2003-08-27

Efficient XML Stream Processing with Automata and Query Algebra

Jinhuj Jian
Worcester Polytechnic Institute

Efficient XML Stream Processing with Automata and Query Algebra

by

Jinhui Jian

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

August 2003

APPROVED:

Professor Elke Rundensteiner, Thesis Advisor

Professor Kathi Fisler, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

XML Stream Processing is an emerging technology designed to support declarative queries over continuous streams of data. The interest in this novel technology is growing due to the increasing number of real world applications such as monitoring systems for stock, email, and sensor data that need to analyze incoming data streams. There are however several open challenges. One, we must develop efficient techniques for pattern matching over the nested tag structure of XML as data streams in token by token. Two, we must develop techniques for query optimization to cope with complex user queries while given only incomplete knowledge of source data. When considering these challenges separately, then automata models have been shown by several recent works to be suited to tackle the first problem, while algebraic query models have been regarded as appropriate foundations to tackle the second problem. The question however remains how best to put these two models together to have an overall effective system. This thesis aims to exactly fill this gap.

We propose a unified query framework to augment automata-style processing with algebra-based query optimization capabilities. We use the automata model to handle the token-oriented streaming XML data and use the algebraic model to support set-oriented optimization techniques. The framework has been designed in two layers such that the logical layer provides a uniform abstraction across the two models and any optimization techniques can be applied in either model uniformly using query rewritings. The physical layer, on the other hand, allows us to refine the implementation details after the logical layer optimization. We have successfully applied this framework in the Raindrop stream processing system. We have identified several trade-offs regarding which query functionality should be realized in which specific query model. We have developed novel optimization techniques to

exploit these trade-offs. For example, a query rewrite rule can flexibly push down a pattern matching into the automata model when the optimizer decides that it is more efficient to do so. To deal with incomplete knowledge of source data, we have also developed novel techniques to monitor data statistics, based on which we can apply optimization techniques to choose the optimal query plan at runtime. Our experimental study confirms that considerable performance gains are being achieved when these optimization techniques are applied in our system.

Acknowledgements

I have been a big fan of the notion that laziness is the father of creation, as I would be one of the greatest creators under such an assumption. However, this kind of *creativity* is not enough for a master's thesis, let alone the training of a master student. For this reason, I would like to thank my advisor, Prof. Elke A. Rundensteiner, for her help, guidance, and most importantly, the encouragement that keeps me on track and stimulates me to continue for a higher goal.

I would also like to thank the entire DSRG research group, especially those involved in the Rainbow and the Raindrop projects. I greatly enjoy their ideas and their enthusiasm. Without their discussion, many ideas in this thesis would never come out.

I thank my reader, Prof. Kathi Fisler, for her considerate reading of this thesis and the feedbacks that help clarify many of my hidden assumptions.

Finally, I thank my family and my girl friend Zheng for their constant support and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation of XML Stream Processing

Starting from the ground-breaking research paper of Codd [Cod70], database theory has matured and systems based on this theory have acquired industrial strength. The relational database schemas and entity relationship have served extremely well in most theoretical researches and commercial applications. Until recently, however, the database literature has only focused on querying data assumed to be kept on a central local storage. This is not necessarily a shortcoming because it fits well in the client-server architecture and provides a closed and stable environment for the query optimizer, which enables high performance processing.

The emergence of the Internet, however, has changed the assumed computational architecture. Both computing resources (e.g., CPU and storage) and information resources (e.g., relational tables) are now widely distributed in geographic space yet tightly connected via the Internet. There is a clear demand for the next generation database systems to process or to query data that is stored at a remote site or is received through the network.

While the Internet has been developed to resolve the geographical differences

of computers, the XML [W3C] has been developed to resolve the logical differences of information, i.e., heterogeneous data models and divergent data schemas. These logical differences have been the major obstacle for sharing data from different databases. Now the self-contained schema-less XML data model allows heterogeneous data to be represented in a uniform fashion. Because XML has been widely accepted as the *wired format* for data [ABS00], we can assume that efficient processing of XML data streams will be the major issue for the next generation database systems.

In fact, combining the power of the Internet and the XML language has lead to numerous new applications. The *Selective Dissemination of Information* applications, for example, involve timely distribution of data to a large set of customers, and include stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery [AF00]. The *DBMS-Active, Human-Passive (DAHP) Model* [CCC+02], for another example, allows a system to monitor large scale sensor networks connected through computer networks [CCC+02]. All these new applications require computing paradigms different from the traditional database management systems that assume local data storage. This thesis aims to tackle exactly this new challenge.

Figure 1.1 illustrates the general XML Stream Processing (XSP) scenario. A set of *input data streams* is transmitted in the XML format by data providers through computer networks. *User queries* are registered (i.e., user queries are specified and stored in the system during the entire execution) to specify user interests. The *XSP engine* is responsible for evaluating the user queries over the input data streams and for generating the expected result.

One of the major challenges in XSP comes from the fact that data is stored from a remote site that is beyond the control of the processing engine. It has several implications on the design of the XSP engine. First, the input data is

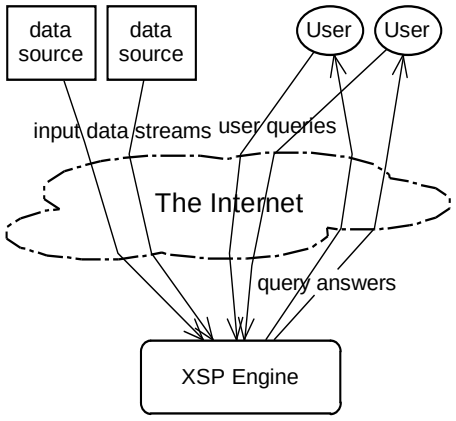

Figure 1.1: An XSP system

continuously streamed into the system, but when exactly a specific portion of the data will be received is unknown. This is due to the turbulence of remote computers and computer networks. Second, data statistics (e.g., data size and data value distribution) are also unknown beforehand. This means the query optimizers, which largely rely on precise data statistics, will not work as well as on local storage. Third, many applications of XSP have an implicit preference of prompt response, which requires the query engine to process data on-the-fly while the data is being transmitted.

1.2 Motivation of the Raindrop Approach

To tackle the challenges described in Section 1.1, automata-based approaches have been adopted in a number of recent research papers [AF00, DFFT02, ILW00, CFGR02, MGOS03, LMP02, GS03, PC03]. While these works have shown that automata-based models are suited for XPath-like pattern retrieval over token-based XML data streams, we find these automata-based approaches suffer from being not as flexibly optimizable (e.g. for performance) as, for example, traditional database systems that

are based on query algebras [Cod70, A+76, Cha98]. Query algebras have also been applied in querying XML data [CFI+00, BFHR02, ZPR02, JAKL+02]. Although these works have not focused on querying stream data per se, their strengths in processing complex queries (e.g., in the XQuery language) and optimization are certainly needed in our system.

We thus now propose to integrate both the automata model and the query algebra model into one system to leverage the strengths inherent in either of them. This is appropriate because the automata-based approach can process XML patterns on the fly but lacks query optimization techniques, while the algebra-based approach is strong in query optimization but lacks the capability of on-the-fly pattern processing.

Beyond the basic idea of integrating the both models, we propose a unified framework that allows us to apply optimization techniques uniformly in the automata model, in the algebraic model, and even crossing the boundary of the two models (see Chapters 4 and 5). This unified framework allows us to reason about query logic and optimization at the algebraic level, and only thereafter play with the implementation details specific to the automata or to the general query plan. This is to be compared with the Tukwila data integration system [IHW02] that also attempts to integrate automata and query algebra. But the integration in Tukwila is assumed to be fixed at compile time and thus cannot exploit the full advantages of this powerful dual query model (see Chapters 4 and 5).

In brief, the contributions of this thesis include:

1. We have identified the need for both the automata model and the algebraic model in querying XML data streams. We have also discussed the challenges in designing such a hybrid system with both query models.

2. From analyzing the structure of XQuery expressions, we propose a two-tier

system design to integrate both the automata model and the algebraic model for XQuery evaluation.

3. Based on the two-tier system design, we have developed an efficient framework for XSP, in which automata- and algebra-based models are flexibly integrated. The framework is designed in two layers such that the logical layer allows us to reason about query logic at the algebraic level and the physical layer describes the implementation details.

4. We compare the bottom-up and top-down approaches in evaluating XQuery binding trees. Thereby we develop a number of novel optimization techniques that can move query functionalities into or out-off the automaton, which are not available in any previous work.

5. We have designed techniques that allow a part of the automaton to be modified safely during execution, such that the above optimization techniques can be applied at runtime.

6. We have developed a working prototype system using Java SDK 1.4.

7. We have conducted experimental studies with the prototype system to evaluate the above framework and optimization techniques. The results confirm the merits of our framework and optimization techniques.

1.3 Outline of Thesis

The rest of the thesis is organized as follows. Chapter 2 provides background on XML and the XQuery language. It also specifies the notations that are used through this thesis. Chapter 3explains the rationale behind the Raindrop framework while

Chapter 4 presents the framework in detail. Chapter 5 introduces compile time optimization, while Chapter 6 explains applying the optimization techniques at runtime. Chapter 7 presents our experimental studies. Chapter 8 discusses other related research work. Chapter 9 concludes the thesis and presents potential future directions.

Chapter 2

Background

2.1 The XML Language

The Extensible Markup Language (XML) is a simple and flexible text format derived from the Standard Generalized Markup Language (SGML) [W3C]. It is a new standard adopted by the World Wide Web Consortium (W3C) to complement HTML for data exchange on the Web [ABS00].

In its basic form, XML is simply a format to encode data. The basic logical component in XML is the *element*, which is represented in text format as a piece of text bounded by an open tag (such as <book>) and a matching close tag (such as </book>), as in Figure 2.1. The *well-formedness* property requires that the open tags and the close tags must be balanced and properly nested. This property ensures that an XML element can be logically represented using a *node-labeled tree*. In a node-labeled tree, a node represents an element and is labeled with the element's name. An edge represents the *parent-child* relationship between two elements, i.e., the child element is directly nested in the parent element (see Figure 2.2).

The node-labeled tree view (or simply the tree view) has been widely accepted as the "default" representation for XML. It is, however, inappropriate to equate these

```
<bib>
   <book year="1994">
      <title>TCP/IP Illustrated</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price> 65.95</price>
   </book>

<book year="2000">
      <title>Data on the Web</title>
      <author><last>Abiteboul</last><first>Serge</first></author>
      <author><last>Buneman</last><first>Peter</first></author>
      <author><last>Suciu</last><first>Dan</first></author>
      <publisher>Morgan Kaufmann Publishers</publisher>
      <price>39.95</price>
   </book>

 <book year="1992">
      <title>Advanced Programming in the Unix environment</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price>65.95</price>
   </book>

</bib>
```

Figure 2.1: An example XML document

two concepts. XML by itself is a format to encode data, and no more. The tree view is *one* logical representation of the XML format. An alternative is introduced in the Simple API for XML (SAX) [sax]. SAX is a set of abstract programmatic interfaces that project an XML document onto a stream of well-known method calls. Each method call corresponds to a particular part of the XML document, which is called an *XML token*. Hence in SAX, an XML document is viewed as a stream of tokens. This stream-of-token view, being linear, is very different from the tree view.

2.2 The XPath Language

The XML Path Language (XPath) is defined by the W3C for addressing parts of an XML document. XPath expressions are defined against a document's node-labeled tree view to identify a set of nodes.

Most XPath expressions are of the form "/axis::node/axis::node/axis::node...",

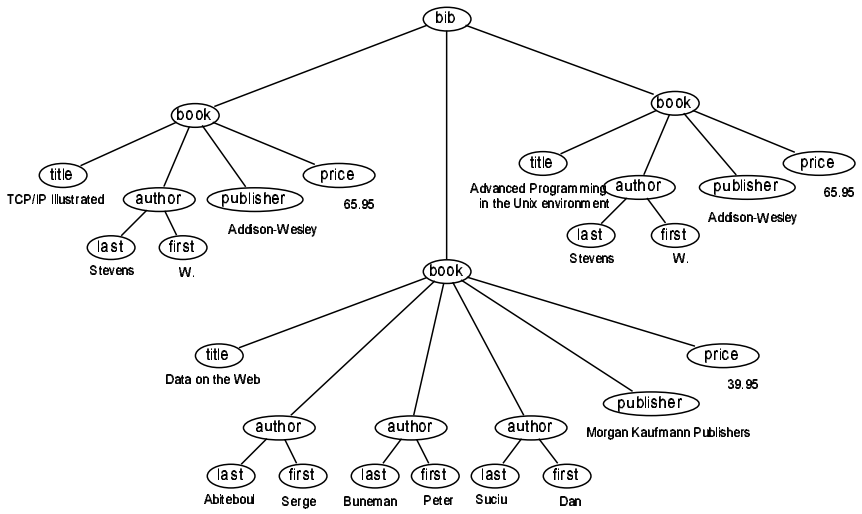

Figure 2.2: The node-labeled tree view of the example XML document

as in the query "/child::bib/child::book/child::title", which is called a *location path* or simply *path*. A path defines a traversal over the node-labeled tree. A path can be either an *absolute path* or a *relative path*. The former starts with "/" that stands for the root and its evaluation leads to a traversal from the root element of the node-labeled tree. A relative path does not starts from the root. Instead, a collection of XML elements must be specified by the context in the evaluation serving as the starting point of the traversal. Each step in a location path, called a *location step*, is separated by the symbol "/". A location step consists of an *axis identifier* and a *node test*, separated by "::". The *simplified notation* of an XPath uses "/" as a shorthand for the child axis and "//" for the descendant axis, as in "/bib/book/title".

The semantics of an XPath expression can be defined by the *default* algorithm. Here location steps are evaluated in order one at a time. Each location step is evaluated against the nodes in the *context node-set*. At initialization, the context node-set is defined as consisting of the root node for an absolute path, or the context node-set is given beforehand for a relative path. The resulting node set then serves

as the next context node-set for the next position step. This process continues until the final position step is evaluated. The resulting node set is defined as the result of the entire expression. For example, applying the query "/bib/book/title" on the document in Figure 2.1 will result in three title elements.

2.3 The XQuery Language

The XQuery Language is defined by the W3C to query a broad spectrum of XML information sources, including both databases and documents [W3C02b]. XQuery is designed to be a general query language that supports rich functionalities such as selection, projection, join, and aggregation.

