemma 2 in Section 3.5). The number of alternative plans explored in *ExhaustOpt* is then 2^n. We can see that when n = 10, the optimization time already far exceeds the execution time on both XML streams 1 and 2 (Rows 2 and 5 in Table 3.4). When n = 20, the number of alternative plans explored by *ExhaustOpt* explodes and makes *ExhaustOpt* obviously impractical. Hence we do not report it here.

The number of plans explored by *ExhaustOpt* is fixed given a query. That is why *ExhaustOpt* explores 32 and 1024 plans on both XML streams 1 and 2 when n = 5 and 10 respectively. In contrast, the number of plans explored by *GreedyOpt* can vary with different streams. For the same query, *GreedyOpt* on XML stream 1 explores less plans than on XML stream 2. This is because *GreedyOpt* terminates when no mode change of a pattern retrieval in the current plan yields a better plan.

Although *GreedyOpt* explores much less plans than *ExhaustOpt*, it generates

	n	ExhaustOpt			GreedyOpt			Initial Plan	Search Time of *ExhaustOpt* + Exec. Time of Opt. Plan/	Search Time of *GreedyOpt* + Exec. Time of Opt. Plan/
		# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	# of plans explored	Opt. Time (ms.)	Exec. Time (ms)	Exec. Time (ms)	Exec. Time of Initial Plan	Exec. Time of Initial Plan
Stream 1	5	32	592	1543	9	225	1543	1821	117%	96%
	10	1024	15921	5439	27	532	5439	6349	336%	94%
	20	∞	∞	N/A	144	2072	9402	12468	N/A	92%
Stream 2	5	32	508	3987	15	245	3987	5340	84%	79%
	10	1024	14982	9283	54	821	9283	14611	166%	69%
	20	∞	∞	N/A	204	2978	22271	36841	N/A	68%

Table 3.4: *ExhaustOpt* and *GreedyOpt* for Extra-Same Queries in Figure 3.17 (1)

the same plan as *ExhaustOpt* (compare the columns of “Plan Exec. Time” in $ExhaustOpt$ with that in $GreedyOpt$. *GreedyOpt* succeeds to final optimal plans in all cases in this experiment setting.

The last two columns in Table 3.4 summarize the “effectiveness” of both *ExhaustOpt* and *GreedyOpt*. We define “effectiveness” of a search strategy as (the time spent on finding a plan + the time spent on executing the plan found)/(the time spent on executing the initial plan). The less the number is (i.e., spent less time on finding a plan that runs faster), the more effective the search algorithm is. *GreedyOpt* is more effective in stream 2 than in stream 1. This is because in stream 1, the initial plan is close to the optimal plan while in stream 2, the initial plan is significantly worse than the optimal plan.

Testing Extract-Different Queries

We now evaluate the extract-different queries conforming to the template (2) in Figure 3.17 on the two XML streams. Alternative plans of an extract-different query extract different amount of data. Table 3.5 shows the result. Again, for the three queries on both streams, *GreedyOpt* finds the same plan as *ExhaustOpt* but in much less time than *ExhaustOpt*. In Stream 1, the initial plan itself is the optimal plan. The plan search is a pure overhead. However, when n = 10 or 20, the overhead is ignorable, taking 4% or 3% of the overall execution time respectively. In stream 2, the plan found by *GreedyOpt* cuts down the execution time of the initial plan by 20% to 40%.

	n	ExhaustOpt			GreedyOpt			Initial Plan	Search Time of *ExhaustOpt* + Exec. Time of Opt. Plan/	Search Time of *GreedyOpt* + Exec. Time of Opt. Plan/
		# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	Exec. Time (ms.)	Exec. Time of Initial Plan	Exec. Time of Initial Plan
Stream 1	5	32	502	1248	5	152	1248	1248	140%	112%
	10	1024	15306	5042	10	225	5042	5042	403%	104%
	20	∞	∞	N/A	20	302	9021	9021	N/A	103%
Stream 2	5	32	516	3902	15	206	3907	5165	86%	79%
	10	1024	14120	9123	54	811	9315	15059	154%	67%
	20	∞	∞	N/A	204	3104	22197	35981	N/A	62%

Table 3.5: *ExhaustOpt* and *GreedyOpt* on Extract-Different Queries in Figure 3.17 (2)

3.10.4 Comparing ExhaustOpt and GreedyOpt on Wide-and-Complex Pattern Trees

We now compare the *ExhaustOpt* and *GreedyOpt* for wide-and-complex queries conforming to the template in Figure 3.16 (b). Our experiments consist of two parts. In the first part, we test on a set of data streams with varying data characteristics. The purpose is to observe how *GreedyOpt* behaves on relatively "random" data sets. In the second part, we focus on studying when *GreedyOpt* fails to find the optimal plans.

We generate XML streams conforming to the DTD describing Ebay's auction data from University of Washington's XML repository [60]. The root element contains a sequence of $listing$ child elements. The DTD of a $listing$ element is as follows: $<!ELEMENT\ listing$ ($seller\_info$, $payment\_types$, $shipping\_info$, $buyer\_protection\_info$, $auction\_info$, $bid\_history$, $item\_info$)$>$. Among the seven child elements of $listing$, four of them (e.g., $seller\_info$ and $auction\_info$) have nested structures, i.e., they can have children again. We design a query, shown in Figure 3.18, which navigates into each nested element. For each nested element, we pose a filter on each of its child elements. More specifically, $\$b$, $\$c$, $\$d$ and $\$e$ have 2, 2, 12 and 5 child elements and thus 2, 2, 12 and 5 filters respectively.

Table 3.6 shows the data characteristics of four XML streams conforming to the DTD. "Sel." in the table denotes the abbreviation we use for selectivity.

The query in Figure 3.18 contains 25 patterns whose modes can be changed (i.e., $\$b$, $\$c$, $\$d$, $\$e$ and their 21 filters). The number of alternatives that will be explored by *ExhaustOpt* is so large that *ExhaustOpt* is clearly impractical. Therefore we terminate *ExhaustOpt* after it has explored 1000 plans and return the best plan

for $\$a$ in /*listing*
let $\$b$:=$\$a/seller\_info$[$seller\_rating > 4$][$seller\_name$ contains “SF”];
$\$c$:= $\$a/bid\_history$[...]...[...];
$\$d$:= $\$a/auction\_info$[...]...[...];
$\$e$:= $\$a/item\_info$[...]...[...]
where $\$b$ and $\$c$ and $\$d$ and $\$e$
return $\$a$

Figure 3.18: Wide-and-Complex Query on Ebay Data: requiring to return a *listing* whose $\$a/seller\_info$, $\$a/bid\_history$, $\$a/auction\_info$, and $\$a/item\_info$ satisfy 2, 2, 12,and 5 Filters Respectively

Stream	Sel. of $\$b$	Sel. of $\$c$	Sel. of $\$d$	Sel. of $\$e$
1	10%	50%	70%	90%
2	90%	10%	50%	70%
3	70%	90%	10%	50%
4	50%	70%	90%	10%

Table 3.6: Random Data Sets Conforming to Ebay’s DTD: Each Stream around Size 55M

among these 1000 plans. Table 3.7 compares *ExhaustOpt* and *GreedyOpt* for the query in Figure 3.18 on the streams in Table 3.6.

	ExhaustOpt		GreedyOpt			Initial Plan
	Plan Exec. Time (ms.)	Opt. Time (ms)	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	Exec. Time (ms)
Stream 1	30072	15043	57	852	23088	30072
Stream 2	25087	14893	59	825	22209	38690
Stream 3	24508	16012	76	1118	21924	25828
Stream 4	42301	15567	37	545	18590	42301

Table 3.7: ExhaustOpt and GreedyOpt for Query in Figure 3.18 on XML Streams in Figure 3.6 (ExhaustOpt Limited to Explore 1000 Plans)

In Streams 1 and 4, *ExhaustOpt* fails to find any plan better than the initial plan in the first 1000 plans it has explored. This is because selectivity of $\$b$ is rather low so that the optimal plan retrieves $\$b$ in the automaton. When we call

$ExhaustOpt$, we pass it a parameter $navsToBeTried$ (see Algorithm 7), which is a list of pattern retrieval operators whose modes would be changed. The first operator appearing in the $navsToBeTried$ list happens to be $\$b$. $ExhaustOpt$ thus explores all alternative plans with $\$b$ retrieved out of the automaton first. These plans are all worse than the initial plan so that $EnumSearch$ explores the first 1000 alternative plans to no avail. $GreedyOpt$ instead makes steady progress to finding a better plan during each iteration. On all four streams, *GreedyOpt* explores a limited number of alternative plans yet in all cases it finds a plan that cuts the initial execution time by 15% to 56%.

3.10.5 Comparing ExhaustOpt and GreedyOpt on Deep-and-Simple Pattern Trees

We now compare the *ExhaustOpt* and *GreedyOpt* for deep-and-simple queries conforming to the template in Figure 3.19. According to a DTD survey [16], the depth of an XML document is usually less than 8. Therefore we limit the number n in Figure 3.20 to be less than 8. We generate a XML stream in which all patterns in the queries have the same selectivity of 70%. Table 3.8 compares *ExhaustOpt* and *GreedyOpt* for the queries in Figure 3.19 on this stream.

```
 for $v in p_0, $v_1 in $v/p_11, $v_2 in $v_1/p_21,  …,  $v_n in $v_n-1/p_n1
return
    <result> $v, $v_1, … $v_n </result>
```

Figure 3.19: Queries Conforming to Wide-and-Deep Pattern Tree in Figure 3.16 (c)

We observe two phenomena in Table 3.8 as follow.

n	ExhaustOpt			GreedyOpt			Initial Plan
	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	Exec. Time (ms)
3	3	125	3790	3	125	3790	3790
4	4	123	3892	4	123	3892	3892
5	5	150	4012	5	150	4012	4012
6	6	145	3991	6	145	3991	3991

Table 3.8: ExhaustOpt and GreedyOpt for Deep-and-Simple Pattern Trees on XML Stream with Size of 51M

1). *ExhaustOpt* and *GreedyOpt* explore exactly the same set of alternative plans. This is because every pair of pattern retrieval in the plan has a pattern dependency relationship. As long as one pattern retrieval has been moved out in the initial plan, no other pattern retrieval can be further moved out in the newly derived plan. Therefore after exploring n alternative plans each of which corresponds to moving out one pattern retrieval in the initial plan, both *ExhaustOpt* and *GreedyOpt* terminate.

2). No matter what the value of n is, the best plan is always the one which retrieves all patterns in the automaton. Due to the pattern dependency, p_{31} must be retrieved after p_{21}; p_{41} must be retrieved after p_{31} and so on. Regardless of whether these patterns are retrieved in or out, the execution order is always serialized. Retrieving these patterns out of the automaton does not provide any extra benefit. The plan in which all pattern retrieval is pushed into the automaton ensures that the least amount of data is buffered.

3.10.6 Comparing ExhaustOpt and GreedyOpt on Deep-and-Complex Pattern Trees

It is interesting to observe that for queries conforming to the deep-and-complex pattern tree in Figure 3.16 (d), $GreedyOpt$ terminates very quickly. According to Lemma 2 in Section 3.5, two operators that have a pattern dependency relationship cannot both undergo mode changes. Suppose from a current plan, the mode change on p_{i2} ($1 < i < n$) in Figure 3.16 is chosen, then the mode changes on its ancestor and descendant patterns, including p_{11}, p_{21}, ..., $p_{(i-1)1}$, will no longer be considered. Suppose the mode change on p_{i1} is chosen, then even more mode changes are disqualified for consideration, including mode changes on patterns p_{11}, p_{21}, ..., and p_{n1}.

To illustrate the property of quick termination of $GreedyOpt$, we test the queries conforming to the template in Figure 3.20. We then run these queries on the same XML stream used in Section 3.10.5. Table 3.9 reports the results.

```
for $v in p_0
let $v_11 := $v/p_11,
    $v_12 := $v/p_12,
    $v_21 := $v_1/p_21,
    $v_22 := $v_1/p_22,
    …,
    $v_n1 = $v_{n-1}/p_n1,
    $v_n2 = $v_{n-1}/p_n2
where $v_11 and $v_12 and … $v_n1 and $v_n2
return $v
```

Figure 3.20: Queries Conforming to Wide-and-Complex Pattern Tree in Figure 3.16 (d)

n	ExhaustOpt			GreedyOpt			Initial Plan
	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	Exec. Time (ms)
3	147	2296	7356	10	205	8059	9122
4	595	8674	10086	14	364	11202	13569
5	2387	38500	12176	17	487	12176	17045
6	9555	180078	13408	20	647	14280	20055

Table 3.9: ExhaustOpt and GreedyOpt for Deep-and-Complex Ternary Pattern Trees on XML Stream with Size of 51M

Even for the queries involving a large number of patterns, $GreedyOpt$ terminates rather quickly. Let us use the last row in Table 3.9 as an example. When n = 6, i.e., the depth of the pattern tree in Figure 3.16 (c) is 6. There are 18 patterns in total in the tree. $ExhaustOpt$ explores 9555 alternatives while $GreedyOpt$ only explores 20 alternatives. Even though $GreedyOpt$ fails to find the optimal plan, the plan it finds still cuts down the execution time of the initial plan by 29%.

3.10.7 Study on when GreedyOpt Fails to Find Optimal Plan

We now investigate when GreedyOpt may fail to find the optimal plans. We study extract-same and extract-different queries, shown in Figure 3.21, conforming to the wide-and-complex pattern tree in Figure 3.16 (b) with n = 5. Since each p_i ($1 < i$ $< n$) has two child patterns p_{i1} and p_{i2}, there are 15 patterns in the query in total.

For each query, we perform extensive experiments on different data sets. We also test with different initial plans. Note that in the one-time optimization scenario, we always use an initial plan that retrieves all patterns in the automaton. However, in the continuous optimization scenario, the initial plan of each optimization is the plan found in the last optimization. Therefore the initial plan can be any kind of plans. We find that in two cases GreedyOpt fails to find the optimal

```
for $v in p_0,                      for $v in p_0,
where $v/p_1[p_11] [p_12]           let $v_1 := $v/p_1
    and …                            where $v/p_1[p_11] [p_12]
    and $v/p_n[p_n1][p_n2]               and …
return $v                                and $v/p_n[p_n1][p_n2]
                                     return $v_1
            (1)                                 (2)
```

Figure 3.21: Extract-Same and Extract-Different Queries Conforming to Wide-and-Complex Pattern Tree in Figure 3.16 (b)

plans.

Case 1: Missing Synergy Benefits

In the first case, $GreedyOpt$ fails to find the optimal plan of the extract-different query in Figure 3.21 (b). The characteristics of this case are as below. The initial plan retrieves all the patterns but p_{11} and p_{12} in the automaton. The optimal plan found by *ExhaustOpt* retrieves all patterns in the automaton. $GreedyOpt$ fails to find the optimal plan. In the first iteration, $GreedyOpt$ changes the mode of one pattern retrieval at a time. No single mode change leads to a better plan in this iteration. $GreedyOpt$ then terminates.

However $ExhaustOpt$ finds that if it pushes both p_{11} and p_{12} into the automaton, $\$v$ does not need to be extracted. Instead, only $\$v_1$ needs to be extracted. This way we cut the extraction cost by (cost of extracting $\$v$ - cost of extracting $\$v_1$). If the cost cut is large enough, then pushing in p_{11} and p_{12} can yield a better plan than the initial plan. However this better plan is not considered by GreedyOpt. GreedyOpt only considers a mode change on one single pattern retrieval at each time. When all single mode changes fail, GreedyOpt would not further explore the synergy that may result from the combination of two mode changes.

