

Query Processing and Optimization in Native XML Databases

Ning Zhang

David R. Cheriton School of Computer Science
University of Waterloo
nzhang@uwaterloo.ca

TECHNICAL REPORT CS-2006-29 AUGUST 2006



Abstract

XML has emerged as a semantic markup language for documents as well as the *de facto* language for data exchange over the World Wide Web. Declarative query languages, such as XPath and XQuery, are proposed for querying over large volumes of XML data. A number of techniques have been proposed to evaluate XML queries more efficiently. Many of these techniques assume a tree model of XML documents and are, therefore, also applicable to other data sources that can be explicitly or implicitly translated into a similar data model.

The focus of this thesis is on efficient evaluation and optimization of path expressions in native XML databases. Specifically, the following issues are considered: storage system design, design of physical operators and efficient execution algorithms, and the cost-based query optimizer.

The proposed storage system linearizes the tree structure into strings that can be decomposed into disk pages. Simple statistics are kept in the page headers to facilitate I/O-efficient navigation. Based on this storage system, a hybrid approach is developed to evaluate path expressions that exploit the advantages of navigational and join-based approaches. Path expressions that only contain “local” axes (**child**, **parent**, **attribute**, **self**, **following-sibling**, and **preceding-sibling**) are evaluated by means of the proposed “Next-of-Kin” (NOK) operator. A general path expression that contains both local axes and global ones (**ancestor**, **descendant**, **ancestor-or-self**, **descendant-or-self**, **following**, and **preceding**) is decomposed into NOK subtrees whose intermediate results are structurally joined to produce the final result. Experiments show that the navigational operator can be an order of magnitude faster than join-based approaches in some cases, but slower in others. Thus a cost-based query optimizer is necessary to choose the optimal execution plan based on estimates of the cost of each operator.

The cost of an operator heavily depends on the cost model and its input. The inputs to the cost model are usually the cardinalities of path expressions. In this thesis, a synopsis structure called XSEED is proposed to estimate the cardinality of a path expression. An XSEED synopsis can be constructed by compressing an XML document to a small kernel first, and then more information can be added to the synopsis. XSEED results in more accurate cardinality estimation than previous approaches and is easier to construct, easier to update, and can incorporate query feedback.

Efficient query execution exploits indexes. The last component of the thesis is a feature-based structural index, called FIX, to expedite tree matching on a large tree. This is based on the observation that the navigational operation is expensive and applying it to every tree node is very inefficient. FIX extracts distinctive features from subtrees and uses them as the index keys. Similarly, for each incoming query, the features of the query tree are extracted and used as search keys to retrieve the candidate results. Thus, it is sufficient to match only on these candidate results. The experimental results show that the pruning power of FIX is very high – more than 99% for structure-rich data sets, and more than 20% for data sets with less structural variety.

List of Tables

1.1	Thirteen axes and their abbreviations	11
2.1	Feature comparisons between different storage techniques	39
2.2	Feature comparison between different synopses	54
3.1	Statistics of testing data sets	81
3.2	Testing query categories	82
3.3	Runtime comparisons	84
4.1	Index construction time	115
4.2	Implementation-independent metrics	118
5.1	Hyper-edge table	147
5.2	Synopses construction time	150
5.3	Performance in estimation accuracy	152
5.4	Estimation vs. Evaluation ratios	154

List of Figures

1.1	Web services workflow	6
1.2	a snippet of HTTP log file	7
1.3	An example XML snippet	8
1.4	Example pattern tree	12
1.5	XDB modules and workflow	19
2.1	DTD for XML file in Figure 1.3	23
2.2	The DTD Graph for the DTD in Figure 2.1	25
2.3	XML-to-relational mapping: Shanmugasundaram et al. [122]	26
2.4	XML-to-relational mapping: STORED	27
2.5	XML-to-relational mapping: the Edge-based Approach	30
2.6	XML-to-relational mapping: Interval Encoding	31
2.7	XML-to-relational mapping: XRel	34
2.8	Relational-to-XML mapping: XPath Accelerator	36
2.9	Native XML storage: Natix	37
2.10	Native XML storage: Arb	38
2.11	An automaton corresponding to a path expression	44
3.1	Axes statistics from XQuery Use Cases	58
3.2	Converting axes to $\{., /, //, \blacktriangleleft\}$	60
3.3	Converting axes to $\{., /, //, \triangleleft\}$	62
3.4	NoK Decomposition and relation of NoK pattern matching	65
3.5	An XML file and a pattern tree	66
3.6	Subject tree representation	67

3.7	Data file and auxiliary indexes	73
3.8	The string representation of an XML tree	74
3.9	Page layout for structural information	76
4.1	An XML document and its F&B bisimulation graph	91
4.2	Bisimulation graph for Figure 4.1a	94
4.3	Edge-weighted and matrix representations of bisimulation graph in Figure 4.2	99
4.4	Building and querying indexes	103
4.5	Clustered and Unclustered FIX Indexes	104
4.6	Data Structures: BiSimGraph and Signatures.	106
4.7	Average implementation independent metrics	118
4.8	Runtime comparisons on XMark, Treebank, and DBLP	119
4.9	DBLP with values	123
5.1	Cardinality estimation process using XSEED	128
5.2	An example XML tree and its XSEED kernel	131
5.3	Counter stacks for efficient recursion level calculation	135
5.4	Update synopsis	136
5.5	Example of ancestor independence assumption breaks	144
5.6	Estimation errors for different query types on DBLP	152
5.7	Different MBP settings on DBLP	153