Query:

```
<result>{
FOR $b in doc("bib.doc")//book
WHERE $b/price > 50
RETURN
    <expensive_title>
        $b/title/text()
    </expensive_title>
}</result>
```

Result:

```
<result>
    <expensive_title>TCP/IP Illustrated</expensive_title>
    <expensive_title>Advanced Programming in the Unix environment</expensive_title>
</result>
```

Figure 2.3: An example XQuery and its result

The basic building block of XQuery is the *expression*. A frequently used expression is the *FLWR* expression, in which the *for* and the *let* clauses bind variables to sequences of XML nodes or atomic values, the *where* clause selects the bound values based on predicates, and the *return* clause formats the bound values and constructs the final results. For example, Figure 2.3 shows an example FLWR query and its result when it is evaluated over the XML document in Figure 2.1. In addition,

XQuery is a functional language which allows various kinds of expressions to be nested with full generality. This allows very complex queries to be constructed from composition.

The semantics of an XQuery expression can be defined by the *default* algorithm that evaluates expressions and clauses one at a time. The default algorithm works for any query. In some situations, however, more efficient algorithms together with various query optimization techniques can tremendously accelerate the evaluation process. In these cases, finding more efficient algorithms and more effective optimization techniques is the core task of a general query engine.

2.4 Notations

We shall adopt the following notations in Table 2.1 through this thesis.

Notation	**Description**
<x>, </x>	XML element tags
$\langle f_1, f_2, ... \rangle$	A tuple consisting of a set of fields each with an implicit binding name (as the *name perspective* from [AHV95])
$t_1 \circ t_2$	Tuple concatenation
$[e_1, e_2, ...]$	A list consisting of elements e_1, e_2,...
$[e]$	A singleton list consisting of a single element e
$e \leftarrow list$	A generator of a list, i.e., an iterator. e is a variable iterated over the elements of the list

Table 2.1: Notations

Chapter 3

Preliminary Analysis

3.1 Overview

This chapter serves as a preparation for a detailed discussion of the Raindrop framework. We shall explain the rationale behind the framework in an intuitive manner. The next chapter (Chapter 4) will adopt a more formal and complete treatment.

The Raindrop framework aims to offer efficient XQuery evaluation over continuous XML data streams. Besides those general concerns for a DBMS such as parsing and evaluating user queries, it faces other challenges that are specific to XML and to stream processing. One such challenge is the integration problem of the automata model and the query algebra model.

Automata are state-transition models that have been shown to be effective for on-the-fly XML pattern retrieval [AF00, DFFT02, ILW00, CFGR02, MGOS03, LMP02, GS03, PC03]. A state in such a model represents which location step in an XPath expression, or a *pattern*, that is being matched. An input symbol represents an XML token drawn from an XML data stream, which can be an open tag, a close tag, or a PCDATA. This mapping is straightforward and efficient. However, when such a model is applied to XQuery evaluation, which is more complex than evaluating an

XPath expression, it fails to support optimization techniques that can boost performance. On the other hand, query algebras have been used in relational databases specifically for performance improvement. It is thus appropriate that we should combine the strengths inherited from both. There are, however, a few challenges that must first be tackled.

As mentioned above, automata are based on a token-based data model. Tokens are totally ordered and there is no query semantics for an individual token unless it is placed together with others. Also, automata imply a token-driven control model, that is, each input token will immediately trigger a predefined sequence of actions. In contrast, query algebras such as the relational algebra [Cod70] adopt a tuple-based data model. A tuple is a self-contained data structure that has query semantics independent from other tuples. It can be individually processed by a query operator without being related to other tuples. Also, the token-driven control model is rigid compared to the variety of more flexible control strategies that query algebras allow [M$^+$03]. For instance, a scheduler can choose a particular query operator to run at a particular moment in favor of performance. Such scheduling decisions can be made based on global properties including data statistics and runtime system resources, which is not allowed in the token-driven strategy.

To combine both techniques means to resolve their differences in both the data model and the control model. We approach this integration problem by using query algebra as the overall representation for optimization and evaluation and applying automata only as implementation techniques encapsulated inside individual query operators. The advantage is that we can now reason about query logic at the algebraic level and only thereafter play with implementation details. It is our goal to bring the strengthes of algebraic systems to XML stream processing. It is thus not surprising that our query algebra largely resembles those in relational databases and especially those in native or extended-relational XML databases. Hence the

algebra by itself is not of particular interest here. We shall instead focus on the integration part.

3.2 Analysis of XQuery Evaluation

3.2.1 Pattern Retrieval in XQuery

Pattern retrieval is a specific problem in querying XML. XML has a paired and nested tag structure, which can be visualized as a tree-like structure, i.e., a document tree (see Chapter 2). With this tree-like representation, a specific element or a specific part of a document can be addressed by a path starting from the root of the tree leading to the target element. The XPath language is such a path language that has been standardized by W3C (see Chapter 2).

```
FOR $a in doc("foo.xml")/r/a
WHERE $a/b > 0
RETURN $a/c
```

(a)

```
FOR $a in doc("foo.xml")/r/a
LET $b := $a/b, $c := $a/c
WHERE $b > 0
RETURN $c
```

(b)

Figure 3.1: Moving variable bindings ahead. Because XQuery is a declarative language, the differences of these two queries in syntax do not imply any difference in actual evaluation strategy. However, before we discuss the differences in detail at the algebraic level (Chapter 4), here we give a hint that more than one strategy is possible to evaluate a query

The XQuery language uses XPath to address locations of XML elements . For example, Figure 3.1 (a) is a typical FLWR query that contains three XPath expressions, i.e., "/r/a", "$a/b", and "$a/c". Each XPath expression in an XQuery represents a *binding*. A binding can have an explicit binding name, or a *binding variable*, e.g., $a in "$a in doc(foo.xml)/r/a". Or it can be implicit, e.g., "$a/b".

For ease of reference, we may give such implicit bindings explicit names, as in Figure 3.1 (b). In fact, this naming process can be viewed as collecting all XPath expressions at the head of a query. Hence a query is now divided into two parts: the variable bindings and the rest.

In Figure 3.1 (b), the $a binding is different from the $b binding. The right-hand-side of the $a binding, or the *associated expression*, contains no other variable. We call such a binding an *independent binding*. In contrast, the associated expression of the $b binding contains other variables, i.e., $a. We call such a binding a *dependent binding*.

When all bindings in Figure 3.1 (b) are viewed as a whole, it forms a *binding tree*, where the independent binding (i.e., $a) is the root and the dependent bindings (i.e., $b and $c) are inner nodes. In terms of evaluating an XQuery, a binding tree is used to produce a tuple stream in which each tuple consists of one or more bound variables. For example, a tuple for the binding tree in Figure 3.1 may be of the form $\langle \$a, \$b, \$c \rangle$.

Another difference between the $a binding and the $b binding is that the former is defined in a *for* clause while the latter is defined in a *let* clause. We call them *unnested binding* and *nested binding*, respectively. This distinction defines the contents in the output tuple stream of a binding tree. For example, given a collection of three elements, an unnested binding would produce three tuples each contains one element, while a nested binding would produce only one tuple that contains all three elements.

Given the document in Figure 3.2, the output tuple stream contains two tuples: $\langle a1, [b1, b2], [c1, c2, c3] \rangle$ and $\langle a2, [b3], [c4, c5] \rangle$. In other words, an unnested binding binds to a single element in each tuple while a nested binding binds to a sequence of elements. The unnested binding can be viewed as a *join* condition based on which the nested bindings are partitioned and merged.

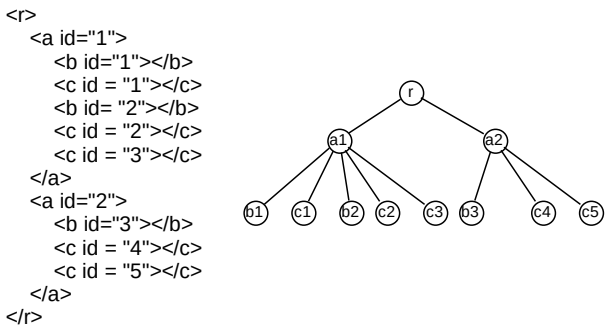

Figure 3.2: An example document

There is yet another difference between the $a binding and the $b binding. $a is never used in the query except being the *depending variable* for the $b and the $c bindings. We call such a binding a *supporting binding*. In contrast, $b is used in the *where* clause and $c is used in the *return* clause. We call such a binding a *principal binding*.

The supporting bindings in the output tuple stream of a binding tree are redundant. Hence for Figure 3.1(b) a tuple can simply have the form $\langle \$b, \$c \rangle$. In this situation the *join* nature of a supporting and unnested binding is especially obvious.

```
FOR $a in doc("foo.xml")/r/a
WHERE $a/b > 0
RETURN {
    FOR $c in $a/c
    WHERE $c/d > 0
    RETURN $c/e
}
```

(a)

```
FOR $a in doc("foo.xml")/r/a,
      $c in $a/c
LET $b := $a/b,
      $d := $c/d,
      $e := $c/e
WHERE $b > 0 AND $d > 0
RETURN $e
```

(b)

Figure 3.3: Moving variable bindings ahead in a nested query. Again, the differences in syntax imply no actual differences in evaluation strategy. Instead, detailed discussion at the algebraic level will be presented in Chapter 4

Next we show another example to illustrate the above concepts. The query in Figure 3.3 (a) contains a nested query where the subquery can be pulled up and

merged with the outer query, as in Figure 3.3 (b). The output tuple stream of this query should be of the form $\langle \$b, \$d, \$e \rangle$ (after dropping the supporting bindings).

3.2.2 Modeling Pattern Retrieval

A *pattern tree* generally contains more than one variable binding, as $a, $b, and $c in Figure 3.1 (b). One way to model these bindings is to put them all together into one single query operator, as in Tukwila [IHW02]. This is straightforward because the tree-like structure of an automaton diagram closely resembles a pattern tree. However, it is rather rigid because the set of patterns in an operator is fixed beforehand and is not applicable to certain optimizations, as we will discuss below.

As suggested in [W3C02b], the purpose of the *for* and the *let* clauses in an XQuery is to produce a tuple stream in which each tuple consists of one or more bound variables. This means the output of evaluating a pattern tree should be a tuple stream. However, there is more than one way to generate the tuple stream. A top-down analytic approach first evaluates the root variable and then identifies its descendants by navigating through the pattern tree down to the leaf variables. In contrast, a bottom-up synthetic approach first evaluates the leaf variables and then constructs the higher level variables by climbing the pattern tree up to the root variable. These different approaches have different performance characteristics, i.e., one may be better than another in some settings but vice versa in other settings (see Chapter 5).

We model pattern retrieval in such a way that both approaches are supported and one approach can be flexibly switched into the otherf via a unified optimization process. In short, we use one operator to represent each variable binding (more details in Chapters 4 and 5). In a top-down analytic approach, we use an *extract* operator to evaluate the root variable. Other variables are then evaluated by the

navigate operators. In a bottom-up synthetic approach, the *extract* operators are used to evaluate all leaf variables. Other variables are then evaluated via the *sjoin* operators.

3.2.3 Using Automata for Pattern Retrieval

We have been vaguely using the term "automata" to mean "state transition machines". We now limit our scope to NFA augmented with a runtime stack. Strictly speaking, this corresponds to a PDA-equivalent, but we stick to the terminology of NFA and put aside the runtime stack as an auxiliary structure, mainly because the notation is simpler and more closely related to an XPath expression.

Because XPath is essentially a regular expression, constructing an NFA to recognize a given pattern is exactly like in most standard texts [HMU01]. The example in Figure 3.4 shows an NFA constructed to recognize the XPath "//a/*/b".

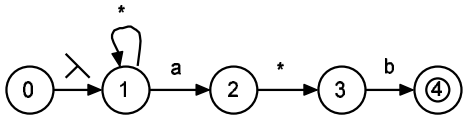

Figure 3.4: An automaton constructed to recognize "//a/*/b"

The input to an NFA is an XML data stream consisting of a sequence of XML tokens, which can be an open tag, a close tag, or a PCDATA. The runtime configuration of an NFA consists of a set of *current states* and the auxiliary *stack*. Initially, a *start state* is put into the *current set* and the *stack* is set to empty. The tokens are read in one at a time.

- When an open tag is read in, the tag is compared with all outgoing transitions from all current states. All matched transitions are fired; the resulting states are activated and are put into the new current set. The previous current set is

put onto the stack. If no matched transition is found, an empty set becomes the current set.

- When a close tag is read in, the top element of the stack, i.e., the previous current set, is popped and becomes the new current set. Hence the states in the current set are activated for the second time.

- When a PCDATA is read in, the states of the NFA remain unchanged.

This process is very similar to that of a standard NFA, except that a stack is used to handle the paired and nested tag structure. Each accepting state will be activated twice for each matching element, one for the open tag and one for the close tag. We shall refer to the two activations as *open match* and *close match*, respectively. In fact, we can view the automaton as a *matching machine* that outputs matching events when matching elements are found, then another module, *operator stem* as defined below, will extract the specific data from the input stream according to the matching events. As will be discussed below, such a matching machine is adopted as part of *automata operators*, e.g., the *extract* operator and the *sjoin* operator.

The Extract Operator

An *extract* operator retrieves proper elements in an XML data stream according to a given XPath expression. In part, this corresponds to a *scan* operator for a relational table access. The difference is that a relational table is generally stored on a local disk and thus can be retrieved simply by reading certain disk sectors. An *extract* operator, however, must analyze an input stream, match the given pattern, and extract specific parts. This is a lot like recognizing a regular expression with an automaton. In fact, we shall use a matching machine to implement an *extract* operator. Like all *automata operators*, an *extract* operator consists of two parts: the *operator stem* and its *associated automaton*, i.e., the matching machine.

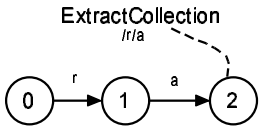

Figure 3.5: A partial plan resolving the *for* clause in Figure 3.1(b). The dashed line connects the associated automaton and the operator stem, which passes the matching events

The input to an extract operator is an XML token stream while the output is an XML element stream. This task can be divided into two parts, matching the input tokens and constructing the output elements. The matching part is done through an associated automaton. As discussed above, the matching machine will notify when the open tag (open match) and the close tag (close match) are read in. Specifically, an XML element is a string starting from its open tag and ending with its close tag. Hence the operator stem can simply start collecting incoming data with the open match and stop with the close match.