To experimentally illustrate this case, we design two XML streams as below.

1). In XML stream 1, children of bindings of $\$v_1$ in Figure 3.16 (b) are bound to either $\$v_{11}$ or $\$v_{12}$. Therefore extracting the bindings of $\$v_1$ costs almost the same as extracting the bindings of $\$v_{11}$ and $\$v_{12}$.

2). In XML stream 2, bindings of $\$v_1$ contain many children other than bindings of $\$v_{11}$ and $\$v_{12}$. Therefore extracting bindings of $\$v_1$ costs significantly more than extracting only bindings of $\$v_{11}$ and $\$v_{12}$.

For each stream, we design two queries as below.

1). In query 1, p_{11} and p_{12} have low costs and low selectivity. There is also a costly filter in the format of "$\$v_{n1}$/text() contains ...". Therefore p_{11} and p_{12} are favored to be retrieved in the automaton. Doing so reduces the cost of the costly filter.

2). In query 2, p_{11} and p_{12} have high costs and high selectivity. All the other patterns however have low costs and low selectivity. Therefore p_{11} and p_{12} are favored to be retrieved out of the automaton.

Combining the above two XML streams and two queries, we get the four settings in Figure 3.10. For each setting, we test both the extract-same and extract-different queries in Figure 3.21. Therefore there are eight settings in total. For each setting, we apply $ExhaustOpt$ and $GreedyOpt$ on an initial plan that retrieves all patterns but p_{11} and p_{12} in the automaton. The results are reported in Figure 3.22.

setting	Data Characteristics				Query Characteristics		
	Data Size (M)	size of $\$v_1$/(size of $\$v_{11}$ + size of $\$v_{12}$)	selectivity of p_{11}/p_{12}	selectivity of other patterns	complexity of p_{11} and p_{12}	complexity of other patterns	filters
1	52	100%	90%	10%	complex (involving "//")	simple	a complex filter (i.e., involving a costly "/text() contains ...") is posed on $\$v_{n1}$ and has a selectivity of 90%
2	52	100%	10%	10%	simple (not involving "//", length = 1)		
3	58	300%	90%	10%	complex		
4	58	300%	10%	10%	simple		

Table 3.10: Environment Settings for Testing Case of "Missing Synergy Benefits"

We see that $GreedyOpt$ works well in most of these 8 cases. It only fails to find the optimal plan in the setting 4 (see the highlighted row in Figure 3.22). Note that $GreedyOpt$ for the extract-same query with the same setting (the row in italics font) however yields the optimal result. The "extra synergy benefits" save the extraction cost of $\$v_1$ when all children of $\$v_1$ are retrieved in the automaton. However if $\$v_1$ has to be extracted anyway, then this cost cannot be saved no matter whether the patterns are retrieved in or out of the automaton.

Case 2: Accounting of Cost Cut from Secondary Effect

In the second case, $GreedyOpt$ fails to find the optimal plans for both queries in Figure 3.21. The characteristics of this case are as below. In the first iteration, $GreedyOpt$ finds that pulling out p_{11} alone and pulling out p_{12} alone generates two better plans respectively. However pulling out p_1 also causes p_{11} and p_{12} to be pulled out. This is called *secondary effect* when a pattern with descendant patterns is pulled out (see Section 2.4.3). Pulling out p_1 can yield a plan that is even better than the two previous ones. $GreedyOpt$ then chooses to pull out p_1. This new

	Setting	ExhaustOpt			GreedyOpt		
	Used	# of plans explored	Plan Chosen	Plan Exec. Time (ms.)	# of plans explored	Plan Chosen	Plan Exec. Time (ms.)
Buffer-Same Query	1	3124	no change on initial plan	15502	15	no change on initial plan	15502
	2	3124	p_{11} & p_{12} pushed in	12309	40	p_{11} & p_{12} pushed in	12309
	3	3124	no change on initial plan	18028	15	no change on initial plan	18028
	4	*3124*	p_{11} *&* p_{12} *pushed in*	*15127*	*40*	p_{11} *&* p_{12} *pushed in*	*15127*
Buffer-Different Query	1	3124	no change on initial plan	15578	15	no change on initial plan	15778
	2	3124	p_{11} & p_{12} pushed in	10321	40	p_{11} & p_{12} pushed in	10321
	3	3124	no change on initial plan	19211	15	no change on initial plan	19211
	4	**3124**	p_{11} **&** p_{12} **pushed in**	**12695**	**15**	**no change on initial plan**	**17644**

Figure 3.22: ExhaustOpt and GreedyOpt for Environment Settings in Figure 3.10 Illustrating “Missing Synergy Benefits”. Initial Plan Used: All Patterns but p_{11} and p_{12} Retrieved in Automaton.

.

plan can actually lose to a plan resulted from pulling out both p_{11} and p_{12} but not p_1. The cost cut of pulling out p_1 may come from its secondary effect. In short, $GreedyOpt$ accounts the cut cost to a mode change while the credits should actually be given to the secondary effect.

We design three settings in Figure 3.11. In all settings, p_{11} and p_{12} are inclined to be retrieved out of the automaton because of their high selectivities and high costs.

setting	Data Characteristics				Query Characteristics			
	size of $\$v$/size of $\$v_1$	selectivity of p_1	selectivity of p_{11} and p_{12}	selectivity of other patterns	complexity of p_{11} and p_{12}	complexity of p_1	complexity of other patterns	filters
1	100%	50%	90%	10%	complex	simple	simple	a costly filter on v_{n1}
2	150%	50%						
3	150%	20%						

Table 3.11: Environment Settings for Testing Case of “Wrong Accounting of Cost Cut”: Size of XML Stream 1, 2 and 3 is 42, 62, 59M Respectively

XML streams 1 and 2 differ in the ratio of the size of bindings of $\$v$ to the size of bindings of $\$v_1$. The first two rows in Figure 3.23 show the results of applying ExhaustOpt and GreedyOpt for the extract-different query. The initial plan retrieves all patterns in the automaton. For XML stream 1, the difference between the non-optimal plan chosen by GreedyOpt and the optimal plan chosen by ExhaustOpt is not significant because the cost of buffering the bindings of $\$v$ is close to that of buffering the bindings of $\$v1$. In contrast, for XML stream 2, the cost difference of the two plans is more significant due to the increased difference in their buffering costs.

The XML streams 2 and 3 differ in the selectivity of $\$v$. The last two rows in Figure 3.23 show the results of applying $ExhaustOpt$ and $GreedyOpt$ for the extract-same query. The initial plan retrieves all patterns in the automaton. In XML stream 3, selectivity of p_1 is rather low. Pulling out p_1 is thus not chosen. Therefore $GreedyOpt$ still finds the optimal plan. In XML stream 2, selectivity of p_2 is higher than in XML stream 1. This time pulling out p_1 is chosen while actually pulling out p_{11} and p_{12} only is even better. $GreedyOpt$ fails to find optimal plan. In summary, $GreedyOpt$ on both buffer-different and buffer-same queries can fail to find the optimal plans because of a wrong accounting of the cost cut.

	$Setting$	ExhaustOpt			GreedyOpt			Initial Plan
	$Used$	# of plans explored	Plan Chosen	Plan Exec. Time (ms.)	# of plans explored	Plan Chosen	Plan Exec. Time (ms.)	Exec. Time (ms.)
Buffer-Different Query	1	3124	p_{11} & p_{12} pulled out	15502	27	p_1 pulled out	16013	19036
	2	3124	p_{11} & p_{12} pulled out	18045	27	p_1 pulled out	20549	21202
Buffer-Same Query	2	3124	p_{11} & p_{12} pulled out	18507	40	p_{11} & p_{12} pulled out	18507	21155
	3	3124	p_{11} & p_{12} pulled out	16021	27	p_1 pulled out	16972	18598

Figure 3.23: $ExhaustOpt$ and $GreedyOpt$ Comparison for Settings in Figure 3.10 illustrating “Wrong Accounting of Cost Cut”. Initial Plan Used: All Patterns Retrieved in Automaton.

Conclusion

The case study in Section 3.10.7 sheds some lights on how we can further improve $GreedyOpt$. In case 1, in order not to miss synergy benefits, we can improve the termination criterion in the *GreedyOpt* algorithm. Currently, *GreedyOpt* terminates when no single mode change leads to a better plan than the current plan. Instead, we could further check whether multiple mode changes can lead to a better plan.

In other words, currently $GreedyOpt$ chooses a next plan by 1-lookahead (i.e., one mode change on current plan) while the improved $GreedyOpt$ chooses a next plan by k-lookahead where k ¿ 1 (i.e., multiple mode changes on current plan). In case 2, in order to correctly account the cost cut, we can improve the criterion of which plan to adopt as the current plan in the *GreedyOpt* algorithm. Suppose a best plan, denoted as P, in a search iteration results from a mode change that has secondary effects. In other words, the current plan undergoes one target mode change and several secondary mode changes to become P. Currently, we adopt P as the current plan. Instead, we can also cost a plan P' resulted from only the secondary mode changes. If P' is better than P, we then adopt P' as the current plan.

3.10.8 Comparison of GreedyOpt and GreedyPruneOpt

In continuous optimization scenarios, we now drop the optimization dimension of reordering input subplans to $StructuralJoin$. The greedy algorithm can now be made even more efficient with pruning rules. In the previous sections, the size of XML streams we use vary between 40M - 60M. To study the continuous optimization scenario, we now assume the statistics change for every 20M - 30M of XML stream. We then compare $GreedyOpt$ and $GreedyPruneOpt$ on an XML document about 20M - 30M. We repeat the same queries with the XML streams of the same data characteristics as in Figure 3.17. The only difference is that the size of the XML stream is now 25M instead of 52M used in Figure 3.17.

From Figure 3.24 we can see that Greedy with pruning cuts down the number of plans explored in all six experiments. For wide and simple queries, no patterns have descendant patterns. Therefore the technique of “pruning by bounding cost

Setting	n	Greedy			Greedy with Pruning			Initial Plan
		# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	# of plans explored	Opt. Time (ms.)	Plan Exec. Time (ms.)	Exec. Time (ms.)
Stream 1	5	9	232	782	1	64	782	901
	10	27	475	2865	3	106	2865	3165
	20	144	2271	4821	10	242	4821	6821
Stream 2	5	15	381	2032	10	142	2032	2632
	10	54	823	4742	36	294	4742	7353
	20	204	3126	11405	136	1053	11405	18210

Figure 3.24: GreedyOpt and GreedyPruneOpt for Buffer-Same Queries in Figure 3.17 (1)

cut" described in Section 3.7 takes effect. This technique excludes the pull-out of those $TokenNav$ with selectivity of 10%. Moreover, it also cuts down the unit time spent on processing each alternative plan since we no longer apply the optimization using the input-subplan-reordering rule. Among all six experiments, the pruning technique improves the optimization time most significantly for row 3 in Figure 3.24 since the initial plan has more $TokenNav$ operators that have a selectivity of 10% than any of the other five initial plans.

3.10.9 Overhead of One-time Optimization: From Statistics Collection to Plan Migration

The overhead of run-time optimization is composed of three components, i.e., statistics collection, plan search and plan migration (if any). We study the overhead of each of the three components in the one-time optimization scenario. Since we have already studied the overhead of the plan search time when comparing ExhaustOpt and GreedyOpt in Section 3.10.2, we now focus on the overhead of statistics collection and plan migration.

Query Sets: We design two queries both of which conform to the template in

Figure 3.21 (a) but differ in the number of patterns in the query (n in Figure 3.21 is 5 and 10 respectively; each node has exactly two children). We can compare the overhead of statistics collection in the execution of these two queries, since a query involving more patterns spends more time in statistics collection.

Data Sets: We also design two streams (the design principle is similar to that for XML streams used in Section 3.10.3). In XML stream 1, for either queries mentioned above, 4/5 of p_1, ..., and p_n have a selectivity of 10% while the rest 1/5 have a selectivity of 90%. In XML stream 2, only 1/5 of p_1, ..., and p_n have a selectivity of 10% while the rest 4/5 have a selectivity of 90%. In both streams, all child patterns of p_1, ..., and p_n have the same selectivity as their parent patterns. For a query runs on XML stream 1, the optimal plan is only slightly different from the initial plan which retrieves all patterns in the automaton. In contrast, when the same query is run on XML stream 2, its optimal plan undergoes more dramatic changes from the initial plan. We therefore can compare the overhead of a simple plan migration with a more complicated plan migration process.

Given the above two queries and two queries, we have four experiment settings. Figure 3.25 reports the result in the four experiments. For each query, we illustrate the four cost ingredients of query processing with run-time optimization, i.e., (1) the plan execution time, i.e., the execution time of initial plan + the execution time of the optimized plan, (2) the plan search time by $GreedyOpt$ algorithm, (3) the time for statistics collection and (4) the plan migration time. The costs of the latter three is the overhead of the run-time optimization. We can see that in all four experiments, the plan search time dominates the overhead. The time of statistics collection ranges from 10ms - 20ms while that of plan migration ranges from 0ms - 40ms (the statistics collection time and the plan search time are so small that they are almost unrecognizable in Figure 3.25). Table 3.12 further compares the query processing time without the run-time optimization with that with the run-time optimization. In all four experiments, the query processing with run-time optimization has better performance than that without run-time optimization.

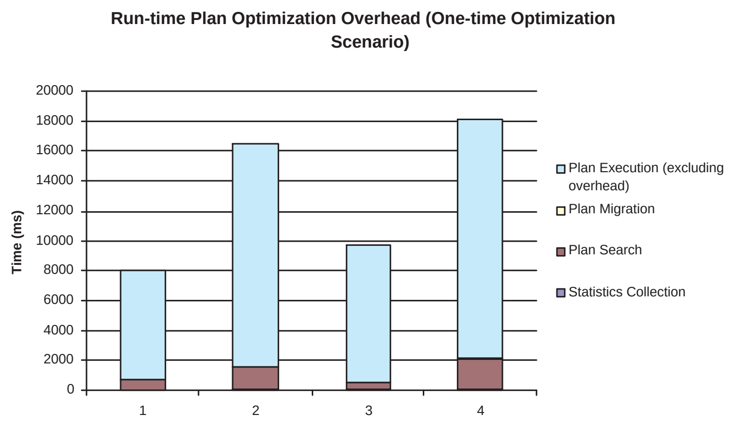


Figure 3.25: Cost Ingredients of Query Processing in One-time Optimization

Setting	Query Processing Time without Run-time Optimization (ms)	Query Processing Time with Run-time Optimization (ms)
1	8690	7982
2	18828	16506
3	17331	9635
4	28415	18135

Table 3.12: Comparison of Query Processing Time with and without Run-time Optimization

3.10.10 Performance of Continuous Optimization

We have studied the plan search performance in the continuous optimization scenario by comparing Greedy and Greedy with pruning in Section 3.10.8. We have also shown in Section 3.10.9 that both statistics collection and plan migration are very cheap. In this section, we focus on the effect of continuous optimization on the query processing rate, i.e., number of bottom input elements processed per second.

We use the buffer-same query that is also used for experiments in Figure 3.22. We generate four XML fragments each of which contains 2500 auctions. The data characteristics of these XML fragments are shown in Figure 3.10. We concatenate these four XML fragments into one stream. If we denote a plan that retrieves all the patterns in the automaton as P_1, and a plan that pulls out p_{11} and p_{12} as P_2. According to Figure 3.22 (first four rows), the run-time optimization will lead to the following plan changes ($P_1 \rightarrow P_2$ denotes P_1 is changed to P_2): $P_1 \rightarrow P_2 \rightarrow P_1 \rightarrow P_2 \rightarrow P_1$. We start optimization every 500 auctions.