List of Algorithms

1	Decomposing a pattern tree into NoK pattern trees	64
2	NoK Pattern Matching	70
3	Primitive Tree Operations	79
4	Constructing FIX for a Collection of Documents	107
5	Index Query Processing	113
6	Constructing the XSEED Kernel	134
7	Synopsis Traveler	140
8	Synopsis Matcher	143

Chapter 1

Introduction

The Extensible Markup Language (XML) [10] has attracted significant research interest due to its expanding use in various applications. Proposed as a simple syntax with flexibility, human-readability, and machine-readability in mind, XML has been adopted as a standard representation language for data on the Web. Hundreds of XML schemata (e.g., XHTML [9], DocBook [2], and MPEG-7 [4]) are defined to encode data into XML format for specific application domains. Implementing database functionalities over collections of XML documents greatly extends our power to manipulate these data. For example, a search engine equipped with such functionalities will be able to answer complex queries such as “find the lowest priced book titled ‘Advanced Unix Programming’ written by Richard Stevens in all Canadian book stores”.

In addition to be a data representation language, XML also plays an important role in data exchange between Web-based applications such as Web services. Web services are Web-based autonomous applications that use XML as a *lingua franca* to communicate. A Web service provider describes services using the Web Service Description Language (WSDL) [40], registers services using the Universal Description, Discovery, and the Integration (UDDI) protocol [6], and exchanges data with the service requesters using the Simple Object Access Protocol (SOAP) [5] (a typical workflow can be found in Figure 1.1). All these techniques (WSDL, UDDI, and SOAP) use XML to encode data. Database techniques are also beneficial in this scenario. For example, an XML database can be installed on a UDDI server to store all registered service descriptions. A high-level declarative XML

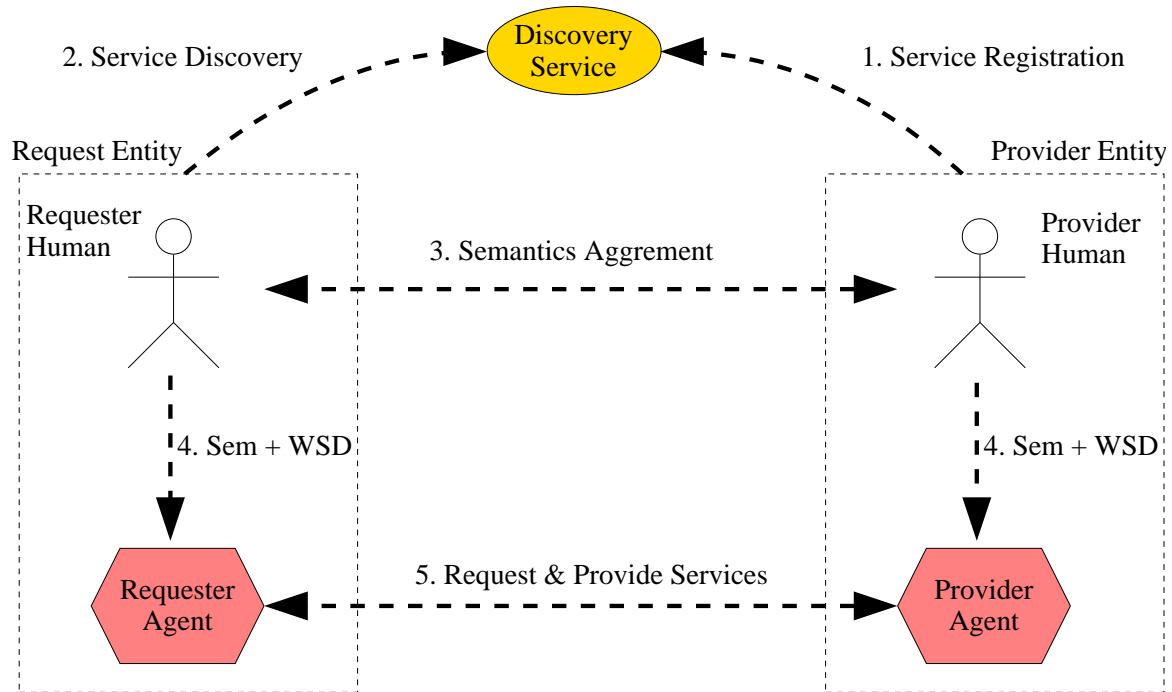


Figure 1.1: A typical Web Service workflow suggested by the W3C Web Services Architecture [8]

query language, such as XPath [22] or XQuery [24], can be used to match specific patterns described by a service discovery request.