Consider the partial query plan in Figure 3.5. Suppose it is run through the XML document in Figure 3.2. Suppose now the open tag "<a>" of the first "a" element is being read in, and we are in state 2. It is easy to conclude that whatever next open tag is encountered (e.g., "<b id="1">), it must be a child of "a". In general, a *context node* is maintained during runtime and is set to the document root at initialization time. Every incoming open tag is connected to the *context node* and becomes the new *context node*; every incoming close tag resets the *context node* to the previous *context node* by referring to a *context stack*.

In general, when the accepting state is activated by the open tag, the process of collecting and connecting incoming tokens is started. When the final state is activated by the close tag, we would have assembled a complete element. This element will then be wrapped into an output tuple. In fact, we can separate the

element-assembling process from the operator. We can use a centralized approach, via a storage manager, to store the retrieved element (more details in Section 4.3).

The SJoin Operator

The *sjoin* operator is a special *join* operator that joins input tuples based on some structural relationship, e.g., the parent-child relationship. Let us consider again the query in Figure 3.1 (b). Recall that the output tuple stream should be of the form $\langle \$b, \$c \rangle$. A bottom-up synthesis approach will first generate two tuple streams separately for the $b binding and the $c binding, repectively. The two tuple streams are then joined by their parent-child relationship, i.e., each joined tuple should contain the "b" and "c" elements belong to the same "a" parent. A query plan for this approach is shown in Figure 3.6.

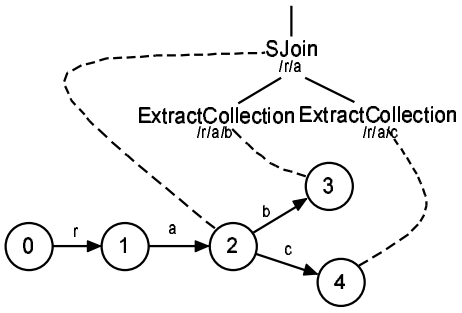

Figure 3.6: A SJoin operator and its associated automaton

Structural join is a very important operation in XML processing [JAKL+02]. Naive algorithms may involve a large number of comparisons. Here we present the JIT (Just In Time) algorithm, which exploits the sequentiality of the input XML tokens.

The *sjoin* operator is an *automata operator*, which has two parts: the operator stem and the associated automaton. For an sjoin operator, the associated automaton

is constructed to recognize the pattern on which the join condition holds, e.g., "/r/a" in our example.

The JIT sjoin algorithm is straightforward. When an sjoin operator is invoked by its associated NFA (on the close match), it makes a cross product out of all its inputs seen thus far. The cross product is guaranteed to be the correct output. The trick is the timing of invocation. Take the document in Figure 3.2 and the plan in Figure 3.6 as example. The sjoin operator is first invoked on the first "", i.e., on the first close match event generated by the associated automaton. At this time, the output of the left Extract is $\langle[b_1, b_2]\rangle$, and the output of the right Extract is $\langle[\ c_1, c_2, c_3\ ]\rangle$. It is obvious that b_1, b_2, c_1, c_2, and c_3 are descendants of the first "a" element (see the id attributes in Figure 3.2). Hence the cross product $\langle[b_1, b_2], [\ c_1, c_2, c_3\ ]\rangle$ is the correct output. Similarly, every consequent invocation of the sjoin operator must have descendants of the current "a" element as input, because descendants of the previous "a" element would have been consumed by the previous invocation of the sjoin and descendants of future "a" elements have not yet arrived. Thus no value comparison is necessary in this operation. The complexity of the JIT join is equal to the complexity of the output tuple construction, i.e., linear in the output size.

The Navigate Operator

We have shown the extract operator and the sjoin operator that can be used to realize the bottom-up synthesis approach to a binding tree. We now introduce the *navigate* operator that builds up the binding tree in a top-down manner. A navigate operator takes as input a stream of elements from a depending binding, i.e., a binding on which other bindings depend. The output is comprised of descendent elements that are identified and retrieved from the input elements (formal definition in Section 4.2). For example, the middle navigate operator in Figure 3.7 takes as input a stream of "a" elements and outputs a stream of "b" elements.

Figure 3.7: A top-down approach. The query plan is executed bottom-up. The bottom-most operator evaluates the root node of the binding tree, while the two consecutive navigate operators evaluate the two leaf nodes

The query plan in Figure 3.7 represents the logic of the top-down approach to the query in Figure 3.1 (b). From an optimization point of view, this query plan differs from the bottom-up counterpart in that the two dependent bindings are evaluated sequentially. This allows for an optimization when a select operator is inserted between the two navigate operators (see Figure 3.8). The top-most navigate operator needs evaluate only the tuples satisfying the selection predicate. If in average every 1 tuple out of 10 satisfies the predicate, the other 90 percent of work is saved (see Chapters 5 and 7 for theoretical and experimental analysis).

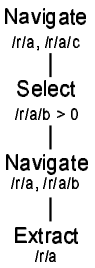

Figure 3.8: An optimization exploiting the top-down approach

Chapter 4

The Raindrop Framework

4.1 Overview

Chapter 3 has explained the rationale of the Raindrop framework. We now present the framework as a whole in detail.

While a number of recent papers [AF00, DFFT02, IHW02, MGOS03, LMP02] have shown that automata theory is suitable for XML stream processing, we now analyze the limitations of automata in terms of query optimization. Automata such as NFA, DFA, and transducer models enforce data-driven execution, which implies an underlying token-based data model. This token-based data model is different from the one adopted in the XQuery language, which instead is a sequence of node-labeled trees [W3C02b] or a collection of tree fragments [JAKL$^+$02]. From the query optimization point of view, mixing these heterogeneous models may complicate the system design and the optimization process. This is because either every operator is required to handle mixed-typed objects or policies must be imposed to ensure a specific type of object can only go to specific operators.

Considering the need for both data models and the limitations of arbitrarily mixing them, we now propose a unified framework to resolve this dilemma. We call

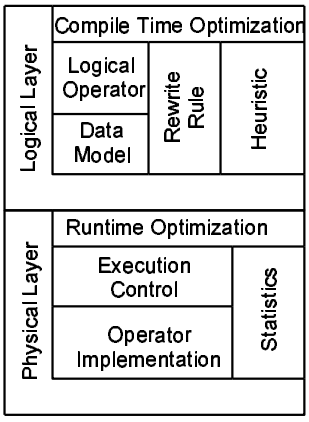

Figure 4.1: Two layers in the unified query model

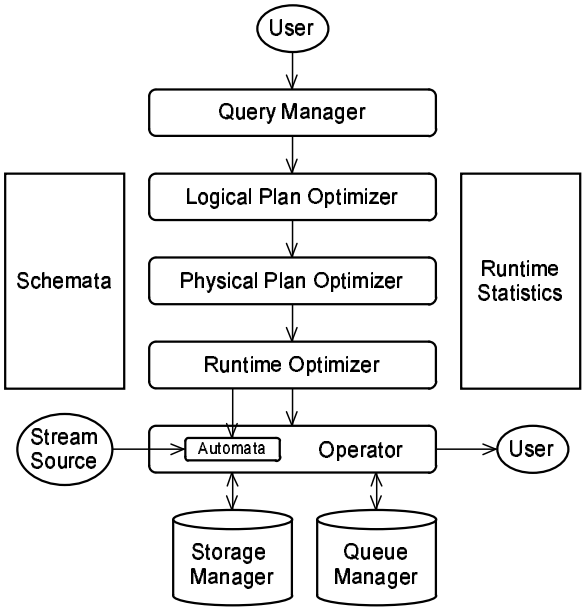

Figure 4.2: System components

it unified because any query functionality, no matter whether it is implemented using automata techniques or not, is uniformly modeled at the *logical layer*. This layer represents the semantics of query plans and serves specifically as the basis for query optimization. We adopt only the tuple model in this layer to simplify the design of operators and particularly to leverage various established query optimization techniques developed specifically for the tuple model [A+76, Cha98]. Note that while the token model is hidden from this layer, the automata-based flavor of the system is still implicitly embodied in the *automata operators* such as extract and sjoin. Hence, we can refine the automata in the same manner as we refine algebraic query plans, thus overcoming the limitations imposed by automata techniques (see Section 4.2 and Chapter 5).

The differences hidden from the logical layer are later resolved in the *physical layer*. This physical layer refines the logical query plan with detailed algorithms to implement the functionalities of query operators. In particular, the physical layer

describes how automata techniques are used to implement the *automata operators* such as extract and sjoin. The token-based data model, which is hidden in the logical layer, is now made explicit inside each *automata operator*. Note that this is a hierarchical design where the logical layer describes the overall query plan and the functionality of each operator, while the physical plan describes the internal implementation of each individual operator. Although the token model is made explicit in the physical layer, it is restricted to being exposed only inside each individual operator and does not cross the boundary between operators. This thus does not impair the homogeneity of the data model at the logical layer.

The physical layer also describes the overall execution strategy for a query plan. It specifies the control flow and the data flow between the physical operators. From the perspective of control, we devise a two-level control mechanism that integrates the data-driven execution strategy common for automata with more flexible execution strategies such as the traditional iterator-based evaluation strategy [Gra93] or the more recently proposed scheduler-driven strategies [M$^+$03]. We employ the concept of *mega operator* to bridge these two execution levels (see Section 4.3.2).

From the perspective of data flow, we use a First-In-First-Out mechanism, or *queues*, to direct intermediate data between physical operators. The basic data unit in the queues is the *tuple*. A tuple may contain both the *value-based* and the *reference-based* data. We adopt a *storage manager* for centralized storage for the referenced data. The purpose of introducing the reference-based data is to support sharing data between tuples (see Section 4.3).

4.2 The Logical Layer

4.2.1 The Data Model

Because [W3C02b] has defined the *XQuery data model* for query evaluation, we now first explain why we in addition need yet another data model. According to [W3C02b], a data model defines the logical view of (1) the source data and (2) the intermediate data of query expressions (i.e., the inputs and outputs of query operators). In the *XQuery data model*, source data is defined as node-labeled trees augmented with node identity, while intermediate data is defined as a sequence of zero or more items, each either a node or an atomic value.

We cannot directly adopt the *XQuery data model* to query streaming XML data. First, streaming XML data can be more naturally viewed as a sequence of discrete tokens, where a token can be an open tag, a close tag, or a PCDATA. In fact, the node-labeled tree view of an XML data stream is incomplete until after the stream is wholly received and parsed.

Second, the *XQuery Data Model* is not suitable for pipelining the execution. To make it clear, we shall draw a comparison with the *relational data model* [Cod70], which is based on sets of tuples. A relational algebraic query plan usually ignores whether it will be executed in a pipelined fashion or in an iterative fashion. In other words, the execution strategy is left out off the logical query model. This design is feasible, however, only because the *relational data model* has a natural atomic execution unit, i.e., a tuple. A query plan executor can choose whatever execution strategy without breaking this atomic unit (i.e., context switch when a tuple is only partially processed). In the XML realm, however, such an atomic unit is not directly available because of its arbitrarily nested structure. In fact, many approaches such as the one suggested by the *XQuery data model* [W3C02b] consider the complete

XML document as one unit, and thus exclude the possibility of pipelined execution. Going to another extreme, [LMP02, GS03, PC03] consider each XML data token as an atomic unit. As discussed in previous sections, this purely token-based approach is rigid and at times too low-level, compared with the set-oriented relational tuple model [Cod70].

Based on the above observations, we now define our logical data model.

1) We define the source XML streams to be sequences of *tokens*, where a *token* can be an open tag, a close tag, or a PCDATA. Formally, we define the domain of *tokens* $\mathcal{T}$ as:

$$\mathcal{T} = \{< x > | x \in \mathcal{E}\} \cup \{< /x > | x \in \mathcal{E}\} \cup \{d | d \in \mathcal{D}\}$$

where $\mathcal{E}$ is the domain of *XML element names* and $\mathcal{D}$ is the domain of *character data* (strings).

2) We define intermediate data of query expressions (i.e., the inputs and outputs of query operators) to be a sequence of zero or more *tuples*, with each *field* in a *tuple* being a sequence of zero or more *items*. Each *item* is either an *XML element* or an *atomic value*. Formally, we define the domain of *tuples* $\mathcal{P}$ as:

$$\begin{aligned} \mathcal{F} &= \{\ [v_1, ..., v_n] \mid v_i \in \mathcal{A} \cup \mathcal{X}, n \ is \ the \ size \ of \ a \ field\}, and \\ \mathcal{P} &= \{\langle f_1, ..., f_n \rangle | f_i \in \mathcal{F}, n \ is \ the \ arity \ of \ a \ tuple\} \end{aligned}$$

where $\mathcal{F}$ is the domain of *fields*, $\mathcal{A}$ is the domain of *atomic values*, and $\mathcal{X}$ is the domain of *XML elements*.

4.2.2 The Logical Operators

Following the data model defined above, every logical operator (except for the extract operator that is always a leaf node in a query plan and takes no input) accepts a sequence of tuples as input and produces a sequence of tuples as output. An

overview of the core logical operators in the Raindrop framework is given in Table 4.1.

Name	**Symbol**	**Description**
Selection	σ_{pred}	Filter tuples based on the predicate *pred*
Projection	π_v	Filter columns based on the variable list v
Join	$\bowtie_{pred}$	Join input tuples based on the predicate *pred*
Aggregate	Δ_f	Aggregate over input tuples with the aggregate function *f*, e.g., average
Tagger	T_{pt}	Format outputs based on the pattern *pt*
Navigate	Φ_{p_1,p_2}, ϕ_{p_1,p_2}	Take input elements of path p_1 and output descendants of the input elements following the path p_2. There are two types of navigate operators that differ only in how to handle collections (see following description).
Extract	Ψ_p, ψ_p	Retrieve elements specified by the path p from the input stream. There are also two types of extract operators the differ only in how to handle collections (see following description).
SJoin	$\bowtie_p$	Join input tuples on their structural relationships, e.g., having a common ancestor of the path p

Table 4.1: A List of Logical Operators

In fact, the first five operators in Table 4.1, including Selection, Projection, Join, Aggregate, and Tagger, are standard operators that can be found in most query engines [Cod70, CFI+00, JAKL+02, IHW02, ZPR02]. We refer readers to [ZPR02] for detail discussion. Here we only focus on the last three operators, i.e., Navigate, Extract, and SJoin, which constitute part of this thesis's contributions.

We still follow the running example of Chapter 3. But for ease of reference, we

also list the input document, the example query, and the query plan in Figures 4.3, 4.4, and 4.5, respectively.