We compare the two plan execution processes, one with the run-time optimization and one without run-time optimization. Figure 3.26 shows the processing rate over time. For plan execution without run-optimization, there are four periods in each of which the processing rate is rather consistent. For plan execution with run-time optimization, there are two small time windows (around 18s and 28s) in which the processing rates are significantly lower than those in its neighboring time windows. These two windows indicate the time when optimization for XML stream fragments 3 and 4 happens. Since the query engine spends time (0.2s and 0.6s respectively) on plan search without processing any input, the processing rates decrease. The optimization for XML fragment 1 happens around the 3rd second so that we can see the processing rate starts to increase from this point. The optimization for XML fragment 2 happens around 8s. There is however not an obvious processing rate decrease as that for XML fragments 3 and 4. This is because the plan chosen for XML fragment 2 is faster than any plans chosen for other XML fragments. The plan search for XML fragment 2 takes 0.6s, but in the rest of 1.6s the processing rate is rather high. So on average the processing rate is not significantly lower than before.

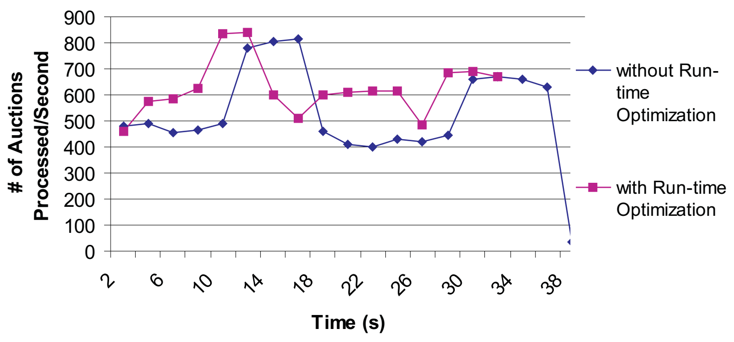


Figure 3.26: Processing Rate of Wide and Complex Query in Continuous Optimization Scenario

Chapter 4

Schema-based Optimization in Automaton Processing Style

4.1 Introduction

Using schema knowledge to optimize queries, known as semantic query optimization (SQO), has generated promising results in deductive [74], relational [66] and object databases [44]. Naturally, it is also expected to be an optimization direction for XML stream query processing. In contrast, pattern retrieval is specific to the XML data model. Therefore, recent work on XML SQO techniques [11, 26, 30, 35, 53] focuses on pattern retrieval optimization. Most of them fall into two categories: techniques applicable to both persistent and streaming XML, or techniques only applicable to persistent XML. We however focus on SQO specific to XML stream processing.

In Section 1.3.3, we have listed a few drawbacks of the related work in XML Stream SQO field. First, most of the work [17, 35] 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. Addressing a more powerful query language such as XQuery will bring more complexities to issues such as how to decide whether a schema constraint is useful and how to execute the optimized query. Second, most of current work overlooks synergy of general and stream specific optimizations. For example, type inference, which infers the types of the nondeterministic navigation steps such as "*" or "//", can be combined with the stream specific XML SQO to enable more optimization opportunities. Third, there lacks strategies for applying possibly overlapping optimization techniques. [17, 35] 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.

To overcome the above drawbacks, 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:

- 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).

- 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.

- 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).

- 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.

- We perform a set of experiments on both real and synthetic data which illustrates that our SQO techniques can significantly improve the performance with little overhead.

4.2 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 [53, 72] on the query trees when an XML Schema is given.

```
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 4.1: Grammar of Supported XQuery Subset

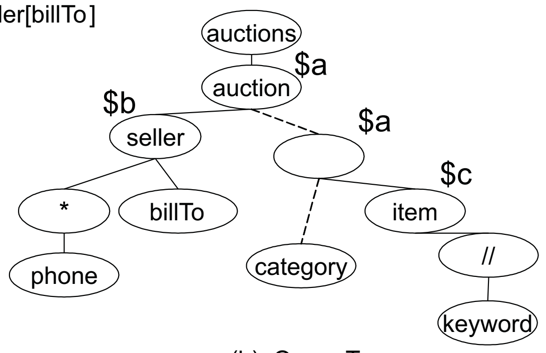
4.2.1 Query Tree

We support a subset of XQuery as shown in Figure 4.1. 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. A large range of common XQueries can be rewritten into this subset [56]. For example, a query with “let” clauses can be rewritten into an XQuery without “let” clauses (by Rule NR1 in [56]). A query with FWR expressions nested within a “for” clause can also be rewritten to our supported subset format (by Rule NR_4 in [56]). The filter expression in an XPath can be moved into the “where” clause (e.g., Figure 4.2 (a) may be rewritten from “for $\$a$ in /*auctions*/*auction*, $\$b$ in $\$a$/*seller*[*sameAddr*] ...”). In short, syntax in Figure 4.1 cover a large portion of commonly used XQuery expressions.

We propose *query trees* to represent the structural patterns in an XQuery. Figure 4.2 (b) shows such a tree for the XQuery in Figure 4.2 (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 4.2 (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*.

```
for $a in /auctions/auction, $b in $a/seller[billTo]
where $b/*/phone=“508-123-4567”
return
<auction>
  for $c in $a/item
    where $c//keyword=“auto”
    return
      <iteminfo>
        $a/category, $c
      </iteminfo>
</auction>
```

(a) Example Query



(b) Query Tree

Figure 4.2: XQuery and Query Tree

There are two kinds of patterns in an XQuery. XPaths in “for” clauses describe required patterns, e.g., in Figure 4.2 (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.

4.2.2 Type Inference

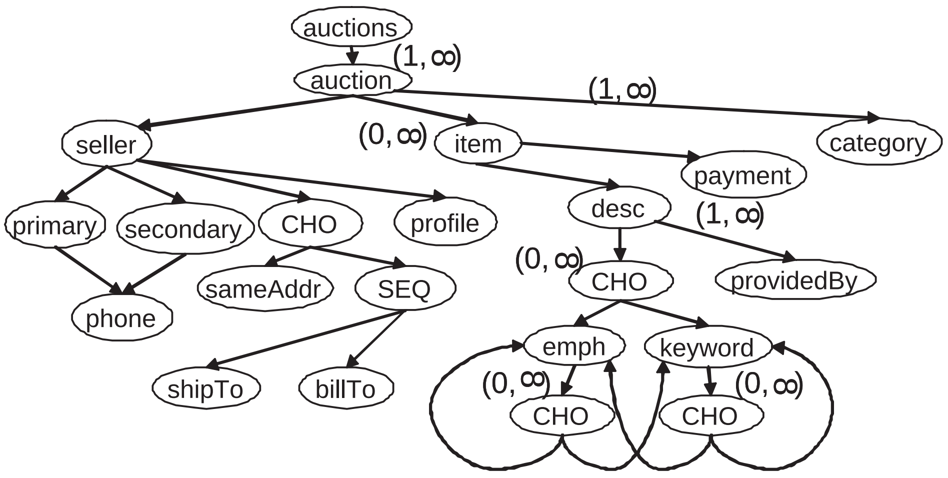
We assume that an XML schema is given for each stream source. An XML schema is modeled as a directed graph with ordered edges. A node in the *schema graph* represents an element type, a sequence group (labeled with "SEQ"), or a choice group (labeled with "CHO"). Each edge from node u to node v is labeled by (*minOccur*, *maxOccur*), indicating the minimal and maximal occurrence of v within u. The default edge label is (1, 1). Figures 4.3 (a) and (b) show the schema (for compactness, we use an equivalent DTD) and its graph representation.

Figure 4.4 shows the query tree from Figure 4.2 (b) after type inference [53, 72]. 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 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 4.4, we refer to them as $keyword_1$ and $keyword_2$ respectively.

A "*" is resolved to a union of types. In Figure 4.4, "*" 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 *key*-

```
<!ELEMENT auctions (auction+)>
<!ELEMENT auction (seller, item*, category+)>
<!ELEMENT seller (primary, secondary,   sameAddr|(shipTo , billTo), profile)>
<!ELEMENT primary (phone)>
<!ELEMENT secondary (phone)>
<!ELEMENT item ( desc, payment)>
<!ELEMENT  desc((emph|keyword )*, providedBy +)>
<!ELEMENT  emph (#PCDATA|emph|keyword )*>
<!ELEMENT keyword (# PCDATA|emph|keyword )*>
…
```

(a) Schema



(b) Schema Graph

Figure 4.3: XML Schema and Schema Graph

word or *emph* elements (refer to Figure 4.3). The "//" node in Figure 4.2 (b) is now expanded to a *desc* node and a "//" node in Figure 4.4.

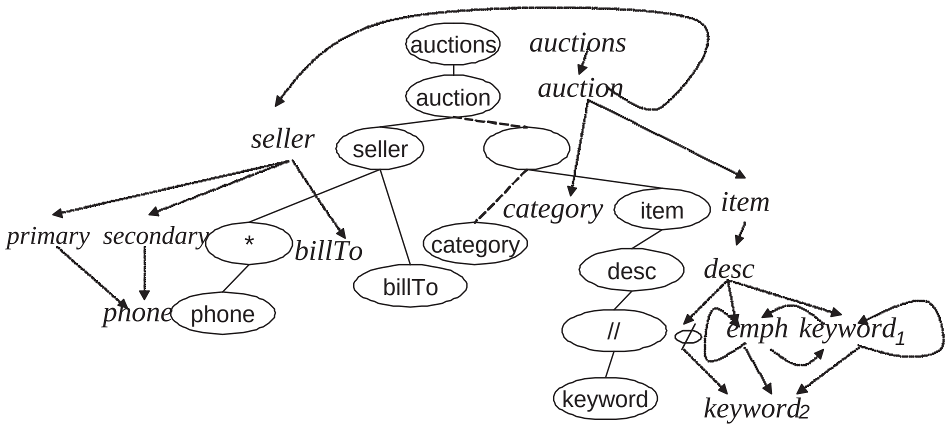


Figure 4.4: Query Tree after Type Inference

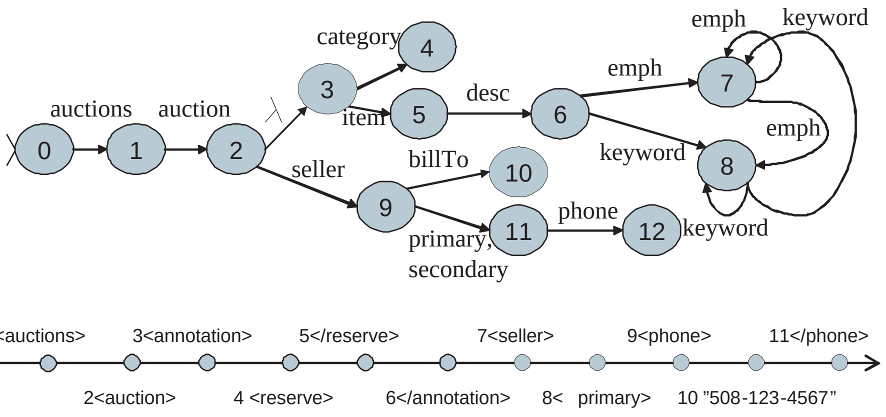
4.3 Guidelines for Stream XML SQO

SQO in essence is a heuristics-based optimization. It however still has to be based on some common beliefs in the characteristics of the physical implementations. For example, the classical "selection pushdown under join" heuristics-based rewriting rule is built on the assumption that a selection is usually cheaper than a join. We therefore 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.

4.3.1 Automata-based Implementation

Automata are widely used [30, 34, 35, 42, 52, 65] for pattern retrieval over XML token streams. We describe one basic automata model [30, 42] that is general and serves as the core of most other automata [34, 35, 65]. The pattern retrieval in the automaton consists of three tasks as below.

Locating Tokens. Figure 4.5 shows the automaton for retrieving the patterns in Figure 4.4. 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 4.5.



[0]	[1] [0]	[2,3] [1] [0]	[ ] [2,3] [1] [0]	[ ] [ ] [2,3] [1] [0]	[ ] [2,3] [1] [0]	[2,3] [1] [0]	[9] [2,3] [1] [0]	[11] [9] [2,3] [1] [0]	[12] [11] [9] [2,3] [1] [0]	[12] [11] [9] [2,3] [1] [0]	[11] [9] [2,3] [1] [0]

Figure 4.5: Automaton Implementation

A stack is used to store the history of state transitions. Figure 4.5 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 4.5, 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 2.5.

4.3.2 Necessity of Physical Implementation Analysis

Without close analysis of the physical implementation, an apparently useful SQO technique can be actually useless. In Figure 4.5, both $\$a$/*item* and $\$a$/*category* need to be located. We know from the schema in Figure 4.3 that *category* only occurs after *item* in an *auction*. We might expect to save some time by postponing the locating of *category* till *item* has been located, i.e., removing the transition from state 3 to state 4 and only recovering it when state 5 is activated. This is similar to the SQO technique in XSM [52] which removes transitions that will not happen.

Transition entries are usually implemented as a hash table [81, 42, 34] for performance reasons. A transition lookup, i.e., a hash table lookup, costs constant time [34]. Therefore cutting down the number of transitions in the entries of state 3 does not affect the lookup cost. Therefore in the above example, the new automaton does not save any cost. In XSM [52] however, the transition lookup at state s is implemented as a linear search on all possible transitions of s. It is worthwhile to cut down the number of transitions in XSM.

4.3.3 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.

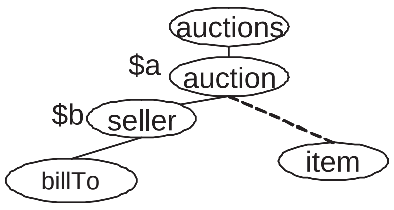
The second opportunity is that an extraction operator should be prompted to revoke the buffering flag once the data it is extracting is known to be irrelevant to the final results. This saves buffering cost.

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 4.6, 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 4.3 indicates *item* occurs after *seller* within an *auction*. The locating and buffering $\$a$/*item* is saved. Figure 4.7 summarizes the guidelines of designing XML SQO.