XML is also used to encode (or annotate) non-Web data. These non-Web data are usually semistructured or unstructured. Annotating unstructured data with semantic tags to facilitate queries has been studied in the text community for a long time (e.g., the OED project [65]). In this scenario, the primary objective is not to share data with others (although one can still do so), but to take advantage of the declarative query language developed for XML to query the structure that is discovered through the annotation. For example, Figure 1.2 shows a snippet of network log file for HTTP requests. Each line in the log file includes different types of information (e.g., IP address, time, and request status etc.). When these types of information are parsed, annotated with XML tags, and stored in an XML database, a database query can be posed to help finding potential denial-of-service intrusions: find the IP addresses that requested more than 100 failed requests in a

```
125.250.248.130 -- [11/Mar/2006:16:06:53 -0500] "POST /xmlrpc.php HTTP/1.1" 404 284
125.250.248.130 -- [11/Mar/2006:16:06:57 -0500] "POST /wordpress/xmlrpc.php HTTP/1.1" 404 294
64.151.107.252 -- [12/Mar/2006:00:11:47 -0500] "GET /phpmyadmin/index.php HTTP/1.0" 404 282
217.35.79.90 -- [12/Mar/2006:03:56:02 -0500] "GET /w00tw00t.at.ISC.SANS.DFind:) HTTP/1.1" 400 403
129.97.128.230 -- [12/Mar/2006:04:02:14 -0500] "HEAD / HTTP/1.0" 200 0
124.0.225.70 -- [12/Mar/2006:09:32:35 -0500] "GET /cacti//graph_image.php HTTP/1.1" 404 296
```

Figure 1.2: a snippet of HTTP log file

continuous 5 second period.

All the above examples demonstrate some of the potential benefits that database functionality on XML data can bring to these applications. As more and more data are encoded into XML format, the demands of managing XML data in an XML database are increased. This thesis develops techniques for storage, query processing, and query optimization over XML databases.

1.1 Background

This section first introduces the basic definitions for the XML data model and XML query languages. A detailed discussion of related work is included in Chapter 2.

Database systems organize data in an abstract data model, which forms the basis for expressing queries. XML documents are usually modeled as trees [54]. This data model is used in a simplified version to suit the objectives of this thesis, since many features, such as typing and schema validation, are not the focus of the thesis.

XML markups (or tags) divide data into pieces called *elements*, with the objective to provide more semantics to the data. Elements can be nested but they cannot be overlapped. Nesting of elements represents hierarchical relationships between them. As an example, Figure 1.3 is a snippet of bibliography data with XML markup.

Definition 1.1 (XML Document) An XML document contains a *root element*, which has zero or more nested subelements (or *child elements*), which can recursively contain subelements. For each element, there are zero or more *attributes* with atomic values (or CDATA) assigned to them. An element also contains an optional value (or PCDATA). A

```
<bib>
  <book year = "1999">
    <title> Principles of Distributed Database Systems </title>
    <author> M. Tamer Ozsu </author>
    <author> Patrick Valduriez </author>
    <price currency = "USD"> 98.00 </price>
  </book>
  <article year = "1987">
    <title> Mind Your Grammar: a New Approach to Modeling Text </title>
    <author> Gaston H. Gonnet </author>
    <author> Frank Wm. Tompa </author>
    <published_in> VLDB'87 </published_in>
  </article>
</bib>
```

Figure 1.3: An example XML snippet

total order, called *document order*, is defined on all elements and attributes¹ corresponding to the order in which the first character of the elements or attributes occurs in the document. □

For instance, the root element in Figure 1.3 is `bib`, which has two child elements: `book` and `article`. The `book` element has an attribute `year` with atomic value “1999”. Element `book` also contains subelements (e.g., the `title` element), and an element can contain a value (e.g., “Principles of Distributed Database Systems” for the element `title`).

The above definition simplifies the original XML definition [10] by removing auxiliary information such as comments, namespaces (NS), and processing instructions (PI). Another omitted feature is IDREFs, which define references between elements. IDREFs are not widely used in practice (see XQuery Use Cases [32]), and they make XML a more complex graph data model, which is much harder to manage. Following other XML data management system prototypes (e.g., Niagara [102] and TIMBER [79]), this thesis implements a system prototype for a core subset of XML data and queries. IDREFs and other

¹In the XQuery and XPath data model [54], the document order is undefined on attributes and it is implementation dependent. This thesis defines document order on attributes as well.

auxiliary or advanced features are considered for future work.

Definition 1.2 (XML Tree) An XML document is modeled as a ordered, node-labeled tree $T = (V, E)$, where there is a special node *root* corresponding to the document itself. Each non-root node $v \in V$ corresponds to an element/attribute/text and is characterized by:

- a unique identifier denoted by $ID(v)$;
- a unique *kind* property, denoted as $kind(v)$, assigned from the set $\{\text{element, attribute, text}\}$;
- a label, denoted by $label(v)$, assigned from some alphabet Σ .

A directed edge $e = (u, v)$ is included in E if and only if:

- $kind(u) = kind(v) = \text{element}$, and v is a subelement of u ; or
- $kind(u) = \text{element} \wedge kind(v) = \text{attribute}$, and v is an attribute of u ; or
- $kind(u) \in \{\text{element, attribute}\} \wedge kind(v) = \text{text}$, and v is the text value of u . □

As in Definition 1.1, the above definition omits the comment nodes, namespace nodes, and PI nodes from the XQuery Data Model [54].

Definition 1.3 (XML Data Model) An instance of XML data model is an ordered collection (sequence) of XML tree nodes or atomic values. □

Using the definition of XML data model and instances of this data model, it is now possible to define the query languages. Expressions in XML query languages take an instance of XML data as input and produce an instance of XML data as output. XPath [22] and XQuery [24] are two important query languages proposed by W3C. Path expressions are present in both query languages and are arguably the most natural way to query the hierarchial XML data. In this thesis, the focus is on *path expressions*. Other constructs such as FLWOR expressions or user-defined functions are beyond the scope of this thesis.