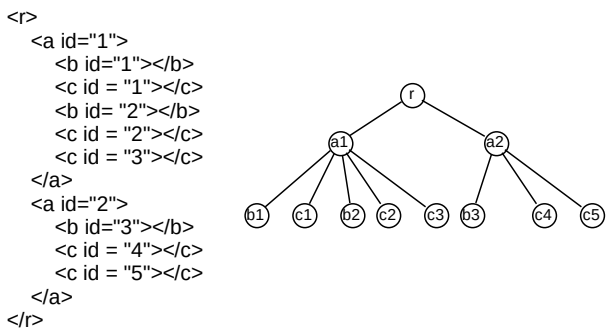

Figure 4.3: Input XML document

```
FOR $a in doc("foo.xml")/r/a
LET $b := $a/b, $c := $a/c
WHERE $b > 0
RETURN $c
```

Figure 4.4: Example user queries

NavigateUnnest

The NavigateUnnest operator evaluates a relative path (the second argument p_2) over the initial context node-set specified by an absolute path (the first argument p_1). An output tuple is constructed from each resulting element by inserting the descendant as a field into the input tuple. For example, $\Phi_{/r/a,b}(\ \langle a1 \rangle\ ) = [\ \langle\ a_1, b_1\ \rangle, \langle\ a_1, b_2\ \rangle\ ]$, where "a1" is the first "a" element in Figure 4.3, and "b1" and "b2" are the two "b" descendants of "a1".

Formal definition is given by:

$$\Phi_{p_1,p_2}(\ T\ ) = [\ \langle t \circ\ f \rangle \mid t \leftarrow T, f \leftarrow follow(\pi_{p_1}(t), p_2)\ ]$$

The *follow* operation denotes the evaluation of the relative path p_2 over the initial context nodes-set specified by the absolute path p_1. We also extend the notation

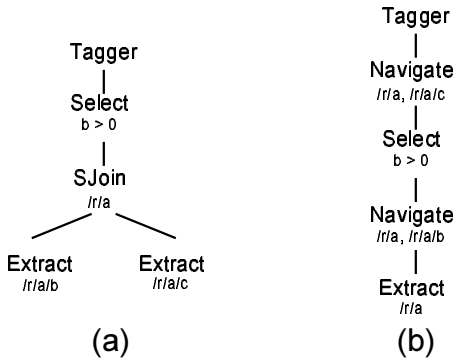

Figure 4.5: Two equivalent query plans for the query in Figure 4.4

of the projection operation π, where $\pi_p(t)$ denotes the XML elements from tuple t that conform to the path p.

NavigateCollection

The NavigateCollection operator evaluates a relative path (the second argument p_2) over the initial context node-set specified by an absolute path (the first argument p_1). The only difference between NavigateCollection and NavigateUnnest is that, for all resulting elements, NavigateCollection constructs one collection from all such elements. This collection is then concatenated to the input tuple as one field. Following the above example, $\phi_{/r/a,b}(\ \langle a1 \rangle\ ) = [\ \langle\ a1, [\ b_1, b_2\ ]\ \rangle\ ]$.

The formal definition is given by:

$$\phi_{p_1,p_2}(\ T\ ) = [\langle t \circ\ \ f \rangle \mid t \leftarrow T, f = [\ f' \mid f' \leftarrow follow(\pi_{p_1}(t), p_2)\ ]\ ]$$

ExtractUnnest

The ExtractUnnest operator finds from the input data stream all elements that conform to an absolute path (the argument p). A new tuple is generated for each such element. For example, giving the document in Figure 4.3 as the input stream,

$\Psi_{/r/a/b}(\ ) = [\ \langle b_1 \rangle, \langle b_2 \rangle, \langle b_3 \rangle\ ]$, where b_1, b_2, and b_3 are the "b" elements conforming to "/r/a/b".

The formal definition is:

$$\Psi_p(\ ) = [\ \langle n \rangle \mid n \leftarrow follow(p)]$$

We extend the notation of the *follow* operation, where $follow(p)$ denotes all elements from a given data stream that conform to the absolute path p. An *ExtractUnnest* operator does not take in any input tuples. Instead, as will be further explained in the physical layer, the operator's internal mechanism, namely its associated automaton, is responsible for analyzing the input stream and extracting the desired data.

ExtractCollection

The ExtractUnnest operator finds from the input data stream all elements that conform to an absolute path (the argument p). The only difference between ExtractCollection and ExtractUnnest is that, for all elements matched from the input, ExtractCollection constructs one collection from all such elements. This collection is then used as the only field for the output tuple. Following the above example, $\Psi_{/r/a/b}(\ ) = [\ \langle\ [\ b_1, b_2, b_3]\ \rangle\ ]$.

The formal definition is:

$$\psi_p(\ ) = [\ \langle n \rangle \mid n = \ [\ n' \mid n' \leftarrow follow(p)]\ ]$$

SJoin

The Structural Join operator concatenates input tuples based on their structural relationship such as having a common ancestor. For example, $[\ \langle\ [\ b_1, b_2, b_3]\ \rangle\ ] \bowtie_{/r/a}$ $[\ \langle\ [\ c_1, c_2, c_3, c_4, c_5]\ \rangle\ ]\ = [\ \langle\ [\ b_1, b_2\ ], [c_1, c_2, c_3]\ \rangle, \langle\ [b_3], [c_4, c_5]\ \rangle\ ]$

The formal definition is:

$$T_x \ \bowtie_p \ T_y = [\langle t_x \circ t_y \rangle \mid t_x \leftarrow T_x; t_y \leftarrow T_y; precede(t_x, path) = precede(t_y, path)]$$

The *precede* operation denotes the "ancestor" relation in XPath [W3C02a]. Although a *Structural Join* operator can be implemented like a normal join, i.e., by value comparisons, it can be more efficiently implemented using an associated automaton (see Chapter 3).

4.3 The Physical Layer

4.3.1 Implementing Data Queues

Recall from the logical layer that the input and output of query operators is defined as a sequence of tuples. A data queue is used to pass the sequence between different query operators. A data queue also serves as a buffer between different query operators, which allows for set-at-a-time operations.

Let us look more closely at the data inside the queue. A tuple may contain both reference-based and value-based data. We have mentioned that introducing reference-based data is for sharing of data. We now explain the reason.

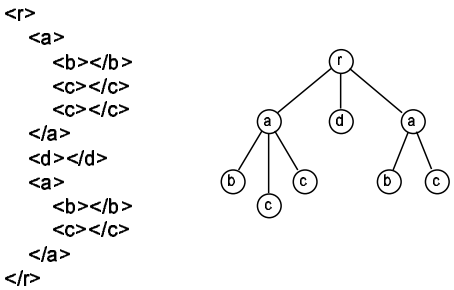

Figure 4.6: An example XML document tree

Recall also from the logical layer that a tuple contains a sequence of items, which

can be an XML element. Consider a tuple $\langle \$r, \$a, \$b \rangle$ that stores an "r" element, an "a" element, and a "b" element. If value-based data is used, this tuple will look like Figure 4.7 (a). Note that the "b" element, i.e., " ", is stored repeatedly in the tree fields of the tuple. This is obviously a waste of space and potentially a waste of CPU cycles because the contents need maintenance. As a better solution, we adopt reference-based data, as shown in Figure 4.7 (b). Now only the *least common ancestor*, i.e., the "r" node, is stored centralized in the storage manager, and tuples store only references pointing to respective elements.

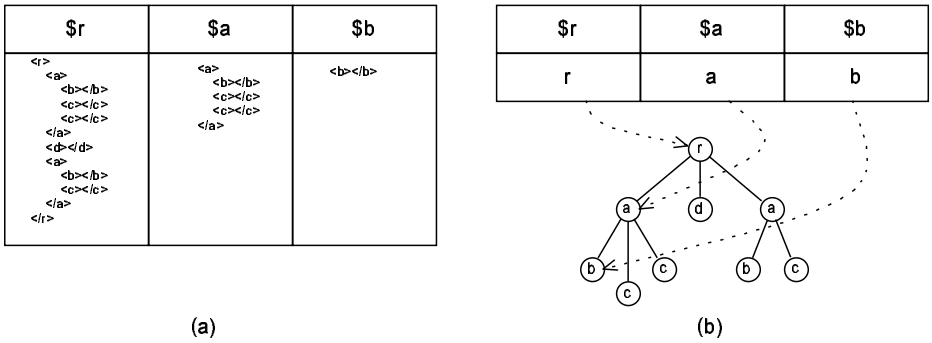

Figure 4.7: Value-based tuple vs. reference-based tuple

4.3.2 The Control Flow

A general assumption in stream systems is that data arrival is unpredictable [M+03]. This characteristic makes iterator-based execution strategies [Gra93] not directly applicable in the stream context, because an operator may block the entire execution thread when the required input stream has not yet been received. One solution is to let the incoming data "trigger" the execution, which leads to a purely data-driven (or event-driven) execution strategy. This approach is adopted by for example [LMP02, GS03, PC03]. The disadvantage is its rigidity: every incoming data token will immediately trigger a fixed sequence of operations. This rigidity excludes, for

example, the possibility of deferring certain operations and then batching their processing, which may be more efficient because the cost of context switching between operators can be reduced. It also excludes the possibility that, by deferring certain operations, these operations may become unnecessary due to dependencies between operations (see the example in Section 3.2.3 and more details in Chapter 5).

Another solution, as adopted by [M+03], uses a *global scheduler* that invokes the *run* methods of different query operators from the given query plan based on a variety of scheduling strategies. In this approach, a *scheduling cycle* consists of two steps: (1) making a scheduling decision, i.e., decide which operator to run and how long it should run, and (2) invoke the chosen operator for the decided amount of time. The disadvantage is its over generality. In principle the scheduler-driven strategy subsumes the data-driven strategy, i.e., the general strategy can simulate the rigid one. It is easy to conceive, however, that this simulation may be less efficient than directly applying the data-driven strategy because of the cost in making such scheduling decisions.

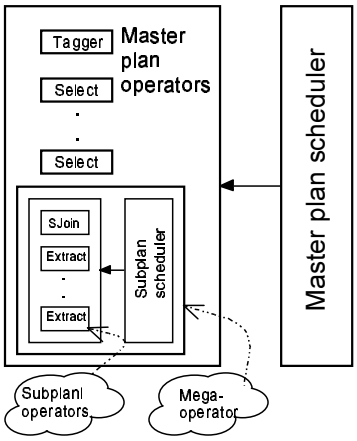

Figure 4.8: Two-level scheduling

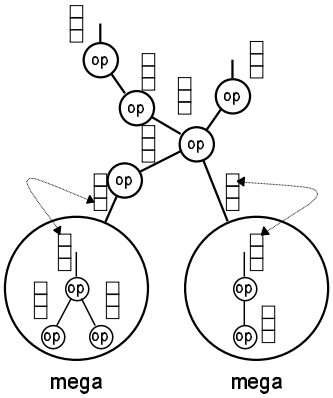

Figure 4.9: Data transfer

The Raindrop framework adopts an integrated approach to take advantages from both the above two control strategies. This is achieved by creating a *scheduler's*

perspective over a query plan. From the scheduler's perspective, a query plan is organized into two levels. The top level, the *master plan*, preserves the overall structure of the query plan, except that certain operators, namely, the automata operators, are grouped together and replaced by a so-called *mega operator*. The replaced operators are abstracted as one computational subtask, namely, a *subplan* that composes the bottom level of the plan structure. The master plan and its subplans are connected via the mega operator, that is, each subplan is represented by a mega operator at the master plan. This mega operator is then responsible to transfer data between the two plans.

This mechanism allows different scheduling strategies to be applied to the master plan and to the subplans. We adopt the scheduler-driven approach at the master plan, i.e., a global scheduler decides which operator at the master plan to run at any given time and invoke the operator to run a certain time. The subplan is driven by an associated automaton, which reacts to the immediate availability of input data.

Chapter 5

Compile Time Optimization

5.1 Overview

The task of compile-time optimization, or simply optimization, is to improve the query plan generated by the query parser and to provide a quality input to the query executor. Because the Raindrop query engine has both the flavor of automata and query algebra, we shall first overview the optimization techniques applied in both paradigms individually. Thereafter we discuss optimization techniques that are applicable when both paradigms are considered as a whole.

From the automata point of view, optimization strategies for pattern retrieval mainly focus on three aspects. First, the schema information, e.g., DTD and XML Schema, can be exploited to rewrite certain patterns. For example, given an XPath expression "//r/*/b", the schema may imply that the "r" element can only be a child of the "root" element, and the "r" element can only have one child element "a". Exploiting these schema constraints allows us to rewrite the given expression into "/root/r/a/b". Because evaluating a normal child-axis node test is generally less expensive than evaluating a descendant-axis or a wildcard test, the latter expression is preferred over the former one.

Second, non-deterministic automata can generally be mapped to deterministic-equivalents. This technique has been favored by some previous work [MGOS03], but it comes at the cost of potentially increasing the size of the automata exponentially in the number of states. It also loses the nice resemblance between the diagrams of non-determinate automata and binding trees. This resemblance makes it much easier to modify the automata at runtime accounting for changes to the binding tree. This kind of runtime adaptivity is discussed in Chapter 6.

Third, automata for different patterns can be merged into one automaton. In fact, this ability to share computation has been one of the major driving forces for automata to be applied in XML filtering where a very large number (e.g., thousands or tens of thousands) of user queries may potentially occur.

From a traditional DBMS point of view, query rewriting at the algebraic level and choosing the proper physical operator implementations have been two major approaches to optimization. Query rewriting mainly focuses on reducing the intermediate data size between query operators. Rewriting rules such as selection pushdown, projection pushdown, and join order rearrangement have been applied virtually in all traditional DBMS [Cha98]. Choosing proper physical operator implementations, on the other hand, mainly focuses on reducing the cost, especially the number of disk page accesses, inside each individual operator.

For an XML stream processing system that adopts both the automata and the query algebra, all above optimization techniques are potentially applicable. In fact, the unified framework introduced in Section 4 aims for exactly this purpose. In addition, certain not-yet exploited optimization techniques can be identified when the automata paradigm and the query algebra paradigm are considered as a whole. We shall focus on one of such techniques, i.e., to flexibly decide which query functionality to be implemented in which paradigm. In fact, our unified framework has made this optimization not only feasible but also effortless. For example, the top-

down analytic and the bottom-up synthetic approaches of evaluating binding trees (see Chapter 3) have different flavors in deciding how to implement certain query functionality. We shall discuss this in detail in the following sections.