```
for $a in /auctions/auction,
    $b in $a/seller[billTo]
return
<auction>
  $b/@id, $a/item
</auction>
```

(a) Example Query



(b) Query Tree

Figure 4.6: Filtering Propagation

4.4 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.

A 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 4.7: SQO Design Guidelines

4.4.1 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 4.7. 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 17 *In Figure 4.4, $maxOccur(phone, seller)$ = 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 18 *From Figure 4.3 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 19 *In Figure 4.4, $keyword$ 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 $//keyword$.*

In the rest of the paper, instead of saying the start tag of the first result (Exclusive and Order Rules) or the end tag of the i^{th} result (Occurrence Rule) of a path is an ending mark, we simply say the path or the i^{th} occurrence of a path is the ending mark.

4.4.2 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.

Completeness

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

Definition 7 *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:*

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

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

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

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 4 *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 5 *All possible pairs of rules r_1-r_2 in our current rule set are independent of each other.*

We briefly explain Lemma 5. First, when a rule in Section 5.3 is applied on a node, it is not affected by the ending marks previously found. Equation (1) in Definition 7 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 7 also hold. Lemmas 4 and 5 will be combined later to show our rule application algorithm achieves completeness.

Minimality

A plain node-by-node rule-by-rule application, though ensuring completeness (Lemma 4), may not ensure *minimality*. It may introduce redundant ending marks.

Example 20 ***(Rules Applied on Same Node)*** *Exclusive and Order Rules, if applied on node $billTo$ in Figure 4.4, 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 21 ***(Rules Applied on Ancestor and Descendant Nodes)*** *Suppose the schema for $auction$ in Figure 4.3 is changed to <!$ELEMENT$ $auction$ (..., $item$, ...)>. The Order Rule on node $keyword$ finds an ending mark: $/desc/providedBy$ (see Example 19) 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 4.3 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. There are two kinds of surely-working ending marks, as illustrated below.

- If an ending mark that is found by Occurrence or Order Rule, it is surely-working if it is guaranteed to appear in the stream. For example, the ending mark $/payment$ for $\$c//keyword$ (see Example 19) is not surely-working since even though $minOccur$($payment$,$item$)>0, $minOccur$($item$, $auction$)> 0. In contrast, the ending mark $/category$ for $\$a/item$ is surely-working since $minOccur$($category$,$auction$) $>$ 0 and $minOccur$($auction$,$auctions$)>0.
- If an ending mark is found by Exclusive Rule, it is surely working if (1) it is "alternative" to p, i.e., either B or p must appear (this is stronger than

"exclusive" which only requires the ending mark and p do not coexist); and (2)p cannot involve in a selection predicate because the absence of B only ensures p appears but cannot ensure p satisfies the predicate.

Based on the *surely-working* concept, we have Observations 1 and 2 which generalize the cases illustrated in Examples 20 and 21 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 4.7 has a surely-working ending mark, and (2) $\$v'$ occurs within $\$v$ exactly once.*

4.4.3 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 4 and 5, 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*.

The traverser algorithm (Algorithm 4.4.3) takes two inputs. The first input is a type node of a context node $\$v$. The traverser picks qualifying destination nodes

Algorithm 10 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

```
1: Set $ecas$;
2: for each destination node $\$v'$ of $\$v$ do
3:   $ecas$ = $ecas \cup$ applyRule($\$v'$, $tn$, $atn$);
4:   for each type node $tn'$ of $\$v'$ do
5:     if $maxOccur$($tn'$, $tn$)=1 and $\$v'$ has only one type node that is a descen-
       dant of $tn$ then
6:       $ecas$ = $ecas \cup$ traverser($tn'$, $atn$);
7:     else
8:       $ecas$ = $ecas \cup$ traverser($tn'$, $tn$).
9:     end if
10:  end for
11: end for
12: return $ecas$.
```

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 Algorithm 4.6), 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.

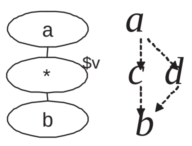
Example 22 *Figures 4.8 (a) and (b) show a query and a schema. The traverser starts from the root node in Figure 4.8 (c) and finds its destination node $\$v$. The rule applier operates on $/a/*$, namely, $/a/(c|d)$ according to the type inference. An ending mark $/a/e$ is found. Next, the traverser navigates 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.*

```
for $v in /a/*,
Where $v/b/text() = "001"
return $v
```

(a) Original query

```
<!ELEMENT a (c?, d?, e)>
<!ELEMENT c (b, b, …)>
<!ELEMENT d (b, …)>
```

(b) Schema



(c) Query Tree

Figure 4.8: Traverser on Context Node with Multiple Types

In Figure 4.4.3, *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.

In *localApplyRule* in Figure 4.4.3, the Occurrence, Exclusive and Order Rules are applied in turn on $dest$. The ending marks will then be found in the order they

Algorithm 11 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

```
1: Set $ecas$;
2: $T$ = type nodes of $dest$ that are descendants of $tn$
3: find $t'$ where
   (i) $t' \in T$ and $t'$ occurs after all other types in $T$
   (ii) $maxOccur(t', tn)$ = 1
4: **if** $t'$ exists **then**
5:    **for** each destination node $dest'$ of $dest$ **do**
6:       applyRule($dest'$, $t'$, $atn$);
7:    **end for**
8:    **if** every $dest'$ has a surely-working ending mark **then**
9:       return an empty set;
10:   **end if**
11: **end if**
12: $ecas$ = $ecas$ ∪ localApplyRule($dest$, $tn$, $atn$).
13: return $ecas$.
```

Algorithm 12 localApplyRule($dest$, tn, atn)

Input and Output: same as applyRule algorithm

```
1: Set $ecas$;
2: $ecas$ = $ecas$ ∪ localApplyOneRule($OccurrenceRule$, $dest$, $tn$, $atn$);
3: **if** there is no surely-working ending mark **then**
4:    $ecas$ = $ecas$ ∪ localApplyOneRule($ExclusiveRule$, $dest$, $tn$, $atn$);
5: **end if**
6: **if** there is no surely-working ending mark **then**
7:    $ecas$ = $ecas$ ∪ localApplyOneRule($OrderRule$, $dest$, $tn$, $atn$).
      07 return $ecas$.
8: **end if**
```

Algorithm 13 localApplyOneRule(r, $dest$, tn, atn)

Input: r - an SQO Rule; $dest$, tn, atn - same as those in applyRule
Output: a set of event-condition-actions