Definition 1.4 (Path Expression) A path expression consists of a list of steps, each of which consists of an axis, a name test, and zero or more qualifiers. There are in total thirteen axes, which are listed in Table 1.1 together with their abbreviations if any. Throughout the rest of the thesis, the abbreviations will be used whenever applicable. A name test filters nodes by their element or attribute names. Qualifiers are filters testing more complex conditions. The brackets-enclosed expression (which is usually called a *branching predicate*) can be another path expression or a comparison between a path expression and an atomic value (which is a string). The syntax of path expression is listed as follows:

$$\begin{aligned}
 \text{Path} &::= \text{Step} ("/" \text{Step})^* \\
 \text{Step} &::= \text{axis} " :: " \text{NameTest} (\text{Qualifier})^* \\
 \text{NameTest} &::= \text{ElementName} \mid \text{AttributeName} \mid " * " \\
 \text{Qualifier} &::= "[\text{Expr} "]" \\
 \text{Expr} &::= \text{Path} (\text{Comp} \text{Atomic})? \\
 \text{Comp} &::= " = " \mid " > " \mid " < " \mid " > = " \mid " < = " \mid " ! = " \\
 \text{Atomic} &::= " ' \text{String} ' "
 \end{aligned}$$

The last step in the list is called a *return step*. □

According to the W3C XQuery Working Draft [24], a “//”) at the beginning of a path expression is an abbreviation for the initial steps `document-node()/ descendant-or-self::node()/`. A “//” in the middle of a path expression, such as `A//B`, is an abbreviation for `/descendant-or-self::node()/B`. Since this thesis uses a simplified data model that only deals with tree nodes, a `::node()` is redundant and thus a “//” is simply treated as an abbreviation for the `descendant-or-self` axis (as listed in Table 1.1).

While the path expression defined here is a fragment of the one defined in XQuery [24] (by omitting features related to comments, namespaces, PIs, IDs, and IDREFs), this definition still covers a significant subset and can express complex queries. As an example, the path expression `//book[author/last = "Stevens"][price < 100]` find all books written by Stevens with the book price less than 100.

Axes	Abbreviations of /axis::
child	/
descendant	
parent	
attribute	/@
self	.
descendant-or-self	//
ancestor	
following-sibling	
following	
preceding-sibling	
preceding	
ancestor-or-self	
namespace	

Table 1.1: Thirteen axes and their abbreviations, if any

As seen from the above definition, path expressions have three types of constraints: the *tag name constraints*, the *structural relationship constraints*, and the *value constraints*. The tag name, structural relationship, and value constraints correspond to the name tests, axes, and value comparisons in the path expression, respectively. A path expression can be modeled as a tree, called a *pattern tree*, which captures all three types of constraints.

Definition 1.5 (Pattern Tree) A path expression can be mapped into a pattern tree $G(V, E)$ as follows, where V and E are sets of vertices and edges, respectively:

- each step is mapped to a node in V ;
- a special root node is defined as the parent of the tree node corresponding to the first step;
- if one step m immediately follows another step n , then the node corresponding to m is a child of the node corresponding to n ;
- if step m is the first step in the branching predicate of step n , then the node corresponding to m is a child of the node corresponding to n ;

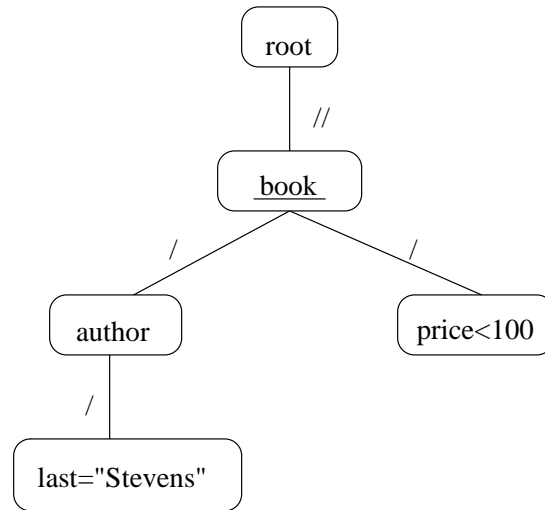


Figure 1.4: A pattern tree of expression `//book[author/last = "Stevens"][price < 100]`

- if two nodes represent a parent-child relationship, then the edge in E between them is labeled with the axis between their corresponding steps;
- the node corresponding to the return step is marked as the return node;
- if a branching predicate has a value comparison, then the node corresponding to the last step of the branching predicate is associated with an atomic value and a comparison operator. □

For example, the pattern tree of the path expression `//book[author/last = "Stevens"][price < 100]` is shown in Figure 1.4. In this figure, the node `root` is the root node and the return node is underlined. This path expression specifies the three types of constraints

in the following formula²:

$$\{ b \mid \text{tag}(b) = \text{“book”} \wedge \exists a, l, p \text{ tag}(a) = \text{“author”} \wedge \\ \text{tag}(l) = \text{“last”} \wedge \text{tag}(p) = \text{“price”} \wedge \\ \text{value}(l) = \text{“Stevens”} \wedge \text{value}(p) < 100 \wedge \\ \text{descendant}(\text{root}, b) \wedge \text{child}(b, a) \wedge \text{child}(a, l) \wedge \\ \text{child}(b, p) \}$$

where $\text{tag}()$ defines the tag-name constraint, $\text{value}()$ defines the value constraint, and $\text{child}()$ and $\text{descendant}()$ define the structural relationship constraints.

In order to test the proposed techniques, this thesis further defines three special cases of path expressions.