5.2 Rewriting Rules

We refer readers to [Cha98, AHV95] for general rewriting rules. Here we only focus on novel rewriting rules that can change the query functionality of the automata, i.e., the way in which the automata are integrated into the overall query plan. We first discuss these rules one by one, then we use an example to illustrate their actual applications.

- Navigation Pushin

  $$\phi_{p_1,p_2}(\bowtie_{p_1}) = \bowtie_{p_1} (\psi_{p_2})$$

  where p_1 and p_2 are XPath expressions and p_1 subsumes p_2, i.e., p_1 is the prefix of p_2.

  From the left hand side to the right hand side, a pattern evaluation that is originally implemented by a navigate operator ϕ is changed to be implemented by an extract operator ψ. This is to say, a pattern evaluation is moved from the top-down analytic approach to the bottom-up synthetic approach. This rule holds if the entry point (i.e., p_1) of the navigate operator is equal to the condition of the sjoin operator (i.e., both are p_1), and p_1 is a prefix of p_2 (e.g., $p_1 = /a$ and $p_2 = /a/b$).

- Redundant SJoin

  $$\bowtie_p (\psi_p) = \psi_p$$

  The sjoin operator $\bowtie$ is redundant because the pattern p evaluated by the extract operator is to be structurally joined on the same path p. This is equal

to a group-by operation over all attributes in a relational algebra, such that the group-by would also be redundant.

- Redundant Extract

$$\bowtie\!\!\triangleleft_{p_1} (\psi_{p_1}) = \bowtie\!\!\triangleleft_{p_1}$$

This rule removes the binding p_1, which must be a supporting binding , and therefor is redundant in the output tuple stream (see Chapter 3).

- Navigation Pushdown

$$\phi_{p_1,p_2}(op) = op(\phi_{p_1,p_2})$$

where *op* denotes an arbitrary operator that does not access the p_2 binding, i.e., they have no interdependency.

From left to right, this rule pushes down a navigate operator so that it is evaluated before *op*.

- Selection Pushdown

$$\sigma(op) = op(\sigma)$$

where op denotes an arbitrary operator that does not modify any binding in the select condition.

This is the general selection pushdown rule, which is applied in virtually every DBMS. We list it here only as our example will refer to it.

We now formally (i.e., using query rewritings) present an example that switches the top-down analytic and the bottom-up synthetic approaches.

$$\bowtie_{/a} (\psi_{/a/b}, \psi_{/a/c})$$

$$
\begin{aligned}
&= \bowtie_{/a} (\psi_{/a}, \psi_{/a/b}, \psi_{/a/c}) \\
&= \phi_{/a,/a/c}(\bowtie_{/a} (\psi_{/a}, \psi_{/a/b})) \\
&= \phi_{/a,/a/c}(\phi_{/a,/a/b}(\bowtie_{/a} (\psi_{/a}))) \\
&= \phi_{/a,/a/b}(\phi_{/a,/a/c}(\psi_{/a})))
\end{aligned}
$$

The first expression applies the bottom-up synthetic approach. It first evaluates the "/a/b" and the "/a/c" bindings, and then it structurally joins them on "/a". The last expression applies the top-down analytic approach. The inner extract operator evaluates the "/a" binding, which is the root in the binding tree. Afterwards, the two navigate operators evaluate the leaf bindings.

If a selection on "/a/c" is added to the query plan, the following rewriting takes advantage of the imposed dependency (see Chapter 3) in the top-down analytic approach and evaluates selection early. This rewriting is visualized in Figure 5.1.

$$
\begin{aligned}
& \sigma_{/a/b>0}(\bowtie_{/a} (\psi_{/a/b}, \psi_{/a/c})) \\
= \; & \sigma_{/a/b>0}(\phi_{/a,/a/b}(\phi_{/a,/a/c}(\psi_{/a}))) \\
= \; & \sigma_{/a/b>0}(\phi_{/a,/a/c}(\phi_{/a,/a/b}(\psi_{/a}))) \\
= \; & \phi_{/a,/a/c}(\sigma_{/a/b>0}(\phi_{/a,/a/b}(\psi_{/a})))
\end{aligned}
$$

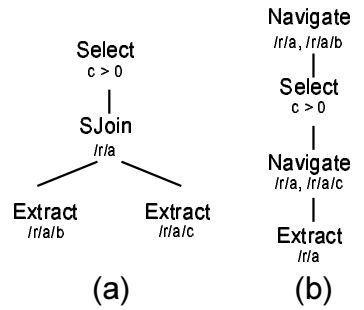

Figure 5.1: Top-down vs. Bottom-up

5.3 Cost Estimation

We now analyze the cost of several operations that are relevant to optimization. It should be noted that the cost estimation in our assumed XML stream processing context is very different from the one in traditional database. Traditional DBMS is IO-bound, i.e., the overwhelming cost factor is disk I/O. The number of page accesses is used as the cost criterion. In the XML stream processing context, however, most operation is assumed to be executed in main memory, i.e., the number of CPU cycles now becomes the cost criterion. This is not to say that an XSP system involves no I/O. On the contrary, the intensive interaction with other systems implies intensive network I/O. It is only because the network I/O is assumed to be unpredictable and uncontrollable, i.e., the system passively waits for data rather than actively issues I/O accesses to acquire data, that we leave it outside the cost model.

Automata

The execution of an automaton is basically a loop of:

1. take an input token,
2. match the input token to identify next transitions,
3. update the current states.

The major cost occurs in the string comparisons from the second step that decides the next transition [Wat97]. In automata implementations that use a hash table to match transitions, the number of outgoing transitions from the current state is irrelevant, because the cost of search in a hash table is basically a constant independent from the number of objects that the table contains, at least when the number is not extremely large. Hence, the total cost of running an automaton on an input stream depends only on the number of tokens that need to be matched in

the transition hash table. Surprisingly, this number does not equal the number of tokens in the entire stream. The following example illustrates the reason.

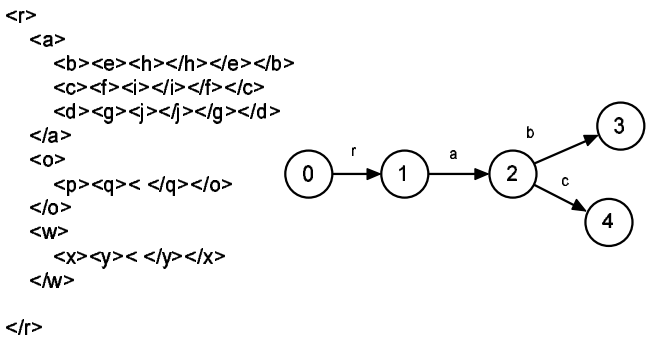

Figure 5.2: A NFA showing necessary string comparisons

Consider the NFA and the document in Figure 5.2. It is clear that the open tags such as <r> and <a> cause one search each in the transition hash table. But once the XPath "/r/a/b" has been matched, the following tags starting from <e> are guaranteed to be children of the found "b" element. What we need is to find the paired . Assuming well-formed inputs, we can achieve this by counting the depth of tokens using a counter. We set the counter to 1 once the is reached. Each following open tag such as the <e> and <h> increases the counter by 1; each following close tag such as the </h> and </e> decreases the counter by 1. When the counter reaches 0, the corresponding will be read in. This simple counting mechanism can also be applied when <d>, <o>, or <w> are read in and found to be mismatches. Since no string comparison occurs after a match or a mismatch is found, the cost can be ignored.

The number of actual comparisons can be shown clearly when the input tokens are represented in a document tree (Figure 5.3). Note that the siblings of matched elements such as "o", the "w" also incur a string comparison, because they are potential matches. We can generalize this as follows. For a node test of the child

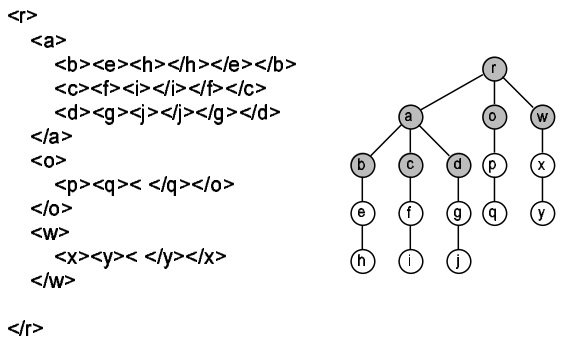

Figure 5.3: A tree representation, the gray nodes incur string comparisons

axis, the number of comparisons equals the number of siblings of the node, or in other words, the number of children of the context node.

In fact, the above analysis depends on the content of the input stream. But as we shall find out shortly, this dependency is identical to that of another operator, the *navigate*. With certain approximation, we can cancel out the dependence, and thus provides useful information in reaching an optimization decision.

Extract

Recall that an *extract* operator consist of two parts: the stem and the associated automaton. As analyzed above, the major cost of the associated automaton, i.e., the number of comparisons, depends on the input stream and the given XPath query. We shall denote this number as N_c. We denote the cost of each comparison as C_c. The cost of the stem is basically a constant for making a new tuple, we shall denote it as C_t. Hence, the cost of an *extract* operator is $C_e = C_c * N_c + C_t$.

Structural Join

Similar to *extract*, the cost of an *sjoin* operator incurs two parts: the stem cost and the automaton cost. Estimating the automaton cost can be done similar to the

extract operator. The cost of the stem is a little different. One single invocation of an *sjoin* is basically like a cross product. Hence the major part of it corresponds to constructing the output tuples, whose number is linear to the size of output. For an *sjoin* that has all source operators as *ExtractCollection*, the cross product degrades to a concatenation, because the output size of the *ExtractNest* is always 1. In this case, the cost of constructing the output tuple is close to a constant. In fact, this case is very common, as collection operations are default in XQuery [W3C02b].

Navigation

Consider the middle navigate operator in Figure 5.4, which matches "b" elements from the output of the bottom-most extract operator. The operation of a navigate operator is basically to navigate through the document tree. For each input "a" element, the operator needs to match all its child elements. Hence the "b", "c", and "d" elements in Figure 5.4 are marked grey, which is identical to Figure 5.3.

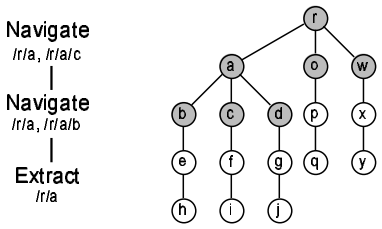

Figure 5.4: Comparisons incurred in a navigate operator

Putting the cost estimation together

Consider the "pushin" plan in Figure 5.5 (a) where all navigation functionalities are pushed into the automaton, and the "pullout" plan in Figure 5.5 (b) with only the minimal navigation functionality pushed into the automaton.

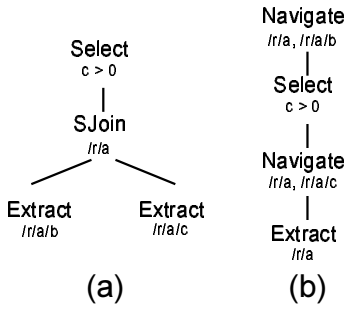

Figure 5.5: The "pushin" plan and the "pullout" plan

Let n_a, n_b, and n_c denote the number of "a", "b", and "c" elements in the input stream, respectively. Let C_{et} denote the cost of the stem in an extract, C_{jt} denote the cost of the stem in an sjoin, C_{nc} denote the cost of each comparison in a navigate, C_{nt} denote the cost of constructing output tuples in a navigate, C_{ac} denote the cost of each comparison in the automaton, C_s denote the cost of evaluating one tuple in a select, and σ denote the selectivity of the select operator. The cost of the pushin plan would be $C_{pushin} = (n_b+n_c)C_{et}+n_aC_{jt}+(n_a+n_b+n_c)C_{ac}+C_sn_a$, and the cost of the pullout plan would be $C_{pullout} = n_aC_{et}+(n_b+n_c)(C_{nc}+C_{nt})(1+\sigma)+C_{ac}n_a+C_sn_a$. Because C_{pushin} is independent of σ while $C_{pullout}$ is a linear function of σ, we can set them to equal and try to find out the critical value σ' such that if $\sigma > \sigma'$, $C_{pushin} < C_{pullout}$, and vice versa.

$$\begin{aligned}
\sigma' &= \frac{C_{ct}(n_b+n_c-n_a)+C_{jt}n_a+C_{ac}(n_b+n_c)}{(n_b+n_c)(C_{nc}+C_{nt})}-1 \\
&\approx \frac{C_{et}(k-1)+C_{jt}+C_{ac}k}{k(C_{nc}+C_{nt})}-1 \\
&= \frac{C_{et}+C_{ac}+\frac{C_{jt}-C_{et}}{k}}{C_{nc}+C_{nt}}-1 \\
&\approx \frac{C_{et}-C_{ac}}{C_{nc}+C_{nt}}-1
\end{aligned}$$

The first approximation occurs when we assume $kn_a \approx n_b+n_c$, which means the

the number of children of the "a" elements is about constant. The second occurs when we assume $C_{jt} \approx C_{et}$, which means the cost of constructing one tuple in the extract operator and in the sjoin operator is about the same.

The point we try to make here is that the final form of σ' is approximately a constant. Therefore, we can establish the heuristic to choose the pullout plan when $\sigma < \sigma'$, and to choose the pushin plan otherwise. This heuristic can guide the query optimizer, as the task of a query optimizer is indeed to choose the optimal plan.

Chapter 6

Runtime Optimization

6.1 Overview

Traditionally, query optimization is applied only at compile time, i.e., off-line. Although the resulting query execution plan may be suboptimal in actual situations, it is adopted for the entire execution without further refinement. This may be desirable for short-running queries, for which the cost of re-optimization at runtime may well exceed the potential benefit because the total cost that optimization could be reduced is relatively small. Also, because traditional DBMSs assume that data is stored locally on disk, data statistics can be readily obtained. This means a query plan, optimized based on the acquired data statistics, may likely be near-optimal.

In the stream processing context, however, data sources are remotely located and may be beyond the control of the query engine. Hence data statistics may not be obtainable beforehand. This usually results in sub-optimal query plans. In addition, stream query executions generally span longer time. Hence it may now be more profitable to optimize a suboptimal plan at runtime. The long-running queries also cause additional problems, e.g., the query evaluation environment may change during execution. These dynamics come in different flavors. First, system

resources available to a specific query may fluctuate due to other queries running in parallel. Second, user requirements (i.e., the user query and user preferences on system performance versus cost) may change before an execution is complete. Third, data statistics may change over time. A formerly optimal plan may become sub-optimal due to such changes. All these kinds of uncertainty and dynamics call for adaptation of the query system to the changing environments, that is, runtime optimizations, in stream processing. While our framework can potentially deal with all the above dynamics, this thesis focus only on the third one.