```
1:  Set $ecas$;
2:  if precondition check on $dest$ passes then
3:    while more ending mark is found do
4:      find the next earliest ending mark $A$ for $dest$ within $dest$'s context node
        of type $tn$;
5:      if postcondition check on $dest$ passes then
6:        $ecas$ = $ecas \cup (A, n, atn)$.
7:      end if
8:      if $A$ is surely-working then
9:        break;
10:     end if
11:   end while
12: end if
13: return ecas.
```

appear in the stream. Following Observation 1, if a surely-working ending mark is found, we terminate the rule application.

4.5 Execution of Optimized Queries

We have incorporated the proposed SQO techniques into *Raindrop*. We describe (1) how to encode the event-condition-actions derived in Section 4.4 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 4.4.

4.5.1 Encoding Event-Condition-Actions

We use our running example to illustrate how the event-condition-actions derived by the rule application algorithm are encoded in an Raindrop plan. The top part in Figure 4.9 shows the plan for the XQuery in Figure 4.2 (a). For ease of illustration, each operator is annotated with an identifier. For example, the inner FWR expression in Figure 4.2 (a) is modeled as the subplan within the box in Figure 4.9. 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 $StructuralJoin_{\$c}$.

The bottom of Figure 4.9 also depicts the automaton for locating the patterns. The automaton has encoded three event-condition-actions derived in Section 5.3. Compared to the original automaton in Figure 4.5, new states have been added for the newly introduced patterns, e.g., state 13 for $\$b/sameAddr$ (see Example 18). The property below must hold in the automata in order for the event-condition-actions to work correctly.

Property 3 *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 4.10 shows two alternative automata constructed for the query tree in Figure 4.8. Both the type node pairs (c, b) and (d, b) in Figure 4.8 are mapped to state 4 in Figure 4.10 (a). The automaton in Figure 4.10 (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 4.8 (b) shows a correct automaton where the above type node pairs are mapped to states 4 and 5 respectively.

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 4.9, 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 4.2. A binding of $\$a$ may still appear in the final results even if it does not contain any qualifying bindings of $\$c$.

4.5.2 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 4.9, 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 4.3).

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.

4.6 Experimentation

We implemented the SQO techniques in *Raindrop* [39, 38] using Java 1.4. Experiments are run on two 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. Also, we do not include the time spent on reading the stream from the sockets in the query evaluation time so as to isolate the network cost.

4.6.1 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 4.11. The notations p_{11}, ..., p_{21}, ..., p_{31}, ... stand for XPath expressions and val_{21}, val_{22}, ... stand for constant strings. Table 4.1 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 4.1: Query Characteristics

Figure 4.12 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 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.

4.6.2 Synergic Effect of Combining Type Inference and Stream SQO

Figure 4.13 shows the synergic effect of combining type inference and stream SQO. We use Q_2, Q_5 and Q_6, each benefiting most from Order, Exclusive and Occurrence Rule respectively, for the study. For each query Q, we pick a path expression $\$v$/$n_1$/$n_2$/.../$n_i$ that has the lowest selectivity. In the first testing, we modify the expression to $\$v$/*/*/.../$n_i$ and get a query Q'. In the second testing, we modify the expression to $\$v$//$n_i$ and get a query Q''. We then run Q' and Q'' without SQO, Q' and Q'' with SQO, Q with type inference and Q with both type inference and

SQO.

In Q_2, the path before modification has a length of 2. The plan with “//” takes 22% more time to finish than the plan with “*” because the previous plan has to perform automata transitions for every element at any depth. The plans after SQO is applied on all the path expressions except the modified one (i.e., Q' and Q'' with SQO) show obvious improvement while the plan combining both type inference and SQO boosts even more performance gain.

In Q_5, the path modified has a length of 5. Since the average depth of PSD is 5.1, almost every element leads to automata transitions. Therefore the plan with “*” costs almost the same as the plan with “//”. Only one path expression in Q_5 offers SQO opportunity. Therefore after we modify the expression, no SQO optimization is possible (Q' and Q'' are the same as Q). The plan with both type inference and SQO cuts down 50% of the execution time of the plan with type inference.

Finally, in Q_6, the expression that has the lowest selectivity occurs rather later in its context node. Early filtering leads to minor improvement. The plan combining both type inference and SQO has performance similar to those with SQO applied on the other path expressions.

4.6.3 Necessity of “Usefulness” Criteria

The data sets used in the rest of the paper are generated by an XML generator ToXGene [24]. They conform to the schema used in XMark [7]. We now illustrate the necessity of introducing only ending marks that satisfy the criteria in Figure 4.7.

For the query in Figure 4.2 (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 4.14. 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.

4.6.4 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. The optimization percentage may be even larger when SQO is applied on a query with larger unit gain, say, SQO helps avoiding the computation of buffering, path recognition and selection evaluation all together.

4.6.5 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.

For testing the overhead of the SQO technique using Occurrence rule, we run the query on three data sets. Each data set is composed of a sequence of *open_auction* element. The ending mark will occur in each *bidder* subelement of an *open_auction* element. The three datasets differ in the average number of *bidder* subelements, i.e., 1, 10 and 20, in an *open_auction*. Therefore, ending marks occur least frequently in data set 1 and most frequently in data set 3. Figure 4.18 reports the results on these data sets. The ratio of the optimized plan with the original plan is 114%, 103% and 112% respectively.

The overhead of SQO technique using Exclusive Rule is reported in Figure 4.19. Note when the Occurrence Rule is applied on a pattern, at most derive one ending mark since the maximal occurrence of a pattern is unique. Unlike the Occurrence Rule, the Exclusive Rule, when applied a pattern, may lead to multiple ending marks (refer to Example 18). We design three schema. When used to optimize a pattern p in the query, they lead to 1, 10, 20 ending marks for p respectively. As explained in Example 18, different number of ending marks for the same pattern should not make much difference in the performance, assuming a hash table lookup time is assumed not to be affected by the number of entries in the hash table. Figure 18 confirms that plan with 1, 10, or 20 ending marks perform very similarly on all three data sets. Overall, the ratio of the plan optimized with SQO with the original plan is around 113%, 101% and 105% on three data sets.

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 4.20. 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.

4.6.6 Summary of Experiments

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.

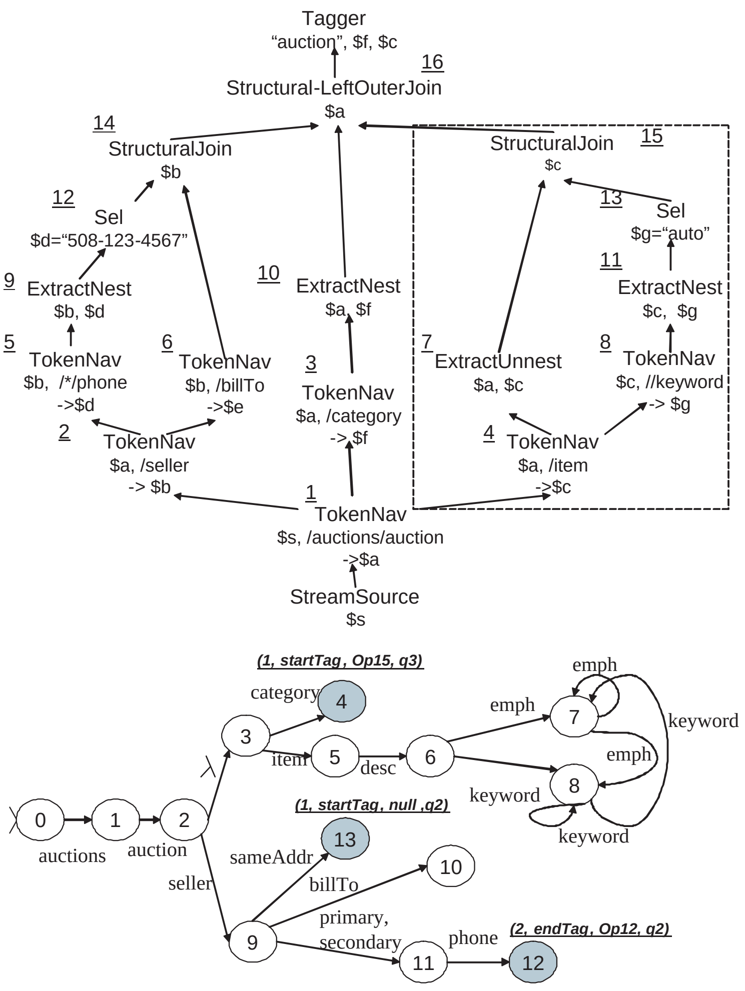


Figure 4.9: Encoding SQO into Algebraic Plan

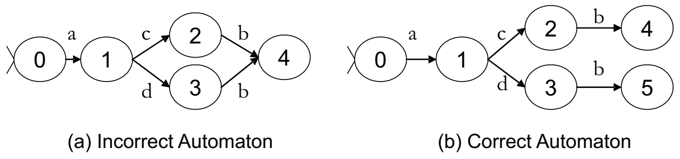


Figure 4.10: "Conflict-free" Property of Automata

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 4.11: Query Template

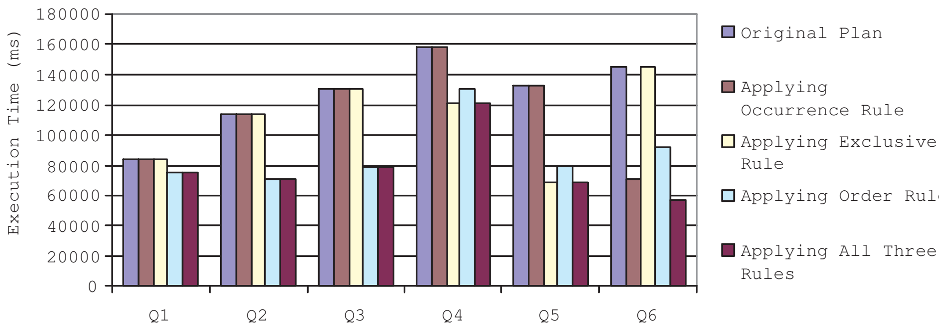


Figure 4.12: Effect of SQO on Queries Using a 800M PSD Dataset

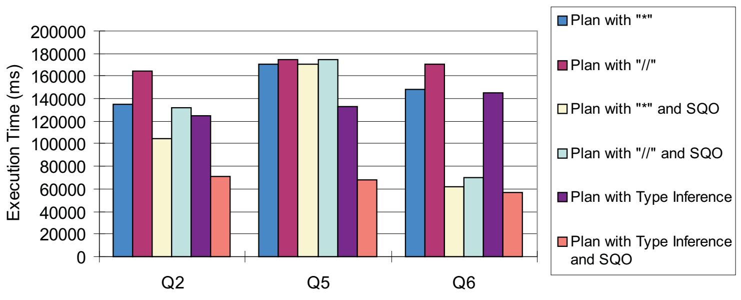


Figure 4.13: Effect of Combining Type Inference and SQO on a 800M PSD Dataset

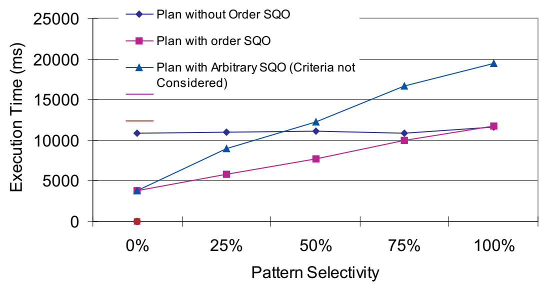


Figure 4.14: Comparing Plans Only Adopting Necessary Ending Marks Satisfying with Plans Adopting All Ending Marks

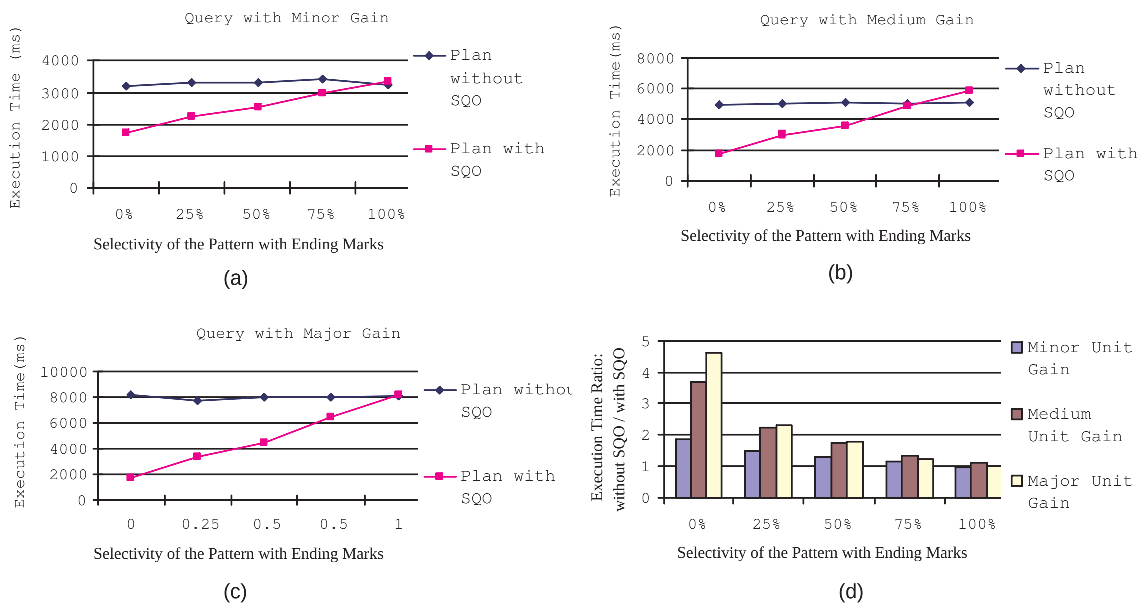


Figure 4.15: Effect of Pattern Selectivity/Unit Gain on Saving Path Location Cost

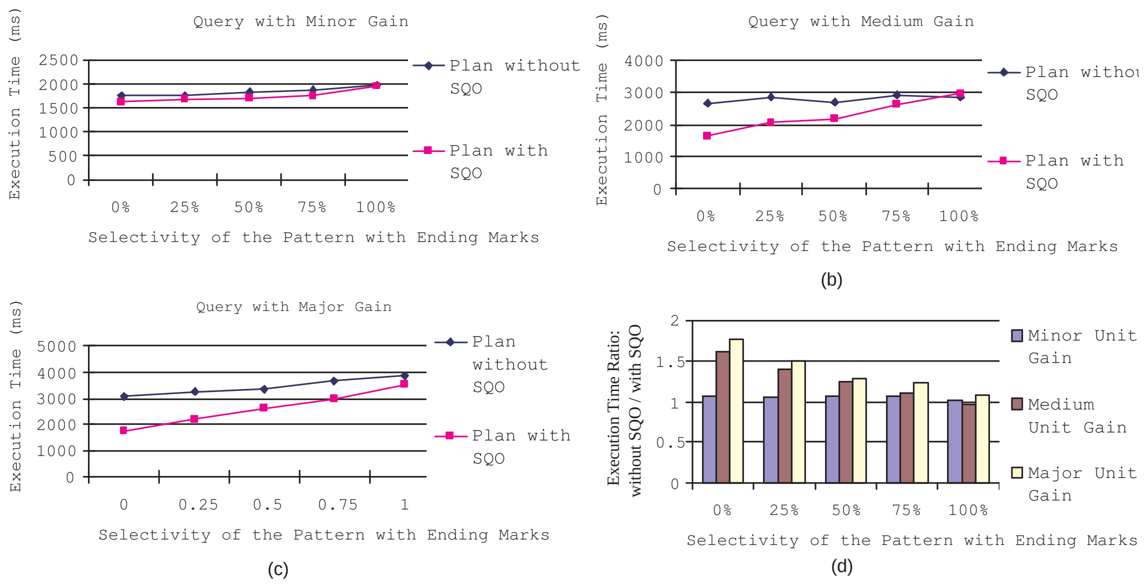


Figure 4.16: Effect of Pattern Selectivity/Unit Gain on Saving Buffering Cost

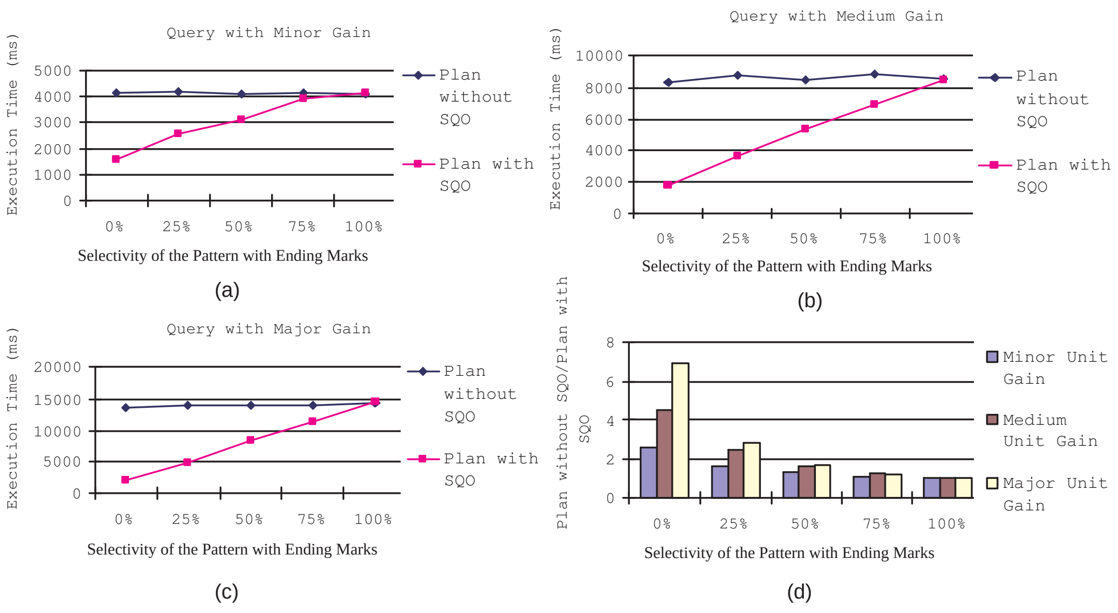


Figure 4.17: Effect of Pattern Selectivity/Unit Gain on Saving Selection Evaluation Cost

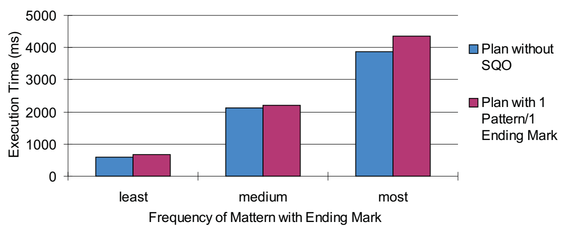


Figure 4.18: Overhead of Applying Occurrence Rule

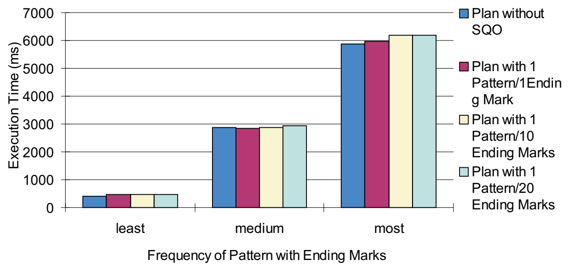


Figure 4.19: Overhead of Applying Exclusive Rule

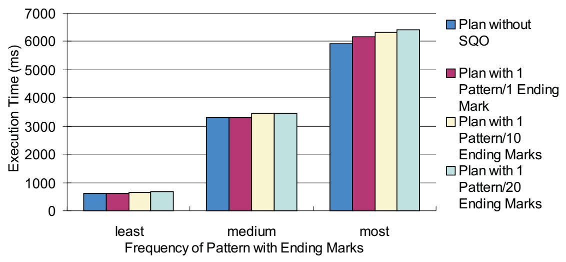


Figure 4.20: Overhead of Applying Order Rule in Worst Case

Chapter 5

Related Work

5.1 Related Work on XML Query Processing Paradigms

Stream processing has attracted a great deal of attention in the networking and mobile-computing communities. Typical stream applications include networking traffic monitoring, sensor network management and web tracking and personalization. Most projects like *Fjord* [55], *Aurora* [20], *Cougar* [28], *CAPE* [67] and *STREAM* [14] address general issues of querying data streams, assuming a tuple-like data model.

Research is also active in the field of querying XML streams. NiagaraCQ [43], while using XML query syntax, mainly addresses on SQL-like filtering on tuple-based inputs. It does not address pattern retrieval related issues. Moreover, NiagaraCQ [43] focuses on the optimization of multiple XML queries by sharing their common expressions, rather than the optimization of one single query.

Several XML query engines [18, 25, 30, 42, 52, 65] focus on optimizing the pattern retrieval in XML queries. XSM [52] and XSQ [65] use the transducer models for pattern retrieval. XSM and XSQ support XQuery and XPath respectively. Basically, they define a template for each component in XQuery or XPath, and then compile the query into a network of such instantiated templates. Though XSM supports queries with more expressive power than XSQ does, XSQ provides more efficient memory management than XSM by promptly cleaning up intermediate buffers when they are no longer needed.

Lazy PDA [34] and XPush [35] are based on deterministic automata. They handle a limited subset of XML query language features. Both of them only return a boolean value indicating whether an XPath expression evaluates to non-empty results. Lazy PDA stands for *lazy deterministic pushdown automata*. It is called “lazy” because it computes the automata states at run-time so that only the states that would actually be transitioned to are computed. This could effectively reduce the exponential blow-up of the number of states compared to when the “eager” PDA would be computed at compile time. Lazy PDA supports only XPath expressions without filters (i.e., linear patterns) while XPush allows XPath expressions to have filters (i.e., tree patterns). XPush extends Lazy PDA by having additional constructs for supporting tree patterns and predicate evaluation.

The above pure automaton approaches [52, 65, 34, 35] use tokens throughout the query processing. They do not support converting tokens into XML element nodes. Therefore they are only able to express a Raindrop query plan that retrieves all patterns on tokens in their constructs, but unable to express a Raindrop query plan that retrieves some patterns on the XML element nodes.

YFilter [80, 30] and Tukwila [42] are closest to our work. They model the whole automaton processing as one operator with fixed interface and coarse granularity. As mentioned in Section 1.2.2, we call their approaches a *loosely-coupled* *automaton and algebra paradigm*. Our work instead uniformly integrates the token-based and tuple-based computations and thus naturally offers query rewrite optimization opportunities. Meanwhile, our physical operators are efficiently implemented by taking advantage of the automata properties.

Another camp of research [32, 33] builds systems using SAX handlers. They define a set of handlers, each for handling certain computations such as evaluating a navigation step, performing a selection and constructing an element. These handlers are nested so that one handler can pass an event it receives to another handler. Again, this is a new methodology not in synch with well-known algebraic optimization techniques. Existing algebra optimization techniques cannot be directly adopted.

The loosely-coupled automaton and algebra paradigm and the SAX handler based paradigm support both tokens and XML element nodes in their query processing. Therefore they are able to express a Raindrop plan that retrieves some patterns on the XML element nodes in their constructs. However, the way they model a query plan is not suitable for exploring the automaton-in-or-out optimization opportunities. The loosely-coupled paradigm does not provide rewrite rules to pull out pattern retrieval from the operator that models the automaton processing. As for the SAX handler based paradigm, it is not clear how to apply cost estimate and search algorithm for optimization.

BEA/XQRL [25] bears some resemblance to an XQuery stream processing system. However, BEA/XQRL actually processes stored XML data. The data are stored as a sequence of tokens. XQuery is compiled into a network of expressions. An expression is equivalent in functionality with an algebraic operator. There are two major differences between BEA/XQRL and Raindrop. First, in BEA/XQRL, all the internal data passed among expressions are always token streams, in contrast to both tokens and tuples in *Raindrop*. Second, the tokens in BEA/XQRL and the tokens in Raindrop are not equivalent concepts in terms of their accessibility. In BEA/XQRL, the token stream is stored (either on disk or in memory) so that the same data can be accessed by expressions multiple times. In Raindrop, tokens arrive on-the-fly. They cannot be accessed more than once unless they are buffered, as explicitly specified by the *Extract* operators. The pull-based model in BEA/XQRL, which assumes a look back on previous tokens is possible, does not work here. It has to work with other execution models in a stream context. As illustrated in Section 2.6, a data driven model (i.e., the push-based model) is a must for buffering some data before a pull-based model can operate on buffered data.

5.2 Related Work on Run-time Plan Optimization

5.2.1 Cost-based Optimization

System R [63] first introduced cost-based optimization for relational databases. Choosing a good join order [70] is the major focus of early cost-based optimization. Later, cost-based optimization is extended to cover all aspects of a query plan, including ordering expensive selection predicates [45, 75], placement of group by [70] etc.

Cost-based optimization has also been actively studied for static XML processing. Lorel [57], a static XML database engine, adopts cost-based optimization techniques. Lorel proposes a set of indexes on XML. For example, a *label index* supports finding all element nodes with a certain name, e.g., finding all *seller* elements. Lorel physical operators provide different ways for finding a path in a bottom-up, top-down or hybrid manner in the XML tree. For example, given a path $seller/phone/primary$, either we find all $seller$ elements first using the label indexes (top-down), or all $primary$ element first (bottom-up) or all $phone$ element first (hybrid). Lorel provides a cost model and a plan enumeration algorithm to choose among different path navigation alternatives. The major search space pruning techniques Lorel uses are heuristics. For example, suppose there are two path expressions starting from the same context variable, e.g., $\$u/p1$ and $\$u/p2$, Lorel does not attempt to reorder them.