Definition 1.6 (Simple, Branching, and Complex Path Expressions) A *simple path expression* is a special path expression without branching predicates. Furthermore, all axes in a simple path expression are $/$ -axes. A *branching path expression* relaxes simple path expression by allowing an arbitrary number of branching predicates, but all axes are still $/$ -axes and no value comparisons in the branching predicates. A *complex path expression* further relaxes branching path expressions by allowing $//$ -axes for any steps, but still no value comparisons. □

1.2 Thesis Scope

The focus of this thesis is on efficient evaluation and optimization of path expressions in native XML databases. The reasons for this focus are the following: (1) path expressions are expressive and new to database management systems; path expressions specify special regular expressions on trees that are awkward or inefficient using relational or object-oriented query languages (SQL or OQL); (2) path expressions are ubiquitous in many XML query languages (e.g., XPath [22], XQuery [24], and XSLT [42]); and (3) path expression processing is complex. Naïve processing strategies result in exponential blowup [66] in

²This formula employs a set semantics and the returning b elements are ordered in document order.

terms of the query size. Therefore, efficient evaluation and query optimization techniques are crucial and are studied in this thesis.

There are three important problems related to the processing and optimization of path expressions:

1. What is the storage system for XML documents to support efficient evaluation as well as storing and updating XML documents?
2. How to evaluate path expressions efficiently for different types of queries?
3. How to choose which physical operators are the best ones given a path expression and an XML database?

There are several desirable features for the XML database storage systems. First the storage system should be *robust* enough to store any XML documents with arbitrary tree depth or width, with any element-name alphabet, and with or without associated schemata. Furthermore, the system should store sufficient auxiliary information to support efficient path query processing. This is in contrast to the unindexed BLOB or CLOB storage scheme which needs to parse the document before every query. Moreover, local update to the document should not cause drastic changes to the whole storage system. Therefore, the design of the storage system should trade off between the query performance and update costs.

One of such possible tradeoffs is to design a “multipurpose” storage system that performs reasonably well on both query and update, and to design an optional index for more efficient query processing. Accordingly, in the query processing module, a baseline path query processor should be developed solely on the storage system in case the index is not available. The index needs to support updates, concurrency control and recovery. The index processor should take into account both tree structures and values, since both structure and value could be the most selective constraints for some workloads.

At the query optimization side, a cost model for each of the physical operators is necessary for a query optimizer to choose the optimal execution plan. The formulae in cost models usually consist of cardinalities of subqueries, therefore, accurate cardinality estimations are crucial. Cardinality estimation should be based only on *a priori* knowledge,

i.e., synopses and statistics that are constructed and collected before queries are processed. There are also many desirable features for the synopsis: robustness, ease-of-construction, accuracy, adaptiveness to memory budget, and efficiency of estimation. Sometimes, these features may interact with each other, so we also need to trade off between them and design a synopsis suitable for practical use.

1.3 Motivation and Contributions

Significant research has been carried out on XML data management, particularly in the XML storage systems, query processing, and query optimization.

There are basically two approaches for storing XML documents: the extended relational approach [49, 122, 58, 137, 139, 70, 72] that converts XML documents to relational tables; and the native approach which XML documents are stored in a special purpose data structure [86, 90, 23]. Several techniques in the extended relational approach are *schema-aware* in that the conversion of XML documents to relational tables is dependent on the schema of the XML documents [122, 49]. Other techniques are *schemaless* in that they store XML documents in relational tables regardless of their schemata [58, 137, 139, 79, 95, 28, 48, 70]. In this approach, XML documents are usually abstracted as node labeled trees or graphs. General encoding techniques are developed to convert the labeled trees or graphs to relational tables. For example, if the XML documents are modeled as labeled graphs, the edge-oriented approach [58] can be used to store every edge as a tuple consisting of the source and target nodes as columns. If, on the other hand, the XML documents are modeled as labeled trees, the node-oriented approach [139, 70] can be used to store every node as a tuple consisting of the orders during some tree traversal (preorder or postorder). Both of these techniques, however, have some problems: the edge-oriented technique is inefficient in evaluating path expressions containing *//*-axes, because recursive operators are needed on the edge table; and the node-oriented technique is inefficient in update since inserting a new element may change the encodings of a large number of elements.

On the other hand, techniques in the native approach [86, 90, 23] are designed for special purpose storage systems that balance query performance and updatability. The storage system can be designed so that inserting or deleting an element does not affect the

global structure. As an example, Natix [86] partitions large XML trees into small subtrees which can fit into a disk page. Inserting a node usually only affects the subtree in which the node is inserted. However, native storage systems may not be efficient in answering certain types of queries (e.g., `//bib//author`) since they require at least one scan of the whole tree (introduced later). The extended relational storage, on the other hand, may be more efficient due to the special properties of the node encodings. Therefore, a storage system that balances the evaluation and update costs still remains a challenge.

Processing of path queries can also be classified into two categories: join-based approach [139, 15, 28, 66, 71] and navigational approach [21, 84, 90, 27]. As mentioned above, storage systems and query processing techniques are closely related in that the join-based processing techniques are usually based on extended relational storage systems and the navigational approach is based on native storage systems. All techniques in the join-based approach are based on the same idea: each location step in the expression is associated with an input list of elements whose names match with the name test of the step. Two lists of adjacent location steps are joined based on their structural relationships. The differences between different techniques are in their join algorithms, which take into account the special properties of the relational encoding of XML trees. For example, the Multi-Predicates Merge Join (MPMGJN) [139] modifies the merge join algorithm to reduce unnecessary comparisons during the backtracking in the merge process. The stack-based structural join [15] eliminates backtracking in the MPMGJN algorithm by keeping the ancestors in a stack. The holistic twig join [28] further improves the binary stack-based structural join to allow multiple input lists, which eliminates the needs to store temporary intermediate results.