Runtime optimization involves two steps. The first step is concerned with finding an alternative, preferably optimal, query plan. This is also the same concern of compile time optimization. The key difference here is that we can not assume *a prior* data statistics in runtime optimization. Instead we need to collect them on the fly. The second step involves plan migration. Because a query plan at runtime cannot be modified blindly, special concerns must be taken to ensure the consistency of the query plan throughout the migration process and the correctness of the finally generated query plan.

In terms of plan migration, the use of automata poses an additional challenge. Automata maintain certain states that are independent from the global data structures (i.e., the overall query plan). Hence special concerns must be taken to ensure that modifying a query plan will not impair the consistency of the states of the automaton and that any knowledge concluded from the automaton before being changed is properly refreshed. This generally involves synchronization between the algebraic query plan and the underlining automaton.

6.2 Runtime Optimization Decision Making

By decision making we refer to finding an better query plan. If all meta-information about a query, i.e., available system resources, user preferences, and data statistics, is accessible, the decision making at runtime would be identical to that at compile time (see Chapter 5). However, the meta-information is generally not known beforehand. Hence here we shall focus on obtaining this information, which can be achieved by collecting them at runtime.

The statistic collection is not quite straightforward as it might seem. For example, the principal type of data statistics concerned in this paper is the selectivity of query operators. Selectivity in terms of an operator stands for the ratio of actual output tuples over potential output tuples. In case of a *select* operator, the selectivity is equal to the percentage of the input tuples that satisfy the predicate of the operator. Although the selectivity may be a constant over the entire input, this constant is not available until the entire stream is read in. Also, since runtime optimization may occur multiple times during the entire execution, what really matters is not the selectivity over the entire stream, but the selectivity over the period between the current optimization stage and the next. Interestingly, the accurate value of this selectivity is only available when the next optimization stage starts, at which moment the acquired value is not useful anymore. For this reason, we can not wait for this accurate value but can only make a best estimation based on inputs that have already been processed.

The estimation in the current implementation is based on a simple update algorithm:

$$S_c = S_n * \alpha + S_o * (1 - \alpha)$$

where S_o, S_n, and S_c are the old selectivity, the new selectivity update, and the current estimate of the selectivity that will be used for optimization. α is the

update ratio.

6.3 The Migration Problem

Given a query plan at runtime, a part of the plan to be replaced (the old subplan), and a new subplan to replace the old subplan, the *migration process* adapts the given query plan to a new plan by replacing the old subplan with the new subplan. A migration is *correct* if the migration does not change the content of the output; the timing issue regarding when the output is generated is irrelevant. This means the output generated from the new plan after migration should be identical to the output generated from the old plan supposing no migration were started. To ensure correctness, we generally need to eliminate loss of data, eliminate duplicates of data, and maintain correct order (in order-sensitive cases).

It is substantially more challenging to modify or migrate a query plan at runtime than at compile time. At the compile time, it suffices to simply remove operators from the old plan, insert new operators, and make proper connection, as one would manipulate an abstract graph. At runtime, however, one cannot blindly remove an operator because it may corrupt certain runtime information that leads to corruption of the system or incorrect output.

This runtime information consists of several parts. First, an operator may have internal states that are relevant to future output, as in a hash join operator. Second, if more than one operator is to be removed, the data passing between these operators may be in an inconsistent state. Hence these operators must be taken as a whole.

Some previous work [KD98] resolves this problem by assuming that a query plan is executed in stages. Modification is limited to a subplan in a stage where the execution has not started. In the stream processing context, however, we cannot make such an assumption because we prefer early output even before the entire input

stream is read in. Setting stages in a query plan would disallow output from one stage unless the entire input is processed in all previous stages. Therefore each stage is a "combined" blocking operator, which is not allowed in strict stream processing.

Besides correctness, there is another problem we need to concern during migration, i.e., the interruption of output during the migration. This is due to the fact that the old subplan may need a clean up process before it can be disconnected from the remaining subplan. This clean up processes may lead to certain interruption in the output, which is undesirable in the cases where smooth output is preferred.

6.4 Conditions for Migration

6.4.1 Conditions on the Scheduler

As mentioned in previous sections, we can not blindly remove or plug in a query operator at runtime. From the scheduler's point of view, one obvious reason is that we can not remove an operator when it is being invoked by the scheduler and thus in the process of running. Another reason is that the scheduler may have its own representation of the query plan; when the plan is modified, the scheduler must be informed or synchronized so that it can update its internal representation. Yet another subtle problem is that scheduling decisions are not made atomically, i.e., modifying the query plan at an arbitrary moment may leave the scheduler in an inconsistent state.

Based on the above concerns, we define the *window of migration* for the scheduler as the moments when migration can be started without corrupting the consistency of the scheduler. The window is open when a scheduling decision is made and the resulting operation is finished executing. In such moments, it is clear that (1) no operator is running, and (2) the next cycle of scheduling decision making has not

yet started, i.e., we can now safely modify the query plan and update the internal representation in the scheduler.

The above window of migration is not always open, though. Sometimes we need to wait for it. But the waiting time is guaranteed to be no more than one scheduling cycle, i.e., the period in which a scheduling decision is made and the scheduled operator is executed. For most scheduling strategies this waiting time is relatively short compared with other waiting times in the migration process (which will be discussed shortly) and thus is generally affordable.

6.4.2 Conditions on the Automaton

From the associated automaton's point of view, to remove or to insert an automata operator also means to remove or to insert the corresponding states in the automaton. We can not blindly do so, however, because the automaton maintains its internal states (e.g., the stack, see Chapter 3) that must be kept consistent during runtime. For example, if a state that has been pushed into the stack is deleted, the automaton would be confused when the state is popped. Also, an operator stem (see Chapter 3) may require its associated state being present during certain periods. For example, an extract operator stores data during open-close match periods (see Chapter 3). If the associated state is deleted between an open match and the corresponding close match, the extract operator would store incorrect data.

Similar to the scheduler synchronization problem, we can define the *window of migration* for an automaton. A state at runtime is *busy* if the state has been activated by a transition and remains in the stack of the automaton; it is *idle* otherwise. Then we say the window of removing a state at runtime is open if the state is idle. The window of inserting a state is open if, supposing it were inserted before the automaton is started running, it would be idle at the moment of insertion.

This mechanism ensures that no operation that are related to a state is in execution or would be in execution when the state is being modified.

Similar to the scheduler, this window of migration is not always open during runtime. However, the waiting time is not guaranteed but instead depends on the content of the input and the delivery of the input, which are assumed to be unpredictable. This means the waiting time is also unpredictable. The next section will introduce a "hook" mechanism that handles this unpredictable situation.

6.5 The Migration Process

The migration process starts when a runtime optimization decision is reached, i.e., both the part in a query plan to be replaced (the old subplan) and a new subplan are given. Because the scheduler and the automata are running asynchronously, we need to ensure that the conditions in the previous section are met at the same time. This is in fact a "who waits for whom" problem.

Because the waiting time for the window of the scheduler is guaranteed to be no more than one scheduling cycle, and because the waiting time for the window of the automata is unpredictable, we take the following steps:

1. wait for the window of migration at the automaton; this is done by installing a hook on the *critical state*, which is defined as the associated state for the root of the binding tree. When the critical state becomes idle, an event is triggered and the following steps are activated.

2. when the window at the automaton is open, the automaton process is suspended. We then wait for the window of migration at the scheduler; this is done by waiting for the current scheduling cycle to finish.

3. redirect the input to the old subplan to the new subplan; a data queue is

placed at the output of the new subplan as a buffer. The suspended automaton process is resumed. Now both the old subplan and the new subplan are running in parallel.

4. because the old subplan will not have new inputs, it will be cleaned up. It is then disconnected from the remaining plan (hence it will not be scheduled again). The buffer at the output of the new subplan is flushed to the input of the remaining plan. Now the output of the new subplan is directly connected to the remaining plan, and the migration process finishes.

In the above approach, the migration process is actually triggered by the unpredictable automata. Hence we are not really spending cpu time to "wait" for the automata. In contrast, if the migration is triggered by the scheduler, we would have to hold on the scheduler, and wait for the automata for an unpredictable time.

6.6 Discussion of Correctness

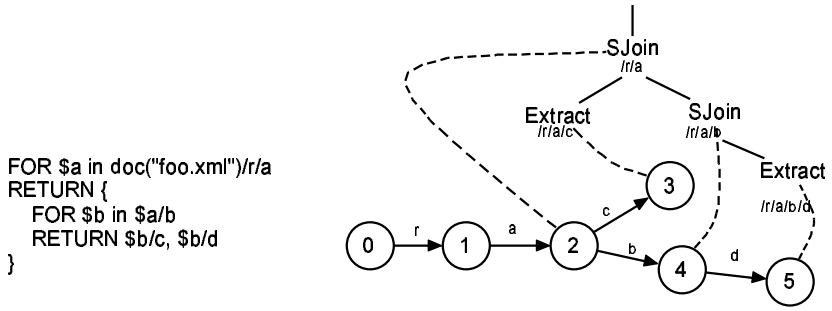

Figure 6.1: Query

Consider the example in Figure 6.1. Notice that the binding tree generates a tuple for each "a" element because the root of the tree binds to "/r/a". In fact, each tuple is generated independently of other tuples. This leads to the distributive

property of a binding tree. Let a_i denotes the ith "a" element, let f denote the output function that maps the input stream into output tuples, then the following holds.

$$f([a_1, a_2, ..., a_n]) = f(a_1) \circ f(a_2) \circ ... \circ f(a_n)$$

Based on this distributive property, we now show that a migration is correct if it does not divide any "a" elements into two parts such that the first part is read in before the migration and the second part thereafter. Let g denote the new binding tree of the new plan. Because the new binding tree is assumed to be correct, for any "a" element we have

$$f(a) = g(a)$$

Now suppose that migration is started after a_i is read in. The following shows that the output tuple stream with migration is equal to the output tuple stream without migration.

$$\begin{aligned} & f([a_1, a_2, ..., a_i, a_{i+1}, ..., a_n]) \\ = \; & f(a_1) \circ f(a_2) \circ ... \circ f(a_i) \circ f(a_{i+1}) \circ ... \circ f(a_n) \\ = \; & f(a_1) \circ f(a_2) \circ ... \circ f(a_i) \circ g(a_{i+1}) \circ ... \circ g(a_n) \\ = \; & f([a_1, a_2, ..., a_i]) \circ g([a_{i+1}, ..., a_n]) \end{aligned}$$

In fact, the migration process discussed in the previous section guarantees that an migration will not divide any "a" elements into two parts such that the first part is read in before the migration and the second part thereafter. This holds for all root binding of a binding tree, because the window of migration is close when any inside tag of the element is being read in, hence potentially dividing the binding element into two parts as above.

Chapter 7

Experimental Study

7.1 Introduction

This thesis as a whole is aimed to answer the following questions:

- Is it feasible to combine the automata model and the algebra model in a stream processing system?
- Are there optimization techniques that are specific to the automata/algebra integration?
- How can these optimization techniques be applied to a system where *a priori* data statistics are unknown?

Although these questions have been answered theoretically in the previous chapters, we now take an empirical approach.

7.2 Experimental Setup

We have implemented the Raindrop system in Java using Sun Java SDK version 1.4. The XML parser used is Xerces 1.0 for Java. We conducted the experiments on

a Pentium III 750 MHZ machine with 384MB memory running Microsoft Windows XP Professional. We allocate 384MB memory for the Java Virtual Machine in every experiment.

In order to minimize interference from other background applications, we close all optional services and applications. All results are kept in main memory during the execution, and only afterwards stored persistently for analysis to avoid disk I/O during execution. We run each experiment 10 times. Because initial results shows that the first run is sometimes inconsistent with the others [1], we omit all first run results.

All Source XML data is synthetically generated using the ToxGene XML generator [BMKL02]. The data schema is a slightly modified version of the auction DTD from [BMKL02]. Thanks to the ToxGene XML generator, we can conveniently change data statistics such as data size, selectivities over specific predicates, and data distribution.

Two queries used in the experiments are shown in Figure 7.1. They are both typical FLRW expressions. They differ in two parameters, i.e., the number of path expressions and whether the descendant axis is used in the path expressions. The query Q1 includes 2 path expressions and none of them use the descendant axis. The query Q4 includes 5 path expressions and all of them use the descendant axeses.

There are more queries used in the experiments than shown in Figure 7.1. They are all alike Q1 and Q4 except that they have different parameter values. For example, Q2 has 5 path expressions without descendant axis, Q3 has 20 path expressions without descendant axis, and Q5 has 20 path expressions with descendant axeses. From Q1 up to Q5, they roughly represent an increase of complexity in expected processing time.

[1] This may because of the initialization of the JVM

```
FOR $a in doc("bib.xml")/articles/article
WHERE $a/review/rate = 5
RETURN
   <best>
      $a/name
   </best>
```

Q1

```
FOR $a in doc("bib.xml")//article
WHERE $a//rate = 5
RETURN
   <best>
      $a//name,
      $a//alpha,
      $a//beta,
      $a//gamma
   </best>
```

Q4

Figure 7.1: Example queries used in the experiments

7.3 System Test and Throughput

The first set of experiments aims to answer the first question in Section 7.1, i.e., is it feasible to combine the automata model and the algebra model in a stream processing system. We also test the capability of the system by analyzing its throughput, i.e., how many inputs can the system process per unit of time.

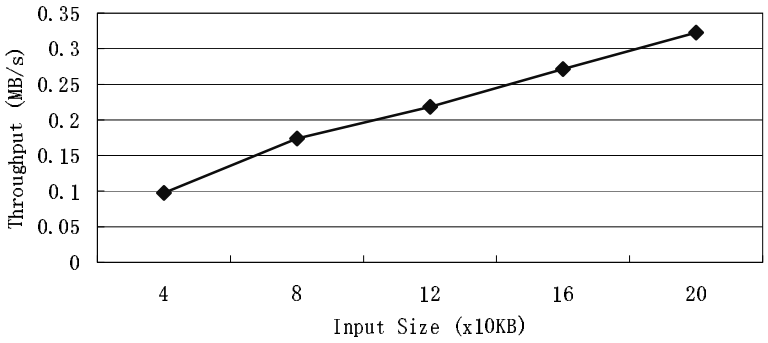

Figure 7.2: System Throughput with small input

Figure 7.2 shows the system throughputs, measured in MB/s, with different size of inputs. The throughput is calculated by dividing the input size by the elapse time of the entire execution. Only relatively small size inputs are shown in this figure. The throughput increases almost in linear over the input size, which suggests a startup overhead incurred in each run. To verify this hypothesis, we try some larger inputs.