Cost-based optimization in Timber [82], another static XML database engine, focuses on choosing an optimal order for structural joins. Timber's search algorithm is based on the traditional dynamic programming algorithm for join ordering [63, 27]. The basic idea of dynamic programming for join ordering is as follows. First, all access paths to every table involved in the join are generated. Second, all partial plans with two-way joins are generated. Partial plans with three-way joins are next generated from the two-way joins and so on. Suppose in the two-way join generating phase, we have found out that join order of (table A, table B) (i.e., A is at the left of the join while B is at the right of the join) is better than the join order of (table B, table A), a three-way join in the order of (table B, table A, any other table) will not be generated since it must be worse than the join in the order of (table A, table B, any other table).

The contribution of Timber's search algorithm is that it tries to eliminate those partial query plans that are guaranteed to lead to suboptimal solutions. Timber calls it *dynamic programming with partial plan pruning*. Timber can start constructing $n+1$-way structural joins before it finishes constructing all n-way structural joins. It ranks all partial plans. It then constructs the next plans from the partial plans in the order of their ranks. The purpose of ranking is to create complete plans that are possibly optimal as early as possible. Therefore, any partial plan that has already costed more than a best complete plan found so far can be excluded. This is essentially a classical A* search strategy [61]. An important property that has to hold for this pruning work is that the cost of a partial plan is independent of how it is joined with the rest of the relations (we say a partial plan has independent cost). In other words, when a partial plan is expanded to a new partial plan, i.e, (i-way structural joins expanded to $i+1$-way structural joins), the new partial plan must cost more than the old partial plan. This is called *autonomously increasing cost* property in A* search. If this property does not hold, the pruning can exclude partial plans that will lead to optimum.

The above idea may seem to bear some resemblance to our problem. A partial plan in our problem can be one in which the token-or-node retrieval modes of a subset of pattern retrieval have been determined. Can we also exclude certain partial plans if they cost worse than a known complete plan? The answer is no. This is because the pattern retrieval in the partial plan may not be independent from the pattern retrieval whose modes have not been determined yet. That is to say, the cost of the partial plan may still decrease. Therefore, a partial plan that is worse than a currently best complete plan is still be possible to be expanded to a new better complete plan.

In summary, dynamic programming with partial plan pruning is not very suitable in our scenario for two reasons. First, it is enumerative which takes too long for run-time optimization. Second, the property of autonomously increasing cost does not necessarily hold here so that partial plan pruning may not always be applicable.

5.2.2 XML Statistics Collection

Statistics are indispensable information for cost-based optimization. Many studies for XML statistics focus on XML's nested structures. For example, Lorel [57] maintains statistics of all paths of length up to m where m is a tunable parameter. They use these statistics to infer selectivity of longer paths. Aboulnaga [2] proposes techniques that can more aggressively summarize the paths by pruning and aggregation to reduce the size of statistics. Their techniques do not maintain correlations between paths. Such limitations are addressed in [84] which maintains statistics for tree pattern query. These techniques all require scanning the whole data.

Another kind of solution for XML statistics collection is to use query feedback [51, 50]. The idea is to issue a query workload on the XML data and learn information about the XML structure and PCDATA values from the query feedback (i.e., query results). Such solution is especially suited for the scenario where XML data is either inaccessible or too large to be completely scanned.

As we have mentioned in Section 3.2.5, these techniques are best suitable in two scenarios. The first scenario is that the stream query engine has to process a large number of queries so that it cannot afford to collect specific statistics for each query. Summary techniques are needed for the requirement of scalability. The second scenario is that user queries can be added after the stream starts to arrive. We should be able to summarize the statistics as the stream runs so that once a new query is added, we can immediately estimate its plan cost. In this way we can better achieve quick response time for the newly added query.

5.2.3 Run-time Re-optimization

Due to the hardware and workload complexity, data complexity and user interface complexity [13], a new query paradigm, *adaptive query processing*, emerges to tackle these problems. Most of the work is in the relational context. *Eddy* [13] is a representative work of this query paradigm. In this paradigm, the query plan is no longer fixed. Instead, each tuple, driven by the processing cost/selectivity of the operators and tuple arrival rate, can go through operators in a flexible order, controlled by a special scheduling operator called *eddy*. In other words, the query plan is reformulated on a tuple-by-tuple basis. The reformulation is based on *lottery scheduling*. Each time an operator is given an input tuple, it is credited one "ticket". The *eddy* operator holds a lottery for each tuple. An operator's chance of winning the lottery (i.e., being assigned the tuple to) corresponds to the count of ticket the operator holds. This lottery scheduling scheme enables a lightweight plan formulation compared to other work on runtime plan reformulation [48].

Eddy's plan reformulation is limited to changing the order of operators, such as changing to execute a join operator A before another join operator B if A turns out to have less selectivity than B. It is not clear how it handles the other aspects of plan re-optimization. For example, suppose there are two alternative plans 1 and 2, composed of operator set $\{op1, op2, op5\}$ and $\{op3, op4, op5\}$ respectively. The corresponding Eddy module is shown in Figure 5.1. The two groups of operators within the dashed lines are exclusive, that is, if a tuple is routed to one group, it cannot be routed to the other group later on. Now none of Eddy's routing schemes is applicable in this situation. When the plan re-optimization is limited to plan reordering, a simple greedy algorithm can be used, i.e., finding the "best-behaving"

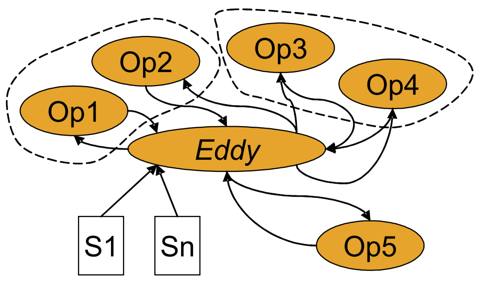


Figure 5.1: Operator Re-ordering in *Eddy*

operator and routing tuples to it as many as possible. However, when two plans share different set of operators, an overall plan performance measurement must be adopted rather than a local operator performance comparison. For example, even if $op1$ is the most efficient among the operators, the overall plan 2 may still perform better than plan 1.

5.3 Related Work on Schema-based Optimization

Semantic query optimization has been long studied in deductive [74], relational [66] and object databases [68, 44]. Due to the flat data models in deductive and relational databases, their SQO techniques are usually for optimizing the filtering on flat values. The major techniques include *join elimination*, *join introduction*, *predicate elimination*, *predicate introduction* and *detection of the empty answer set*. These techniques can be similarly applied to the XML domain as long as the XML counterpart of such schema knowledge is offered. For example, if a key and foreign key constraint between two tables is provided in the XML schema, a semi-join on the two tables (projecting on the table with the foreign key constraint) can be eliminated using the *join elimination* technique.

Object data model, though nested, has a rigid structure in contrast to the irregular structures in XML data model. Therefore SQO research in object databases has not been motivated to optimize the detection of missing patterns. The SQO on nested structure navigation in object databases, such as *access scope reduction* [44], is oriented mainly to the OO-specific class/subclass constraints.

SQO for persistent XML may have some resemblance to the stream-specific SQO. XQRL [26] 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 1.3.3, 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 [30] and XSM [52] 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 respectively. These in essence type inference techniques belong to general XML SQO.

In the automata model XSM adopts, the transition lookup at state s is not implemented as a hash table lookup but as a linear search on all possible transitions of s.

XSM uses schema knowledge to reduce the possible transitions in order to reduce the transition lookup time.For example, to find a path $\$v/a$, an XSM transducer state corresponding to $\$v$ will have two transitions with conditions “next token = <a>” and “next token $\neq$ <a>” respectively. If it is known from the schema that a binding of $\$v$ can have only a subelements, then the second transition can be eliminated. Such an optimization is not applicable to our automaton, since a hash table lookup cost is not related to the number of entries (i.e., possible transitions) in the table.

Both AT&T’s XML stream engine [17] and XPush [35] support boolean XPath matching. Their SQO techniques are less complicated that those for supporting XQueries. The reason has been illustrated in Section 1.3. Both of them consider one case of using the order constraints, which is a subset of our SQO techniques, i.e., the order rule in Section .

The goal of FluXQuery [18] 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 [34, 5]. The stream index SIX [34] 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. XHints [5] extends SIX by supporting more metadata information. 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.

Chapter 6

Conclusions and Future Directions

6.1 Conclusion

Architecture: *Raindrop* accommodates a token-based automaton paradigm and a tuple-based algebraic paradigm within one framework. This is a novel approach compared to the other approaches in the literature [42, 30] which typically model the two processing paradigms separately and thus optimize them separately as well. Our approach instead allows the query optimization to be performed in a uniform manner over all computations. With all the computations under the same umbrella of an algebraic framework, we can apply existing algebraic optimization techniques, such as separation of logical and physical plans, query rewriting and costing etc.

Our algebraic framework consists of three abstraction levels. The highest level is semantics-focused plan level. General optimization techniques that are neither specific to stream nor specific to stored data can be applied. The next level is the stream logical plan level. On this level, a set of rewriting rules are developed to switch pattern retrieval into or out of the automaton. The lowest level is the stream physical plan level. We offer efficient implementations of operators that take full advantage of automata properties at the stream physical level. We also provide multiple models to synchronize the execution of the operators.

Run-time Automaton-in-out Plan Optimization: We provide a unique optimization opportunity that is not explored before. Previous literature considers only the plans in which all pattern retrieval is pushed down into the automaton. Our experimentations in Section 2.7 however demonstrate that such plans do not ensure the optimality. With 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. First, we define a search space. Whether a pattern should be retrieved in or out of the automaton is the core issue in the search space. The side issue that comes with the core issue, namely, where to place the patterns that are pulled out, is also considered by our techniques. Second, we develop a cost model for comparing the alternative plans in the search space. Third, we propose three algorithms for searching for a good plan in the search space. These three algorithms include exhaustive search, greedy search and greedy search with pruning rules.

Moreover, we assume the whole process takes place at run-time. Therefore we tackle two additional problems. First, we embed the statistics collection into the operators so that we can collect statistics at the time of plan execution. Second, we study how to correctly and efficiently migrate a currently running plan to a new plan found by the plan search algorithms.

Schema-based Optimization for Pattern Retrieval in Automaton: For the pattern retrieval performed in the automaton, we provide schema-based optimization. Limited work has been done in SQO techniques on structural pattern retrievals on XML streams. Moreover, these limited work [17, 35] only supports XPath or XPath boolean matching, which is a less powerful query language than XQuery. Our work instead supports SQO on XQuery. We handle the complexities brought by this more powerful query language in the below three aspects.

First, we derive a set of 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. Second, we propose a rule application order to guarantee the quality of the optimized queries. Third, we present how to incorporate these techniques into Raindrop query plans and how to efficiently evaluate such plans enhanced with SQO. Our experiments show that these SQOs can improve the performance significantly while at the same time introducing negligible overhead in most cases.

6.2 Future Work

Current Raindrop system targets baseline scenarios for XML stream processing. We process XML stream in plain text, which is the physical format of most existing XML data sources. We assume that system resources are enough for handling the query processing. There are many interesting future directions to look at if we extend these baseline scenarios or break some assumptions. We list here a few of them which all remain as open problems in the literature.

6.2.1 Supporting XQueries with Window Joins/Aggregations

The XQueries we support in Raindrop system cover two commonly used functionalities, pattern retrieval and simple predicates in the format of $variable\ op\ constant$. We can enhance the Raindrop system to support other commonly used functionalities such as joins and aggregations. In particular, since an stream can be infinite, we need to support joins and aggregations with window semantics. That is, only data that arrive in a certain window are joined [49] or aggregated [12].

There are two major challenges. First, we need to precisely define the semantics of window joins and window aggregations in XQueries. Window joins and window aggregations are proposed for relational streams [49, 12] but not for XML streams yet to the best of our knowledge. Second, with well defined query semantics of window joins and window aggregations, we then must develop new techniques to efficiently evaluate them.

6.2.2 Query over Indexed XML Streams

The continuous nature of XML streams forces query engines to access every token in sequence. This is very different from the situation of static data sets. Static data sets are usually equipped with index structures that allow for the direct access of the desired subset. As an initial step towards this direction, [34, 5] has proposed the notion of indexed XML streams. For example, an index is coupled with a start element tag, indicating the offset of the corresponding end tag from the start tag.

In our schema-based optimization work, we use schema knowledge to skip the processing of certain chunks of XML streams that are irrelevant to the query result, namely, we simply scan the chunks without performing any automaton transitions.

We actually can do better if the indices proposed by [34, 5] are available. If we can derive from the indices where is the next relevant chunk, we can directly access this chunk without having to scan the irrelevant chunk before it.

There are plenty of research issues in this field. First, what kind of stream indices can we provide? Second, how do we represent such indices? Third, how do we send out these indices? Do we interleave it with the stream or do we send out an index-only stream along with the data stream? Fourth, how do we combine existing schema knowledge, such as DTD or XML schema, with these new indices? All these are interesting problems to explore.

6.2.3 XML Load Shedding