The navigational processing techniques, built on top of the native storage systems, match the pattern tree by traversing the XML tree. Some navigational techniques (e.g., [27]) are query-driven in that each location step in the path expressions is translated into an algebraic operator which performs the navigation. A data-driven navigational approach (e.g., [21, 84, 90]) builds an automaton for a path expression and executes the automaton by navigating the XML tree. Techniques in the data-driven approach guarantee worst case I/O complexity: depending on the expressiveness of the query that can be handled, some techniques [21, 84] require only one scan of the data, and the others [90] require two scans.

Both the join-based and navigational approaches have advantages and disadvantages. The join-based approach, while efficient in evaluating expressions having `//`-axes, may not be as efficient as the navigational approach in answering expressions only having `/`-axes. A specific example is `/*/*`, where all children of the root are returned. As mentioned earlier, each name test (`*`) is associated with an input list, both of which contain all nodes in the XML document (since all element names match with a wildcard). Therefore, the I/O cost of the join-based approach is $2n$, where n is the number of elements. This cost is much higher than the cost of the navigational operator, which only traverses the root and its children. On the other hand, the navigational approach may not be as efficient as the join-based approach for a query such as `//book//author`, since the join-based approach only needs to read those elements whose names are `book` or `author`, but the navigational approach needs to traversal all elements in the tree. Therefore, a technique that combines the best of both approaches is needed.

In addition to the join-based and navigational approaches, many XML indexes [28, 81, 82, 20, 129, 116, 128, 64, 99, 87, 131] are proposed to process path expressions efficiently. Some of the indexing techniques are proposed to expedite the execution of existing join-based or navigational approaches. Examples include the XB-tree [28] and XR-tree [81] for the holistic twig joins. Both techniques extend B^+ trees and build indexes on the input lists to skip unnecessary comparisons in the join. Since these are special purpose indexes that are designed for a particular baseline operator, their application is quite limited. Another line of research focuses on string-based indexes [129, 138, 116, 128]. The basic idea is to convert the XML trees as well as the pattern trees into strings and reduce the tree pattern matching problem to string pattern matching. Although their string conversion methods are different, one common problem lies in this approach: since the pattern tree is unordered (no order between predicates), when the pattern tree is converted to a string, an arbitrary order is forced on the siblings. Therefore, the index evaluation algorithm may lose answers (false-negatives). Their solution is to enumerate all possible strings by reordering the siblings in the query tree and pose an index query for each of the string. This, however, will result in an exponential number of query strings and greatly deteriorate the overall query performance.

More recent XML indexing techniques focus on the structural similarity based in-

dexes [99, 64, 87, 131, 88, 33, 75]. Techniques in this approach group XML tree nodes by their structural similarity. Although different indexes may be based on different notions of similarity, they are all based on the same idea: similar tree nodes are clustered into equivalence classes (or *index nodes*), which are connected to form a tree or graph. One common problem of these techniques is that although structural indexes are reasonably small for regular data sets, they could grow very large for structure-rich data. The index operator, which is developed to traverse the index tree or graph, is therefore inefficient in these data sets. Various techniques are developed to cope with this situation: e.g., materializing the index on disk [131], or limiting the similarity definition by tree depth to trade off between the covered query set and the space requirement (e.g., $A(k)$ -index [88], $D(k)$ -index [33] and $M(k)$ - and $M(k)^*$ -index [75]). However, index evaluation still requires complex pattern matching on the whole graph. Another problem with these techniques is that these indexes omitted values. In many cases, value constraints are more selective than structural constraints. Therefore, pure structural indexes do not fully exploit the constraints in the query. Therefore, a unified structural and value indexing technique that avoids searching the entire index remains a challenge.

Finally, a cost-based optimizer is crucial to the query performance. The accuracy of cost estimation is usually dependent on the cardinality estimation. Therefore, many techniques have been proposed to estimate cardinality of path expressions [64, 13, 37, 59, 132, 110, 111, 14, 113, 112, 130]. All of these techniques first summarize an XML tree (corresponding to a document) into a small synopsis that contains structural information and statistics. The synopsis is usually stored in the database catalog and is used as the basis for estimating cardinality. Depending on how much information is reserved, different synopses cover different types of queries. DataGuide [64] is first designed for semistructured data (graph-based OEM data model [109]). It records all distinct paths from a data set and compresses them into a compact graph. Path tree [13] is designed for XML trees and also captures all distinct paths. Furthermore, path trees can be further decompressed if the resulting synopsis is too large. Markov tables [13], on the other hand, do not capture the full paths but sub-paths under a certain length limit. Selectivity of longer paths are calculated using fragments of sub-paths similar to the Markov process. All the above synopsis structures only support simple linear path queries that may or may not contain $//$ -axes.