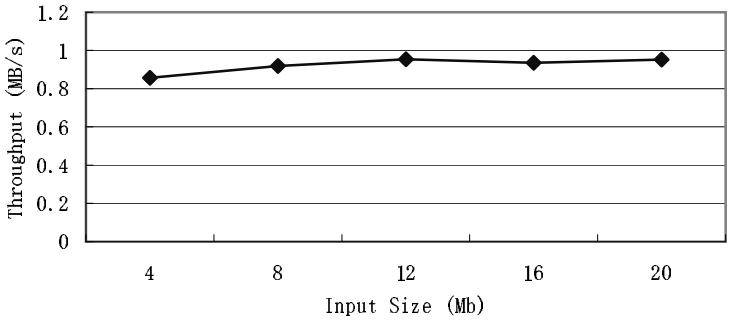

Figure 7.3: System Throughput with large input

Figure 7.2 measures system throughputs in the same manner as in Figure 7.2. But this time we use inputs 100 times larger than those in Figure 7.2. The throughput is almost constant in this case, which suggests the startup overhead is compensated with the longer execution time, which also confirms our hypothesis of startup overhead. When both small inputs and large inputs are combined in Figure 7.4, this effect of compensation is especially clear.

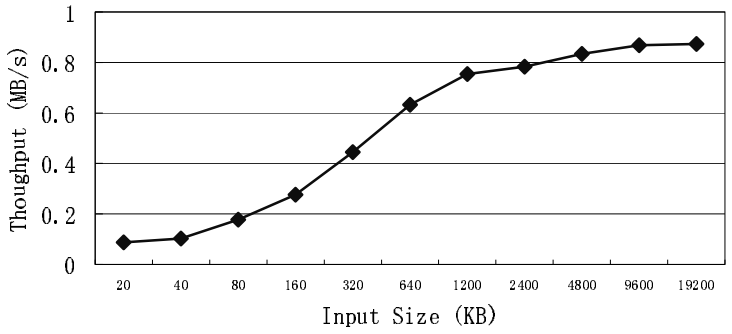

Figure 7.4: System Throughput with both small and larger input

7.4 Output Pattern

The second set of experiments aims to identify the output pattern, namely, the output rate over the entire execution period. This accounts for our preference of

prompt output in the stream context, i.e., partial result is generated as soon as enough input is received. In fact, it is due to this preference that we drop the "load-and-process" approach and instead adopt the "on-the-fly" approach.

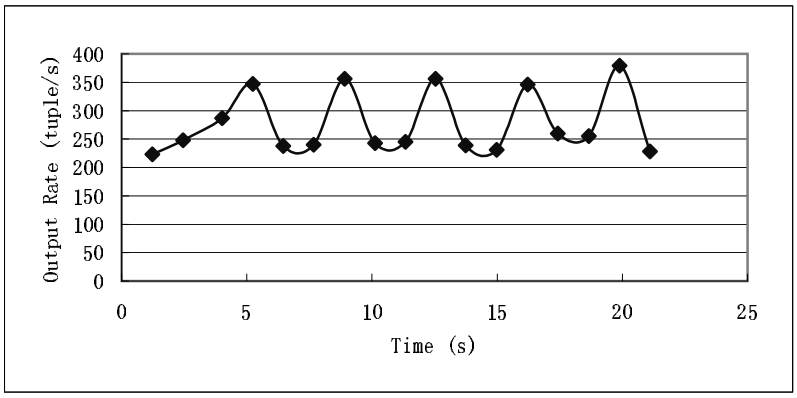

Figure 7.5: Output Pattern over time

Figure 7.5 depicts the number of output tuples in every 1.5 second window. In general, output tuples are generated continuously through the entire execution. The small turbulence of output rate is due to two factors: the uneven data distribution and the varying thread scheduling of JVM. In fact, if wider window is adopted, e.g., every 4 seconds, the output rate is basically smooth.

This relatively smooth output pattern also suggests that we can compare executions of different query plans by analyzing their finishing times. This measurement is adopted in all following experiments.

7.5 Cost Ingredients

This set of experiments aims to analyze the cost ingredients of different system modules. It also helps to identify which part of the system we shall look at more closely to compare different query plans.

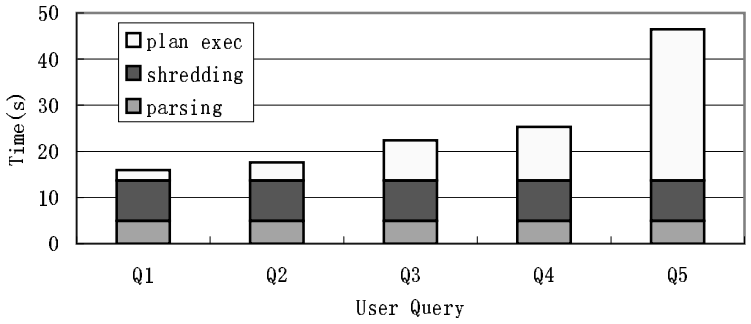

Figure 7.6: Cost Ingredients

In Figure 7.6, each bar shows the execution time of an individual query plan. Each component of a bar shows the elapse time of a system component. The parsing component parses the input XML stream into SAX events, or XML tokens as defined in Chapter 4. The time in parsing is constant for a given input and invariable with different query plans.

The shredding component stores the input XML tokens for future access. How to efficiently store XML is still a hot topic in current research. Because we do not focus on this issue, we assume a relatively simple scheme and adopt DOM-like data structures. The shredding time is also constant for a given input and unchanged with different query plans.

What changes with different query plans is the time of executing the query plans. If fact, this cost ingredient changes dramatically over different query plans. This suggests that in the following experiments, we shall look more closely on the plan execution time for query plan comparison.

7.6 The Pushin vs. Pullout Techniques

This set of experiments aims to analyze the pushin vs. pullout optimization technique (see Chapter 5). This is to answer the second question raised in Section 7.1,

i.e., are there optimization techniques that are specific to the automata/algebra integration. Our goal is to establish the heuristic that chooses proper query plan based on data selectivity. This heuristic will be the foundation for the next set of experiments.

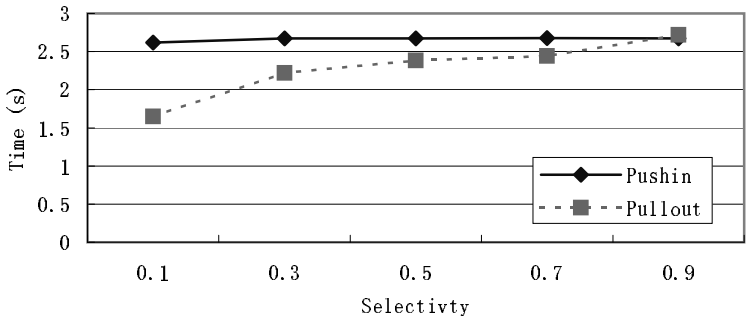

Figure 7.7: Comparing Pushin vs. Pullout with Q1

Figure 7.7 compares the pushin plan and the pullout plan running the query Q1. Each data point in the figure represents a separate run. The X axis represents different data selectivity over the *where* clause of the query. The Y axis represents the plan execution time of each query plan.

Let us first consider the pullout plan. The plan execution time increases over the data selectivity. Recall that the pullout plan first evaluates the predicate and only if the predicate is satisfied it evaluates other path expressions. This means for low selectivity where many tuples do not satisfy the predicate, the cost in evaluating other path expressions is saved. The lower the selectivity, the more the saving. The pushin plan, on the contrary, can not exploit this flexibility and results in a flat curve. This explains that for most selectivities the pushin plan performs more poorly than the pullout plan. It is only at the end of graph, with sufficiently high selectivity, that the two plans match each other.

With more path expressions in the query Q2, we find an interesting pattern (see Figure 7.8). The left end of the chart is similar to Figure 7.7. In the right end of the

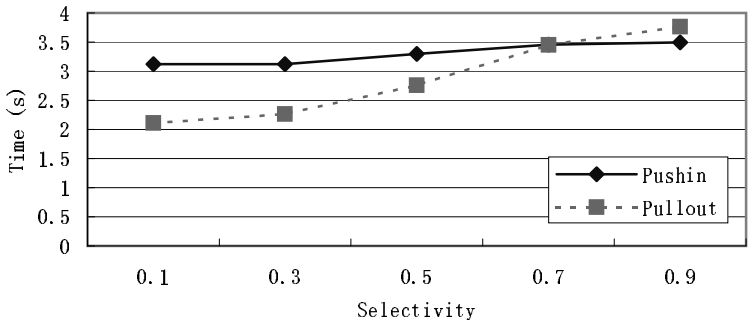

Figure 7.8: Comparing Pushin vs. Pullout with Q2

chart, however, the pushin plan outperforms the pullout plan. This confirms that, with higher selectivity when neither plan can exploit the aforementioned saving, the compact and prefix computation-sharing nature of automata is more efficient than that path expressions are evaluated one by one using non-automata technique.

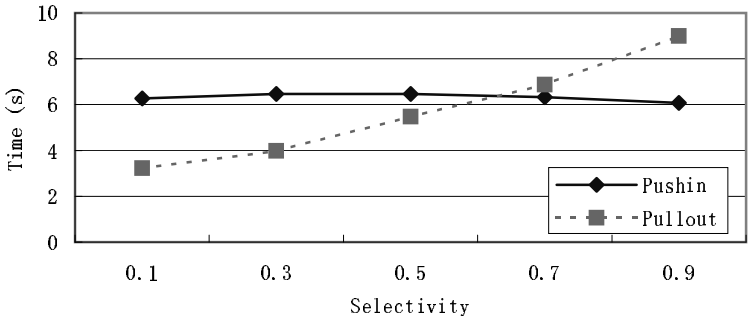

Figure 7.9: Comparing Pushin vs. Pullout with Q3

This pattern is shown more clearly in Figure 7.9, where Q3 includes 20 path expressions. It suggests a general heuristic where the pushin plan should be used with high selectivity and the pullout plan should be used with low selectivity.

When the descendant axis is used in the path expressions, which implies more complex computation in path evaluation, Figures 7.10 and 7.11 confirm the pushin vs. pullout heuristic over different selectivities.

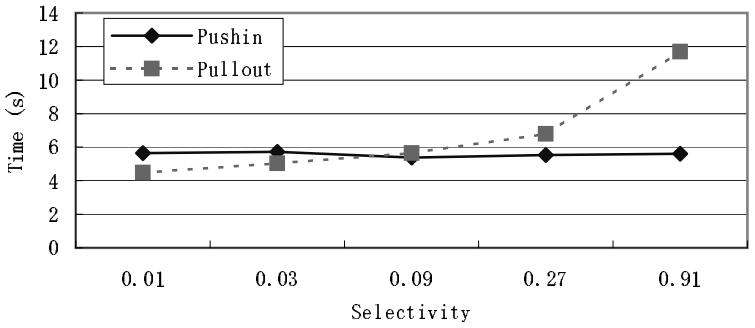

Figure 7.10: Comparing Pushin vs. Pullout with Q4

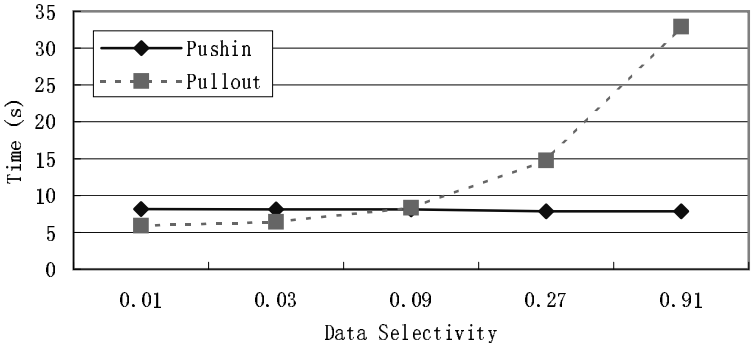

Figure 7.11: Comparing Pushin vs. Pullout with Q5

7.7 Runtime Optimization

Now we come to the last set of experiments that evaluates applying the pushin vs. pullout heuristic at runtime (see Chapter 6). This is based on the heuristic drawn from the last set of experiments, i.e., selectivity can be exploited to select the optimal plan. In this set of experiments, the data selectivity is monitored at runtime. Once its estimation is found to be different from the initial value, a new plan is generated and a plan migration process is initiated. The performance of this adaptive plan selection strategy is then compared with the static ones.

Figure 7.12 shows the results running with the query Q3. Ignoring the curve representing the adaptive strategy, this figure is identical to Figure 7.9. When

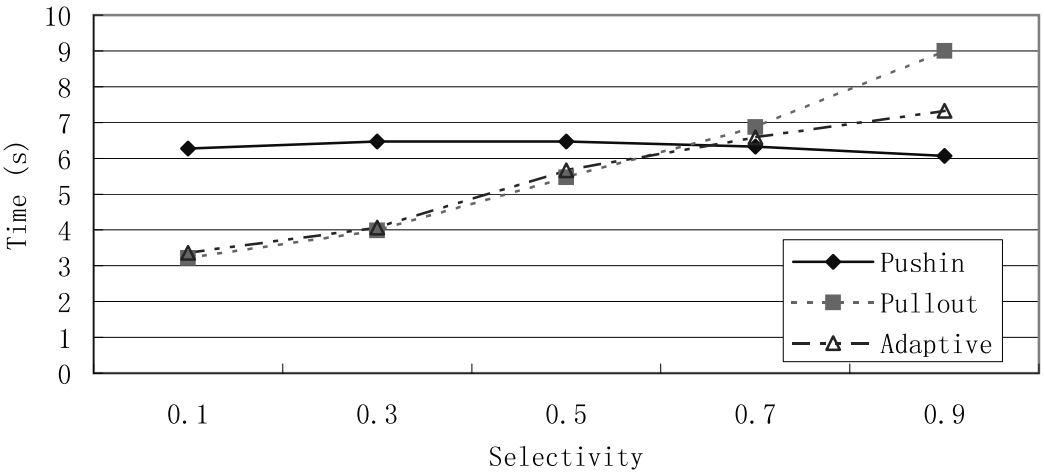

Figure 7.12: Evaluating runtime optimization with Q3

this adaptive strategy is compared with the other two, we find that with lower selectivities it resembles the pullout plan, while with higher selectivities it resembles the pushin plan. It always takes a little more time than then best plan, due to the overhead of monitoring the data statistics and of plan migration.