We can relax the assumption that the available system resources are sufficient to cope with the volume of the incoming data streams and the query workload, namely, we can consider scenarios when the incoming data overwhelms the available computing resources, such as CPU processing speed and memory [20]. When the addition of new resources is not practical, an alternate solution to this problem is dropping input tuples to reduce the system load, called *load shedding*. Two dropping strategies have been proposed so far: *random drop* [73], where tuples are dropped based on system performance, and *semantics drop* [4], where tuples are dropped to minimize the impact on application semantics. System resource limitations is a practical consideration in XML stream processing context as well. However, the main challenge with respect to XML stream processing is that a token, unlike a self-contained tuple, is not meaningful by itself, and arbitrarily dropping tokens might result in not well-formed XML streams. Therefore to extend load shedding techniques to XML stream processing, special dropping strategies must be designed. This is an important open research problem that needs to be investigated.

6.2.4 Adaptive Query Approximation

An alternative approach to handle system resources under strain is to rewrite the query itself (instead of affecting the actual input tuples). In other words, the original query can be rewritten into an approximate one with a decreased accuracy, but requiring less system resources. For example, an XPath expression involving a descendant relationship (such as, //*c*) can be rewritten into one involving a child relationship (e.g., from //*c* to /*c*). The automata implementation for recognizing deterministic child navigation steps is cheaper than that for recognizing nondeterministic descendant navigation steps. Moreover the latter will most likely return less results than the former, thus putting even a smaller burden on system resources.

6.2.5 Query over Compressed XML Streams

XML data can be compressed for the purpose of exchange and archiving. The XMill project [37] compresses the structure (XML tags) separately from the PCDATA. The content is distributed to a set of semantically uniform "containers". For example, one container stores all text values of $seller$ elements while another container stores all text values of $bidder$ elements. Another camp of compressing techniques are more "query-friendly". XGrind [64] does not separate PCDATA from structure. An XGrind-compressed document is still an XML document: tags are dictionary-encoded; PCDATA data are compressed but still stay at their original place in the document. Query processing on compressed XML data in both types of compressed structures is an interesting direction for XML stream processing.

Appendix A

Proof of Final State Duplicate Free Property

Theorem 4 *If the "exclusive-reach" property holds, a final state can have at most one instance in the stack (we say the automaton is "final state duplicate free") except in two circumstances: (1) if there is a* $TokenNav_{\$col1,path}\$col2$ *where* $path$ *contains a "//" and the data is recursive; and (2) if there is a* $TokenNav_{\$col1,path}\$col2$ *where a postfix of* $path$ *is a "//" followed by zero or more "*".*

Proof 4 *We prove the theorem by induction.*

Step 1. Given a $TokenNav_{\$s,p}\$d$ *where* $\$s$ *represents the root element, we encode* p *in an automaton where the start state is* q_0 *and the final state is* q_n*. Figure A.1 shows the contents of a stack where* q_n *appears twice. When* q_n *is pushed onto the stack the first time, there must be another* q' *that is pushed onto the stack at the same time. This* q' *can finally transit to* q_n *so that* q_n *is pushed onto the stack the second time. Since* $p \neq$ *(*n^**)?//(*n^**)?/(*$*$*)?, i.e.,* p *cannot both have a "//" and a last* *navigation step of "*". there can be only two below cases for p. We will illustrate that for neither case, the stack contents in Figure A.1 could possibly occur.*

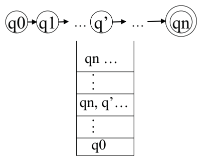


Figure A.1: Stack Containing Duplicate Final States

1). *In the first case, there is no navigation step "//" in p. Correspondingly, there is no self-transition in the automaton. Therefore from any stack top state q_α, it cannot transit to both q_α and q_β. In other words, it cannot transit to two states where one state is "closer" to the final state q_n (if two states are connected by λ transition, they are the same close to q_n). It is thus impossible for a token to enable both q_n and q' to be pushed onto the stack. Therefore q_n cannot appear twice in the stack.*

2). *In the second case, the navigation step "//" appears in p but the last navigation step in p is a deterministic element type, say e, instead of a wildcard navigation step "$*$". Only a token with tag name e can enable q_n to be pushed onto the stack. If the XML input does not have recursion, for an element node of type e in the input, none of its descendant element nodes has type e. Therefore if one token t_1 has enabled q_n to be pushed onto the stack, there cannot be a component token of the element associated with t_1 that also enables q_n to be pushed onto the stack.*

Step 2. Given a $TokenNav_{\$s',p'}\$d'$ where $\$s'$ represents a non-root element. Suppose q'_0 and q'_n are the start and final states of p' in the automaton respectively.

From a state q_0' in the stack, similar to Step 1, as long as $p' \neq$ (n^*)?//(n^*)?/($*$)?, q_0' will not transit to multiple q_n'. Since there will not be another q_0' in the stack, no other q_n' can be transit to. Therefore at any time there will not be multiple q_n' in the stack. $\square$

Appendix B

Computing $P_{\not\Rightarrow\emptyset}(plan)$ for Cost Model

$P_{\not\Rightarrow\emptyset}(plan)$ is used in cost model in Section 3.2.1.

1). If op is $TokenNav_{\$u,p}\$v$, $P_{\not\Rightarrow\emptyset}(op)$ is the probability of a binding of $\$u$ containing at least one bindings of $\$w$.

2). If op is $Select$ or $NodeNav$, $P_{\not\Rightarrow\emptyset}(op)$ is the probability of an operator op generating some output during the processing of one input tuple.

With knowing $P_{\not\Rightarrow\emptyset}(op)$ and $\sigma(op)$ for each opeator in a plan, we can compute $P_{\not\Rightarrow\emptyset}(plan)$ using probability computation technique. We use an example to illustrate this:

Example 23 *In Figure 3.2, we compute $P_{\not\Rightarrow\emptyset}(plan)$ where $plan$ is the one rooted at the entry operator $StructuralJoin_{\$b}$ of $StructuralJoin_{\$a}$. $P_{\not\Rightarrow\emptyset}(plan)$ =*

probability of a binding of $\$a$ containing at least one binding of $\$b$ that passes all operators between $StructuralJoin_{\$b}$ and $TokenNav_{\$a,/seller}\$b$

= 1 $-$ probability of every bindings of $\$b$ failing to pass all operators

= 1 $-$ [1 - probability of a binding of $\$b$ passing all operators$]^{number\ of\ bindings\ of \$b\ in\ a\ binding\ of\ \$a}$

= 1 $-$ [1 - probability of a binding of $\$b$ passing all operators$]^{\sigma(TokenNav_{\$a}\$b)}$

= 1 $-$ [1 - $P_{\neq\emptyset}$($TokenNav_{\$b,/profile}\$e$)$]^{\sigma(TokenNav_{\$a}\$b)}$.

Appendix C

Proof of Optimality of Subplan Evaluation Order

Proof 5 *We assume the contrary of the theorem. That is, given a $StructuralJoin$ that has n input subplans, in an optimal ordering of input subplans, there are two subplans $subplan_i$ and $subplan_{i+1}$ (1 $\leq i < k$) that have $rank(subplan_i) > rank(subplan_{i+1})$. We now compute the costs of $subplan_i$ and $subplan_{i+1}$ using Equation 3 in Section 3.2.2. For simplicity, we use K to denote Expression (7) $\times$ Expression (8.a) in Equation 3. That is, K =*

$\prod_{op\ \in\ operator\ set\ between\ bottommost\ TokenNav\ and\ TokenNav\ that\ retrieves\ \$v} \sigma(op) \times P_{\not\Rightarrow\emptyset}(entry_1) P_{\not\Rightarrow\emptyset}(entry_2) ... P_{\not\Rightarrow\emptyset}(entry_n)$.

Therefore, we have the below equations.

Equation 11 *$Cost$(subplan$_i$) = $K \times P_{\not\Rightarrow\emptyset}$(subplan$_1$) $\times$ ... $\times$ $P_{\not\Rightarrow\emptyset}(subplan_{i-1})$ $\sigma(entryPlan(entry_i))$ $UnitCost(subplan_i)$*

Equation 12 $Cost(subplan_{i+1})$ = $K \times P_{\neq\emptyset}(subplan_1) \times ... \times P_{\neq\emptyset}(subplan_i)$ $\sigma(entryPlan(entry_{i+1}))\ UnitCost(subplan_{i+1})$

Suppose we switch the order of $subplan_i$ and $subplan_{i+1}$. We denote the costs of evaluating a $subplan$ in the new plan as $Cost'(subplan)$. We then have the below equations.

Equation 13 $Cost'(subplan_i)$ = $K \times P_{\neq\emptyset}(subplan_1) \times ... \times P_{\neq\emptyset}(subplan_{i-1})$ $\sigma(entryPlan(entry_{i+1}))\ UnitCost(subplan_{i+1})$

Equation 14 $Cost'(subplan_{i+1})$ = $K \times P_{\neq\emptyset}(subplan_1) \times ... \times P_{\neq\emptyset}(subplan_i)$ $\sigma(entryPlan(entry_i))\ UnitCost(subplan_i)$

We can then derive the below equation.

Equation 15 $Cost(subplan_i) + Cost(subplan_{i+1}) - Cost'(subplan_i) - Cost'(subplan_{i+1})$ = $K \times (P_{\neq\emptyset}(subplan_1) ... P_{\neq\emptyset}(subplan_{i-1})[(1 - P_{\neq\emptyset}(subplan_{i+1})\ \sigma(entryPlan(entry_i))$ $UnitCost(subplan_i) - [1 - P_{\neq\emptyset}(subplan_i)]\ \sigma(entryPlan(entry_{i+1}))\ UnitCost(subplan_{i+1})])$

Because $rank(subplan_i) > rank(subplan_{i+1})$, namely,

$$\frac{\sigma(entryPlan(entry_i))UnitCost(subplan_i)}{1-P_{\neq\emptyset}(subplan_i)} > \frac{\sigma(entryPlan(entry_{i+1}))UnitCost(subplan_{i+1})}{1-P_{\neq\emptyset}(subplan_{i+1})},$$

we have $Cost(subplan_i) + Cost(subplan_{i+1}) - Cost'(subplan_i) - Cost'(subplan_{i+1})$ $>$0. Correspondingly, $\Sigma_{k=1}^{n} Cost(subplan_k) - \Sigma_{k=1}^{n} Cost'(subplan_k) > 0$. This is contrary to the assumption that $\Sigma_{k=1}^{n} Cost(subplan_k)$ is the cost of subplans in the optimal order.

Appendix D

Combination Containing Operators with Pattern Dependency Relationship being Invalid

Suppose $navOp_1$ and $navOp_2$ retrieve two patterns $\$u/p1$ and $\$x/p2$ that have ancestor-descendant relationship (we say the two operators have *pattern dependency* relationship). We want to prove that a combination containing both $navOp_1$ and $navOp_2$ is either redundant, i.e., it produces a same alternative plan as another combination that contains no operators with pattern dependency relationship, or semantics-disallowed, i.e., it produces an alternative plan that is not supported in Raindrop.

We distinguish between three cases: first, $\$u/p1$ and $\$x/p2$ are both retrieved in the automaton; second, $\$u/p1$ and $\$x/p2$ are both retrieved out of the automaton; third, $\$u/p1$ is retrieved in the automaton while $\$x/p2$ is retrieved out of the automaton. The fourth case, i.e., $\$u/p1$ is retrieved out of the automaton while $\$x/p2$ is retrieved in the automaton, is not supported by Raindrop algebra because of the reasons presented in Section 2.4.3. The combination in the first case has been proven to be redundant in Section 3.5. We now prove that the combinations in the second and third cases are either redundant or unsupported in Raindrop.

Second Case: Suppose the combination contains a $NodeNav_{\$u,p1}\$v$ and a $NodeNav_{\$x,p2}\$y$. Changing the modes of both means we push in both $\$u/p1$ and $\$x/p2$. However, pushing in $\$x/p2$ has implied that $\$u/p1$ has to be pushed in as well. For example, in Figure 3.2, pulling out $\$a/seller$ requires $\$b//profile$ to be also pulled out. Therefore, this combination generates the same alternative plan as the combination that contains $NodeNav_{\$x,p2}\$y$ but not $NodeNav_{\$u,p1}\$v$ does.

Third Case: Suppose the combination contains a $TokenNav_{\$u,p1}\$v$ and a $NodeNav_{\$x,p2}\$y$. Changing the modes of both means we end up with a plan which contains a $NodeNav_{\$u,p1}\$v$ and a $TokenNav_{\$x,p2}\$v$. Raindrop does not support such a plan.

Appendix E

Order Insensitive

Suppose $navOp_1$ and $navOp_2$ retrieve two patterns that have no ancestor-descendant relationship. We want to prove that regardless of the order in which we change the modes of $navOp_1$ and $navOp_2$, the two plans derived contain the same operators.

We distinguish between three cases: first, $navOp_1$ and $navOp_2$ are both $TokenNav$ operators; second, $navOp_1$ and $navOp_2$ are both $NodeNav$ operators; third, $navOp_1$ is a $TokenNav$ while $navOp_2$ is a $NodeNav$. We have proven that the order in which we change the modes does not matter in the first case in Proof 3 in Section 3.5. We now prove that order does matter in the second and third cases.

Proof 6 ***Second Case****: Suppose we have two $NodeNav$ operators, $NodeNav_{\$u,p1}\$v$ and $NodeNav_{\$x,p2}\$y$. Pushing in $\$u/p1$ can eliminate the operators or introduce new operators into the plan in four ways. First, $NodeNav_{\$u,p1}\$v$ is rewritten into $TokenNav_{\$u,p1}\$v$ and $Extract_{\$u}\$v$. Second, if before the rewriting $NodeNav_{\$u,p1}\$v$ is the only operator that consumes $\$u$, then the $Extract$ operator that extracts $\$u$ will be eliminated from the plan after the rewriting. Third,* *the ancestor patters of $\$u/p$ that are retrieved out of the automaton will be pushed in. Fourth, if there exists another operator in the format of $TokenNav_{\$u,p'}\$v'$ but there does not exist a $StructuralJoin_{\$u}$ before the rewriting, a $StructuralJoin_{\$u}$ is introduced after the rewriting.*

Later, if we change the mode of $NodeNav_2$, we have the below observations:

1). Mode change of $NodeNav_{\$x,p2}\$y$ can only eliminate the $Extract$ operator that extracts $\$x$. It is impossible that $\$x = \$v$ because $NodeNav_{\$u,p1}\$v$ and $NodeNav_{\$x,p2}\$y$ have no pattern dependency relationship. The mode change of $NodeNav_{\$x,p2}\$y$ will not eliminate the $Extract$ operator that mode change of $NodeNav_{\$u,p1}\$v$ has introduced, i.e., the $Extract$ operator that extracts $\$v$. Hence it will not cancel out the first change resulted from the mode change of $NodeNav_{\$u,p1}\$v$.

2). Mode change of $NodeNav_{\$x,p2}\$y$ can only introduce an $Extract$ operator that extracts $\$y$. It is impossible that $\$y = \$u$ because $NodeNav_{\$u,p1}\$v$ and $NodeNav_{\$x,p2}\$y$ have no pattern dependency relationship. The mode change of $NodeNav_{\$x,p2}\$y$ will not introduce the $Extract$ operator that mode change of $NodeNav_{\$u,p1}\$v$ has eliminated, i.e., the $Extract$ operator that extracts $\$u$. Hence it will not cancel out the second change resulted from the mode change of $NodeNav_{\$u,p1}\$v$.

3). Since $NodeNav_{\$u,p1}\$v$ and $NodeNav_{\$x,p2}\$y$ have no pattern dependency relationship, mode change of $NodeNav_{\$x,p2}\$y$ will not affect those operators whose modes have been changed as a secondary effect of the push-in of $\$u/p$. That is to say, it will not cancel out the third change resulted from the mode change of $NodeNav_{\$u,p1}\$v$.

4). *Mode change of $NodeNav_{\$x,p2}\$y$ cannot eliminate a $StructuralJoin$ operator so that it will not cancel out the fourth change resulted from the mode change of $TokenNav_{\$u,p1}\$v$.*

In summary, a mode change of $TokenNav_2$ that occurs after the mode change of $TokenNav_1$ does not cancel any change that has been made. Therefore the order in which we change the modes of $TokenNav_1$ and $TokenNav_2$ does not matter.

Proof 7 ***Third Case****: Suppose we have a $TokenNav_{\$u,p1}\$v$ and a $NodeNav_{\$x,p2}\$y$. We first prove that a mode change of $NodeNav_{\$x,p2}\$y$ that occurs after the mode change of $TokenNav_{\$u,p1}\$v$ does not cancel any change that has been made. Pulling out $\$u/p$ can eliminate the operators or introduce new operators into the plan in four ways. First, $TokenNav_{\$u,p1}\$v$ and $Extract_{\$u}\$v$ are rewritten into $NodeNav_{\$u,p1}\$v$. Second, if before the rewriting there exists no $Extract$ operator that extracts $\$u$, then an $Extract$ operator that extracts $\$u$ will be introduced to the plan after the rewriting. Third, the descendant patters of $\$u/p$ that are retrieved in the automaton will be pulled out. Fourth, if there exists no other operator in the format of $TokenNav_{\$u,p1'}\$v'$ but there exists a $StructuralJoin_{\$u}$ before the rewriting, this $StructuralJoin_{\$u}$ is eliminated after the rewriting.*

Later, if we change the mode of $NodeNav_{\$x,p2}\$y$, we have the below observations:

1). *Mode change of $NodeNav_{\$x,p2}\$y$ will not eliminate the $NodeNav_{\$u,p1}\$v$ operator. Hence it will not cancel out the first change resulted from the mode change of $TokenNav_{\$u,p1}\$v$.*