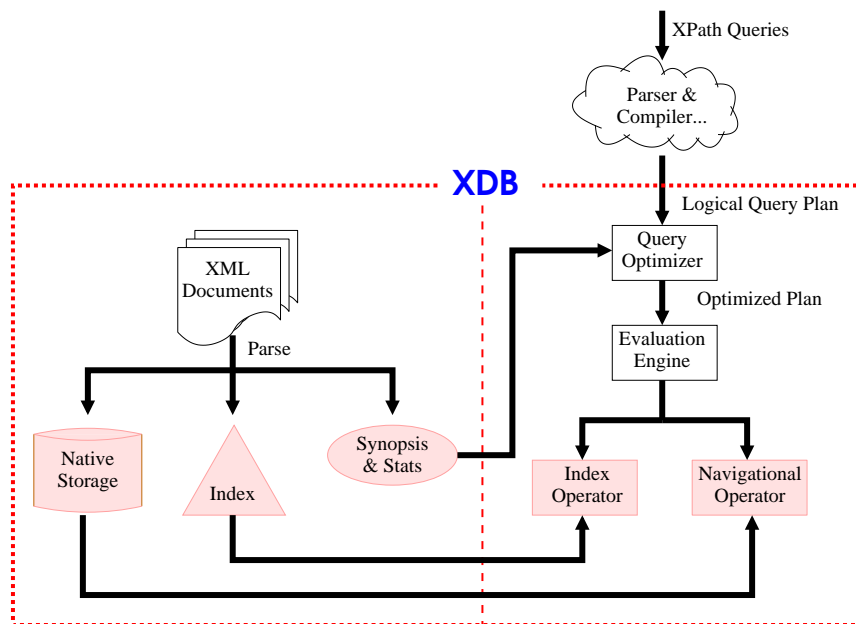


Figure 1.5: The modules and workflow of XDB data management

Structural similarity-based synopsis techniques (XSketch [110] and TreeSketch [112]) are proposed to support branching path queries. These techniques are very similar to the structural similarity-based indexing techniques: clustering structurally similar nodes into equivalence classes. An extra step is needed for the synopsis: summarize the similarity graph under some memory budget. Different synopses develop different heuristics for the summarization since obtaining the optimal summarization is NP-hard. Examples include a bottom-up heuristic that expand from a label-split graph [110] and a top-down heuristic that summarizes a count-stable graph [112]. A common problem of these heuristics is that the synopsis construction (expansion or summarization) time is still prohibitive for structure-rich data. Therefore, a synopsis technique that balances between the construction time and estimation accuracy is needed.

In this thesis, all of the above problems are studied and the following solutions are proposed:

1. A succinct storage scheme is proposed to store a large number of arbitrarily complex XML documents. The storage system supports efficient evaluation of path expressions as well as update.
2. A hybrid join-based and navigational based path query processing strategy is proposed. This hybrid evaluation technique takes advantages of both the join-based and navigational approaches.
3. A baseline navigational operator, called NOK, is developed based on the proposed storage system. By focusing on a special fragment of path expressions, the proposed navigational operator is simpler and more efficient than the previous navigational operators.
4. A feature-based index, called FIX, is proposed to index the numerical features of subtrees in the data. FIX avoids searching the whole graph and provides significant pruning power.
5. A synopsis, called XSEED, is proposed to summarize the XML documents into small graphs for cardinality estimation. The XSEED synopsis can be efficiently constructed, and it provides accurate cardinality estimation.

The proposed techniques have been implemented in a prototype system called XDB (Figure 1.5). The fully developed modules are the shaded boxes at the bottom level. The XDB system follows the two-phase processing model: (1) in the preprocessing phase, XML documents are stored in native storage format, a FIX index could be built based on the storage, and the XSEED synopsis and statistics about the documents are also stored in the system catalog; (2) in the query processing phase, the query optimizer consults the synopsis previously stored in the catalog to estimate the cost of a set of plans and to select the best one. The evaluation engine takes the selected plan and invokes the appropriate physical operators: a FIX index operator or a NOK navigational operator.

In summary, the contributions of this thesis are as follows:

- A native storage system for XML documents is proposed to support efficient evaluation of path queries and update of the XML documents. The proposed storage

system linearizes the tree structure into strings that can be decomposed into disk pages. Simple statistics are kept in page headers to facilitate I/O-efficient navigation.

- A baseline navigational evaluation operator, called Next-of-Kin (NoK) operator, is designed to evaluate a fragment of path expressions in one pass of the native storage. A hybrid approach is proposed to evaluate a general path expression: the path expression is decomposed into NOK expressions, which are evaluated using NoK processor. The intermediate results are structurally joined to produce the final results.
- A novel XML indexing technique, called FIX, is proposed to expedite the evaluation by extracting the distinctive features from the XML trees. Features are used as the index keys to a mature index such as B⁺ tree. For each incoming query, the features of the query tree are extracted and used as search keys to retrieve the candidate results. Experiments show that FIX provides great pruning power to both structures and values in the original data tree.
- An XML synopsis structure, called XSEED, is proposed to estimate the cardinality of path expressions. An XSEED synopsis can be constructed by compressing an XML document to a small kernel first, and then more information can be added to the synopsis. Comparing to previous approaches, XSEED results in more accurate cardinality estimation, is easier to construct and update, and can incorporate query feedback.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 covers the related work on storing and querying XML documents. Chapter 3 describes the hybrid evaluation strategy for a path expression, the XML native storage system and the NoK navigational operator based on it. Chapter 4 introduces the feature-based indexing technique FIX. Chapter 5 presents the XSEED synopsis structure and cardinality estimation. Chapter 6 concludes the thesis and discusses some topics for future study.