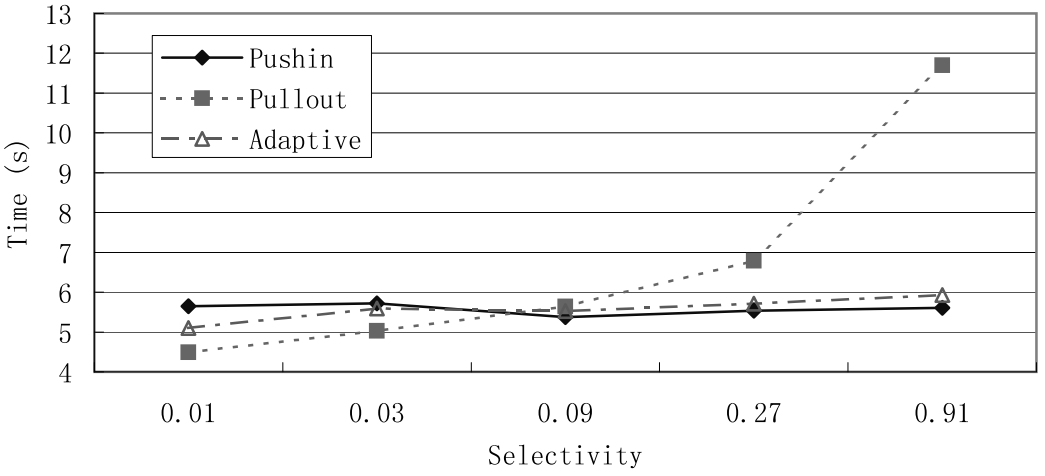

Figure 7.13: Evaluating runtime optimization with Q4

Running other queries confirms the above observation, as shown in Figures 7.13 and 7.14. This in fact answers the third question in Section 7.1, i.e., how can these

optimization techniques be applied to a system where *a priori* data statistics are unknown.

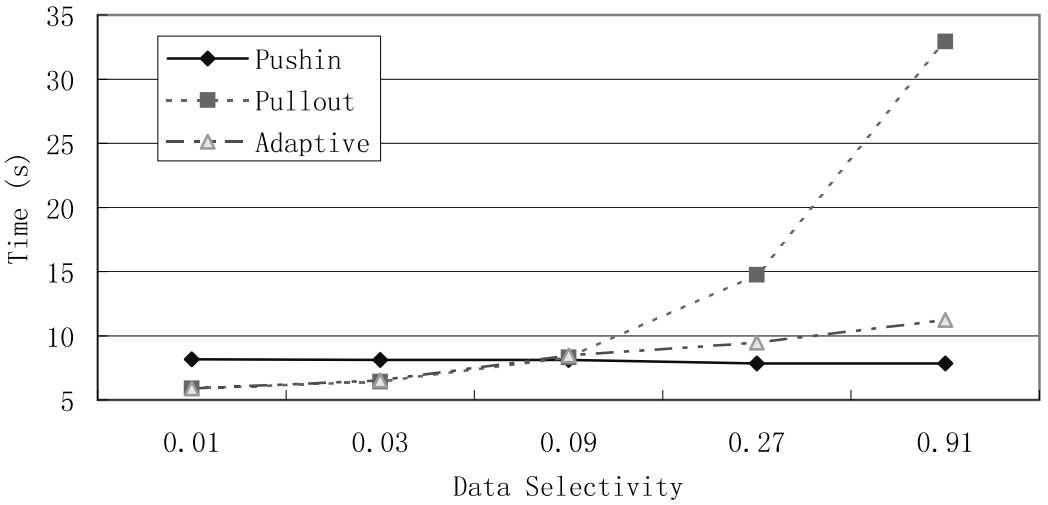

Figure 7.14: Evaluating runtime optimization with Q5

7.8 Discussion

We have experimentally evaluated our prototype system. More specifically, we have verified the pushin vs. pullout optimization technique, applied both statically and adaptively. However, we have not compared our system with other systems. This is mainly because integrating automata-based model and algebraic model in XML stream processing is a still an unexploited area in the literature. Current efforts in pure automata-based systems such as those in SIGMOD 2003 [GS03, PC03] focus only on processing XPath queries. Pure algebraic systems [CFI+00, BFHR02, ZPR02], on the other hand, focus only on non-stream data.

Chapter 8

Related Work

In this chapter we shall briefly overview previous works that are related to this thesis. We roughly divide them into three categories: the automata-based stream processing systems, the algebra-based query engines, and other works discussing runtime optimization.

8.1 Automata-based Stream Processing Systems

For works in this category, the goal is generally to develop a compact and minimal system that can handle a very lager number (e.g., thousands or tens of thousand) of simple XPath queries. It would be too expensive to process the queries one by one. Hence it is the common strategy to process all queries in parallel, or in other words, with one scan of the input.

The problem of processing a very large number of simple queries can be traced at least as early as XFilter [AF00]. It introduces the so-called *selective dissemination of information*, in which information such as email and user news are filtered and sent to users who have registered their interest or queries to the system. XFilter handles a subset of the XPath language, or simple XPaths that consist of node test

with child axis and descendant axis and without predicates. One notable feature of XFilter is that it can process a large number of queries. For this purpose, it builds a prefix tree that can share computation between identical prefixes. Although the prefix tree has no explicit state (as in automata), the token-at-a-time operation largely resembles the state-transition model.

XTrie [CFGR02] is also an early work that processes simple XPaths. Unlike XFilter that intermixes the static prefix tree with runtime information, XTrie isolates the operation and the runtime information from the prefix tree, which results in a cleaner system model.

X-Scan [ILW00] is another similar work. It explicitly adopts the state-transition model, or finite machine. In fact, X-Scan is a query operator in a data integration system that based on an algebraic model. This system, or Tukwila as they call, will also be discussed in the next two sections.

YFilter [DFFT02] is a continued work of XFilter. Like X-Scan, it adopts an explicit state-transition model, or a modified NFA. This work discusses predicates and conducts a set of experiments to test the idea of "push in" predicates into the automata. However, the result is generally negative and advocates the "predicate pull out" strategy.

[MGOS03] also explicitly adopts the state-transition model. But unlike YFilter, it adopts a deterministic model, or a modified DFA. It shows that the number of states will not increase exponentially in most situations if the states are constructed "lazily". However, some of its experiments fail to complete due to memory overflow, which suggests the number of states may still exceed the capacity of normal computers.

Transducer [LMP02] adopts XQuery as its query language. Although it handles only a subset of XQuery, its approach to attach data buffers and buffer operations to transitions is well recognized and has many impacts on later work. Another feature

of Transducer is that it adopts an automatic-code-generating approach, where an XQuery is first mapped to automata and then C source codes are generated from the automata.

Two recent works that are published in SIGMOD 2003 also takes the buffer-enhanced automata model. XPush [GS03] aims to process XPath queries with predicates, which in essence equals to simple FLWR XQuery expressions except for the result construction. A very similar approach is adopted in the XSQ system [PC03], which supports features such as multiple predicates, closures, and aggregation.

8.2 Algebra-based Query Engines

Unlike the works in the automata category where large number of simple queries are assumed, systems with query algebra focus on one or a small number of complex queries. Query optimization is very important in evaluating these complex queries. Hence the flexibility and maturity of query algebra is much preferred.

Historically, using algebra for query processing can be traced back to the famous research paper by Codd [Cod70]. Afterwards, a large volume of research efforts have gone to the relational data model and then to the object-oriented model, resulting in mature and industry-strength algebraic query optimizations.

When this algebraic approach is applied to the XML and the XQuery language, research efforts are divided into two camps. One camp extends the off-the-shelf relational or object-relational databases to support XML. XPeranto [CFI+00] is the pioneer work that *shreds* structured XML documents into flat relational tables. LegoDB [BFHR02] exploits the space of this structured-flat mapping and selects the best mapping strategy based on data statistics. The Rainbow research project [ZPR02] takes a similar approach and develops optimization techniques such as computation push down and schema clean up.

Another camp of research abandons the structured-flat mapping because such a mapping often results in either an unnormalied relational representation or in a very large number of tables [JAKL+02]. Timber [JAKL+02] is a native XML database system that builds novel storage and index data structures, which can accelerate certain operations such as structural join.

Besides the above two camps, Tukwila [IHW02] is a data integration system that develops an X-Scan operator, which is like a scan operator in the relational databases, to hide the XML nested structure from the rest of the system. Hence the X-Scan operator can be viewed as a kind of wrapper over XML data sources, while the rest of they system remains relational.

8.3 Runtime Optimization

Runtime optimization is a less popular topic in the database literature compared with other optimization techniques. This is due to the fact that data statistics are readily available to the optimizer at compile time so that the optimizer can generate optimal or near-optimal query plans. It is also due to the fact that traditional queries are one-time queries that usually finish in a very short time, which shrinks the space of runtime optimization.

Nevertheless, there are a few research papers related to runtime optimization. [CG94] considers the incomparability of costs at compile time and develops a *choose-plan* operator to choose proper join orders at runtime. The mid-query re-optimization technique [KD98] annotates the query execution plan and acquired data statistics at runtime to instantiate the annotation, which allows the optimizer to choose, for example, an indexed-based join or a hash join at runtime. Tukwila [IHW02] also involve runtime optimization. It advocates the *convergent query processing* and develops a framework under which every query operator can be arbitrarily interrupted

and replaced at runtime.

The Telegraph CQ [Cha03] is a continuously adaptive query processing engine, which is based on the adaptive routing module Eddy [AH00]. In Eddy, intermediate tuples are routed to different query operators or the so-called *SteMs*. This suggests that explicit query plans are abandoned and, for example, a set of join operators can be executed in any order. However, operator dependencies are still retained through the *ready-bit* and *done-bit* mechanism, which suggests that operators are executed in stages. The operators that a tuple can be routed to are limited to those in a certain stage.

Chapter 9

Conclusions

9.1 Conclusions

In this thesis, we present a framework to integrate the automata model and the algebraic model for efficient XML stream processing. By implementing the Raindrop system, we prove that this integrated model can be applied to XQuery evaluation over continuous XML data streams.

We also show that the integration is flexible, in that we develop optimization techniques to move query functionality into or out-of the automata. More importantly, we show that this optimization technique can be applied uniformly as other well-studied optimization techniques through query rewriting. This capability allows us to reason about query logic and optimization at the algebraic level, and only thereafter play with the implementation details specific to the automata or to the general query plan. By theoretical and empirical study, we identify interesting patterns regarding the pushin vs. pullout trade-off. Based on this result we develop heuristics to apply this optimization technique based on data statistics.

To tackle the *a priori* statistics unknown problem, we develop runtime optimization techniques. We collect data statistics at runtime, and by identifying safe

migration points for both the automata and the scheduler, we adapt the query plan at runtime.

9.2 Future Work

Due to the limited amount of time allocated to this thesis, there are quite a few issues that arise but were not exploited during this research. Here we list the topics that could be undertaken to continue this work.

- **Join over streams:** we have almost completely ignored the join operation over different data streams. This is due to the fact that the semantics of stream join, especially the order issue, is not yet defined. However, because join operations incur one of the major cost factors in relational databases and likely also the case in the XML realm, optimization involving join is certainly a promising future direction.

- **Multiple queries:** we have only considered optimizations with a single query. Although in principle our optimization techniques can also be applied to a query plan that is merged from multiple queries, we certainly have not focused on this issue per se. It is almost for sure that with larger number of user queries and with more complex query plans certain interesting trade-offs will occur and lead to other interesting optimization techniques.

- **Integration beyond path evaluation:** we have mainly considered moving path evaluations into or out-off the automata. Given the assumption that arbitrary data buffers and buffer operations can be attached to the automata, there is in fact no limit to what the automata can do, in particular, selection, projection, join, and even aggregation. From this perspective, the only

difference between automata and algebraic query plans is the scheduling strategy. We have exploited this difference for path evaluation. But certainly we can also exploit this difference for other operations. Following this direction would lead to a completely unified query model, which I would speculate to be a promising research direction.

Bibliography

[A+76] M.M. Astrahan et al. System R: a relational approach to database management. *ACM Trans. on Database Systems*, pages 97–137, 1976.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on The Web*. Morgan Kaufmann Publishers, 2000.

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[BFHR02] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, and Maya Ramanath. Legodb: Customizing relational storage for xml documents. In *VLDB Demostration*, 2002.

[BMKL02] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: a template-based data generator for xml. In *WebDB 2003*, 2002.

[CCC+02] 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 *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002. To appear.

[CFGR02] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. ICDE*, pages 235–244, 2002.

[CFI+00] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, pages 105–110, 2000.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD*, pages 150–160, 1994.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pages 34–43, June 1998.

[Cha03] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003*, 2003.

[Cod70] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[DFFT02] Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of ICDE*, pages 341–344, 2002.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, pages 73–170, June 1993.

[GS03] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD 2003*, page 419, 2003.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison-Wesly, second edition, 2001.

[IHW02] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4), 2002.

[ILW00] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical Report UW-CSE-2000-05-02, University of Washington, 2000.

[JAKL+02] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.

[KD98] N. Kabra and D.-J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.

[LMP02] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. VLDB*, pages 215–226, 2002.

[M+03] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, pages 245–256, 2003.

[MGOS03] G. Miklau, T. J. Green, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *ICDT*, pages 173–189, 2003.

[PC03] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD*, page 431, 2003.

[sax] Simple api for xml. www.saxproject.org.

[W3C] W3C. Extensible markup language (XML). http://www.w3.org/XML/.

[W3C02a] W3C. XML path language (xpath) version 1.0. http://www.w3.org/TR/xpath, November 2002.

[W3C02b] W3C. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, November 2002.

[Wat97] B. W. Watson. Practival optimization for automata. In *Proc. of the 2nd International Workshop on Implementing Automata*, 1997.

[ZPR02] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I shrunk the XQuery! — an XML algebra optimization approach. In *Proceedings of the fourth international workshop on Web information and data management*, pages 15–22, Nov 2002.