2). Mode change of $NodeNav_{\$x,p2}\$y$ can only eliminate the $Extract$ operator that extracts $\$x$ when there is no other operator that consumes $\$x$. The mode change of $TokenNav_{\$u,p1}\$v$ may introduce an operator that extracts $\$u$. Even though it is possible that $\$x$ = $\$u$, the $Extract$ operator that extracts $\$u$ cannot be eliminated since there exists a $NodeNav_{\$u,p1}\$v$ operator that needs to consume $\$u$. Hence the mode change of $NodeNav_{\$x,p2}\$y$ will not cancel out the second change resulted from the mode change of $TokenNav_{\$u,p1}\$v$.

3). Since $TokenNav_{\$u,p1}\$v$ and $NodeNav_{\$x,p2}\$y$ does not have a pattern dependency relationship, mode change of $NodeNav_{\$x,p2}\$y$ will not affect those operators whose modes have been changed as a secondary effect of the pull-out of $\$u/p$. That is to say, it will not cancel out the third change resulted from the mode change of $TokenNav_1$.

4). If mode change of $TokenNav_{\$u,p1}\$v$ eliminates a $StructuralJoin$, that means there exists no other operator in the format of $TokenNav_{\$u,p1'}\$v'$. If $\$x$ = $\$u$, mode change of $NodeNav_{\$x,p2}\$y$ will not introduce a $StructuralJoin_{\$x}$ operator since there exists no operator in the format of $TokenNav_{\$u,p1'}\$v'$. Therefore, the mode change of $NodeNav_{\$x,p2}\$y$ will not cancel out the fourth change resulted from the mode change of $TokenNav_1$.

Next, we prove that a mode change of $TokenNav_{\$x,p2}\$y$ that occurs after the mode change of $NodeNav_{\$u,p1}\$v$ does not cancel any change that has been made. Pushing in $\$u/p1$ can eliminate the operators or introduce new operators into the plan in four ways. First, $NodeNav_{\$u,p1}\$v$ is rewritten into $TokenNav_{\$u,p1}\$v$ and $Extract_{\$u}\$v$. Second, if before the rewriting $NodeNav_{\$u,p1}\$v$ is the only *operator that consumes $\$u$, then the $Extract$ operator that extracts $\$u$ will be eliminated from the plan after the rewriting. Third, the ancestor patters of $\$u/p$ that are retrieved out of the automaton will be pushed in. Fourth, if there exists another operator in the format of $TokenNav_{\$u,p'}\$v'$ but there does not exist a $StructuralJoin_{\$u}$ before the rewriting, a $StructuralJoin_{\$u}$ is introduced after the rewriting.*

Later, if we change the mode of $TokenNav_{\$x,p2}\$y$, we have the below observations:

1). *Mode change of $TokenNav_{\$x,p2}\$y$ can eliminate neither $TokenNav_{\$u,p1}\$v$ nor $Extract_{\$u}\$v$. Hence it will not cancel out the first change resulted from the mode change of $NodeNav_{\$u,p1}\$v$.*

2). *Mode change of $TokenNav_{\$x,p2}\$y$ can introduce an operator that extracts $\$x$. If $\$x \neq \$u$, then the mode change of $TokenNav_{\$x,p2}\$y$ does not cancel out the second change resulted from the mode change of $NodeNav_{\$u,p1}\$v$. If $\$x = \$u$, then the mode change of $TokenNav_{\$x,p2}\$y$ cancels out the second change resulted from the mode change of $NodeNav_{\$u,p1}\$v$. That is, there is an $Extract$ operator that extracts $\$x$ in the final plan. However, suppose we switch the order of mode change, namely, we change the mode of $TokenNav_{\$x,p2}\$y$ first and that of $NodeNav_{\$u,p1}\$v$ next. The mode change of $TokenNav_{\$x,p2}\$y$ introduces an $Extract$ operator that extracts $\$x$. This $Extract$ operator will not be eliminated by the mode change of $NodeNav_{\$u,p1}\$v$ since $NodeNav_{\$x,p2}\$y$ in the plan needs to consume $\$x$. Therefore the $Extract$ operator that extracts $\$x$ appears in both plans regardless of the order in which we change the modes.*

3). Since $NodeNav_{\$u,p1}\$v$ and $TokenNav_{\$x,p2}\$y$ have no pattern dependency relationship, mode change of $TokenNav_{\$x,p2}\$y$ will not affect those operators whose modes have been changed as a secondary effect of the push-in of $\$u/p$. That is to say, it will not cancel out the third change resulted from the mode change of $NodeNav_{\$u,p1}\$v$.

4). Mode change of $TokenNav_{\$x,p2}\$y$ can eliminate $StructuralJoin_{\$x}$. If $\$x \neq \$u$, then the mode change of $TokenNav_{\$x,p2}\$y$ does not cancel out the fourth change resulted from the mode change of $NodeNav_{\$u,p1}\$v$. If $\$x = \$u$, then the mode change of $TokenNav_{\$x,p2}\$y$ cancels out the fourth change resulted from the mode change of $NodeNav_{\$u,p1}\$v$. That is, $StructuralJoin_{\$x}$ operator does not appear in the final plan. However, suppose we switch the order of mode change, namely, we change the mode of $TokenNav_{\$x,p2}\$y$ first and that of $NodeNav_{\$u,p1}\$v$ next. The mode change of $TokenNav_{\$x,p2}\$y$ eliminates $StructuralJoin_{\$x}$. This $Extract$ operator will not be introduced back by the mode change of $NodeNav_{\$u,p1}\$v$ since no other operator exists in the format of $TokenNav_{\$u,p1'}\$v'$ in the plan. Therefore $StructuralJoin_{\$x}$ operator appears in neither plan regardless of the order in which we change the modes.

In summary, we prove that the order in which we change the modes of a $NodeNav$ and a $TokenNav$ has no impact on the set of operators appearing in the final plan.

Appendix F

Proof of Same Cost Changes

In Figure F.1, given a plan P_1, we get two plans P_2 and P_3 by changing the modes of $navOp_1$ and $navOp_2$ in P_1 respectively. Suppose we now change the mode of $navOp_1$ in P_3 and get a new plan P_4. We want to prove that if $moveScope(navOp_1)$ $\cap$ $moveScope(navOp_2) = \emptyset$, $Cost(P_4) - Cost(P_3) = Cost(P_2) - Cost(P_1)$.

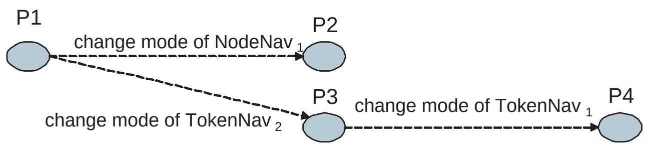


Figure F.1: $Cost(P_4) - Cost(P_3) = Cost(P_2) - Cost(P_1)$

Proof 8 *For simplicity, we use $DestSJ$ and $ConfineSJ$ to represent the destination and confining $StructuralJoin$ operators of a $navOp$. We now consider the case when $navOp$ is in the format of $TokenNav_{\$u,p}\$v$. After the rewriting, the input subplans of below $StructuralJoin$ operators can have changed costs. First, the $DestSJ$ of $TokenNav_{\$u,p}\$v$, i.e., $StructuralJoin_{\$v}$, is eliminated so* *that the costs of input subplans of this $DestSJ$ is now 0. For example, in Figure 3.2, the $DestSJ$ of $TokenNav_{\$a,/seller}\$b$, i.e., $StructuralJoin_{\$b}$, is eliminated in Figure 3.6 when $\$a/seller$ is pulled out. Second, since the $DestSJ$ was an entry operator of an input subplan of $StructuralJoin_{\$u}$, its elimination changes the contents (and correspondingly the cost) of input subplans of $StructuralJoin_{\$u}$. Third, the costs of input subplans of the $ConfineSJ$ are changed as well since the new $NodeNav_{\$u,p}\$v$ operator is added into an input subplan of the $ConfineSJ$. Fourth, for an intermediate $StructrualJoin$ between the $DestSJ$ and $ConfineSJ$, although no operators are removed or added into its input subplans, its costs are still changed. This is because $TokenNav$ before is a descendant but now an ancestor of an entry operator of one input subplan. The selectivity of this entry operator may increase which affects the overall cost of the input subplans to these $StructuralJoin$ operators. We therefore have,*

$$Cost(P_2) - Cost(P_1) = \sum_{sj \in moveScope(TokenNav_1)} (\underbrace{cost\ of\ input\ subplans\ to\ sj\ in\ P_2}_{(a)} - \underbrace{cost\ of\ input\ subplans\ to\ sj\ in\ P_1}_{(b)}) - \underbrace{cost\ of\ TokenNav_1\ in\ P_1}_{(c)}.$$

Similarly, $Cost(P_4) - Cost(P_3) = \sum_{sj \in moveScope(TokenNav_1)} (\underbrace{cost\ of\ input\ subplans\ to\ sj\ inP_4}_{(d)} - \underbrace{cost\ of\ input\ subplans\ to\ sj\ in\ P_3}_{(e)}) - \underbrace{cost\ of\ executing\ TokenNav_1\ in\ P_3}_{(f)}.$

P_4 can be derived by moving out $TokenNav_2$ from the automata in P_2. Since $moveScope(TokenNav_1) \cap moveScope(TokenNav_2) = \emptyset$, $DestSJ$ of $TokenNav_2$ $\notin moveScope(TokenNav_1)$ in P_2 and the new $NodeNav$ rewritten from $TokenNav_2$ $\notin moveScope(TokenNav_1)$ in P_4. Therefore for any $sj \in moveScope(TokenNav_1)$ in P_4, its input subplans are the same as the input subplans of sj in P_2. Moreover, since it is impossible that $TokenNav_2$ is an descendant of any $sj \in moveScope(TokenNav_1)$

in P_2 but the rewritten $NodeNav$ is an ancestor of sj in P_4 (otherwise $moveScopt(TokenNav_1)$ $\cap$ $moveScope(TokenNav_2) \neq \emptyset$), the input subplans of sj in both P_2 and P_4 process the same amount of input data. Therefore, the costs of input subplans of any sj in P_2 must equal to those in P_4. In short, item (d) = item (a). Similarly, we have item (e) = item (b). Also, item (f) = item (c). Finally, we have $Cost(P_2) - Cost(P_1) = Cost(P_4) - Cost(P_3)$.

Bibliography

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

[2] A. Aboulnaga, A. R. Alameldeen and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proceedings of VLDB*, 2001.

[3] A. Aboulnaga and J. F. Naughton. Building XML Statistics for the Hidden Web. In *CIKM*, pages 358–365, 2003.

[4] A. Das, J. Gehrke, M. Riedewald. Approximate Join Processing Over Data Streams. In *Proceedings of VLDB*, pages 40–51, 2003.

[5] A. Gupta and S. Chawathe. Skipping Streams with XHints. Technical Report CS-TR-4566, University of Maryland, College Park, 2004.

[6] A. Halverson and J. Burger and L. Galanis et al. Mixed Mode XML Query Processing. In *Proceedings of VLDB*, 2003.

[7] 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.

[8] A. Snoeren, K. Conkey and D. Gifford. Mesh-based Content Routing using XML. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[9] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[10] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination. In *Proceeding of VLDB*, pages 53–64, 2000.

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

[12] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, Aug/Sep 2004.

[13] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272, June 2000.

[14] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002.

[15] S. Babu and J. Widom. Continuous queries over data streams. In *ACM SIGMOD*, Sep 2001.

[16] B.Choi. What are Real DTDs like, 2002.

[17] 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.

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

[19] C. L. Monma and J. B. Sidney. Sequencing with Series-Parallel Precedence Constraints. In *Mathematics of Operations Research*, pages 4: 215 – 224, 1979.

[20] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, August 2002.

[21] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, 2003.

[22] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, June 2002.

[23] D. Abadi and Y. Ahmad and M. Balazinska and et. al. The design of the borealis stream processing engine. In *Proceedings CIDR*, page to appear, 2005.

[24] 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.

[25] D. Florescu, C. Hillery and D. Kossmann et al. The BEA streaming XQuery processor. In *VLDB Journal 13(3)*, pages 294–315, 2004.

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

[27] D. Kossmann and K. Stocker. Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. In *ACM Transaction on Database System 25 (1)*, pages 43 – 82, 2000.

[28] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Yong Yao. The Cougar Project: A Work-In-Progress Report. In *Sigmod Record 32 (4)*, pages 53–59, 2003.

[29] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT Logical Framework for XQuery. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 29–41, 2004.

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

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

[32] Leonidas Fegaras. The Joy of SAX. In *First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2004.

[33] George Russell, Mathias Neumuller and Richard Connor. Stream-based XML Processing with Tupe Filtering. 2003.

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

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

[36] H. Jiang, H. Lu and W. Wang. Holistic twig joins on indexed XML documents. In *VLDB*, 2003.

[37] H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. In *SIGMOD*, 2000.

[38] H. Su, E. A. Rundensteiner and M. Mani. Raindrop: An XQuery Engine over XML Streams - on Semantic Query Optimization (demonstration). In *VLDB*, 2004.

[39] 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.

[40] Hong Su and Elke A. Rundensteiner and Murali Mani. Automaton Meets Algebra: A Hybrid Paradigm for XML Stream Processings. *DKE Journal*, 2006.

[41] Hong Su, Elke A. Rundensteiner, Murali Mani. Semantic Query Optimization for XQuery over XML Streams. In *VLDB Proceedings*, 2005.

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

[43] J. Chen, D. Dewitt, F. Tian et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

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

[45] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *SIGMOD*, pages 267–276, 1993.

[46] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. In *VLDB Journal Volume 11 Issue 4*, pages 274–291, 2002.

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

[48] N. Kabra and D. Dewitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*, 1998.

[49] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[50] L. Lim, M. Wang and J. Vitter. SASH: A Self-Adaptive Histograms Set for Dynamically Changing Workloads. In *VLDB*, 2003.

[51] L. Lim, M. Wang and S. Padmanabhan et. al. An On-line Self-Tuning Markov Histogram for XML Path Selectivity Estimation. In *VLDB*, 2002.

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

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

[54] M. J. Carey, M. Blevins and P. Takacsi-Nagy. Integration, Web Services Style. In *IEEE Data Eng. Bull. 25 (4): 17-21*, 2002.

[55] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, Feb 2002.

[56] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the 27th VLDB Conference, Edinburgh, Scotland*, pages 241–250, 2001.

[57] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 315–326, 1999.

[58] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML Pattern Matching. In *SIGMOD*, 2002.

[59] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA*, pages 437–448, May 2001.

[60] University of Washington. Xml data repository, 2002.

[61] P. Hart, N. Nilsson and B. Raphael. A Formal Bais for the Heuristic Determination of Minimum Cost Paths. In *IEEE Transactions on Systems Science and Cybernetics SSC4 (2)*, pages 100 – 107, 1968.

[62] P. Mukhopadhyay and Y. Papakonstantinou. Mixing querying and navigation in mix. In *Proceedings of ICDE 2002*, 2002.

[63] P. Selinger, M. Astrahan and D. Chamberlin. Access Path Selection in a Relational Database Management System. In *IEEE COMPSAC*, 1979.

[64] P. Tolani and J. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *ICDE*, pages 225 – 234, 2002.

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

[66] 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.

[67] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, pages 1353–1356, 2004.

[68] S. C. Yoon, I. Y. Song and E. K. Park. Semantic Query Processing in Object-Oriented Database Using Deductive Aproach. In *Proceeding of CIKM*, pages 150–157, 1995.

[69] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, 1998.

[70] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *VLDB*, 1994.

[71] S. Wang, E. A. Rundensteiner and M. Mani. Optimization of nested xquery expressions with orderby clauses. In *XML Schema and Data Management (XSDM)*, Tokyo, Japan, April 2005.

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

[73] Nesime Tatbul, Ugur etintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 309–320, 2003.

[74] 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.

[75] W. Scheufele and G. Moerkotte. Efficient Dynamic Programming Algorithms for Ordering Expensive Joins and Selections. In *EDBT*, 1998.

[76] W3C. XML Query Data Model. http://www.w3.org/TR/query-datamodel, 2000.

[77] 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.

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

[79] X. Zhang, B. Pielech and E. A. Rundensteiner. XAT Optimization. Technical Report WPI-CS-TR-02-25, Worcester Polytechnic Institute, 2002.

[80] Y. Diao, M. Altinel and M. J. Franklin, H. Zhang and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. In *TODS*, pages 467–516, 2003.

[81] Y. Diao, P. Fischer, M. J. Franklin, R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of ICDE*, pages 341–344, 2002.

[82] Y. Wu, J. M. Patel and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, pages 443–454, 2003.

[83] 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.

[84] Z. Chen, H.V. Jagadish and F. Korn et al. Counting Twig Matches in a Tree. In *Proceedings of ICDE*, 2001.

[85] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, pages 431–442, June 2004.