Chapter 2

Related Work

XML data management has attracted significant research interest from both the theoretical (e.g., [65, 12, 45, 100, 55, 52, 80, 16, 17, 61, 121, 67, 140, 19, 68, 91]) and the systems (e.g., [98, 63, 102, 79, 57, 1, 30, 23, 104, 96, 107]) points of view. Research on theoretical aspects of XML data management focuses on the decidability of typechecking, the validity of XML over a schema language (e.g., DTD [3], XML Schema [11] or a fragment of them), integrity constraints and normal forms for XML data, logical level query rewriting, and the time and space complexities of evaluation specific query languages (e.g., XPath [22], XQuery [24]). Excellent surveys on these topics are available [97, 127, 103, 29], therefore detailed discussion of theoretical aspects of XML data management is omitted in this chapter.

System aspects of XML data management concentrate on the storage system for XML documents, query processing techniques for XML queries, indexing techniques for XML documents, and query optimization techniques for XML queries. All of these topics are addressed in this thesis. The following sections survey existing techniques in each of these topics.

2.1 XML Storage

There are generally two approaches to storing XML documents: the extended relational approach and the native approach. In the extended relational approach, XML documents are

```
<!ELEMENT bib (article|book)*>
<!ELEMENT article (title, author*, published_in)>
<!ATTLIST article year CDATA #REQUIRED>
<!ELEMENT book (title, author*, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ATTLIST price currency CDATA #REQUIRED>
```

Figure 2.1: DTD for XML file in Figure 1.3

converted to relational tables and are stored in relational databases or in object repositories (e.g., Shore [31]). In the native approach, XML documents are stored using a specially designed native format. Each of these techniques has advantages and disadvantages, although both techniques are feasible for supporting XML queries.

In this section, existing storage techniques are introduced based on the example shown in Figure 1.3, whose DTD is given in Figure 2.1. The discussion focuses on a comparison of these approaches according to the following desirable features:

- S1: Is the system able to store arbitrarily complex XML documents in terms of width, height, and number of elements?
- S2: Is the system able to store documents with or without schema?
- S3: Does the system preserve orders in XML documents?
- S4: Does the system provide support for efficient query evaluation?
- S5: Does the system support efficient incremental update? In particular local updates¹ should not cause drastic changes to the whole storage.
- S6: Is the storage cost small compared to the original XML document?

The last requirement is included because XML documents are intended to be human-readable, and therefore they are usually verbose and contain redundant information. Note

¹Local updates are updates that only affect a small subset of the XML document. Particular examples are inserting or deleting a node in the XML tree.

that these features are mainly related to supporting query and update. Features related to access control, concurrency control, and recovery etc. are omitted since they are beyond the scope of this thesis.

2.1.1 Extended Relational Storage

In the extended relational approach, an XML document is shredded into pieces and these pieces are reassembled into relational tables [49, 122, 58, 137, 139, 70, 72]. One way of achieving this is to map the XML structure (DTD or XML Schema) to a corresponding relational schema. Another approach is to design a generic relational schema to which any XML document can be mapped. The former is known as *schema-aware* mapping [49, 122] and the latter *schemaless* mapping [58, 137, 139, 70]. Some related work (e.g., [25, 126]) are omitted from this chapter since they focus on some specific subproblems in the XML-to-relational conversion. For example, LegoDB [25] focuses on the selection of the optimal XML-to-relational schema mapping according to cost estimation; and [126] compares different ways of representing orders using relational encodings.

Schema-aware XML-to-Relational Mapping

Techniques that fall into this category require prior knowledge of the structure of the XML documents. The structure (element names and their parent-child relationships) can be discovered from the XML documents, or it can be provided by the associated schema (DTD or XML Schema). Thus, there may be different relational schemata for different sets of XML documents.

Shanmugasundaram et al. [122] propose to map XML documents to relational tables given a DTD. They first simplify the DTD to a special format and then perform the mapping from the simplified DTD to a relational schema. The mapping is based on the *DTD graph* (see, e.g., Figure 2.2) converted from the DTD. Regular unmarked edges in the DTD graph represent one-to-one parent-child relationship; while edges labeled with “*” represent one-to-many parent-child relationship.

Given a DTD graph, the relational mapping is carried out as follows: (1) a table is created for each vertex in the DTD graph; (2) all descendants reachable from *regular* edges

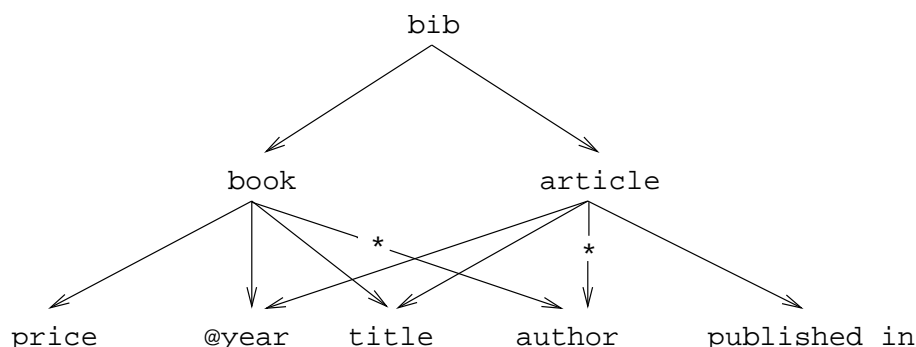


Figure 2.2: The DTD Graph for the DTD in Figure 2.1

are *inlined* into the table; (3) each element has a unique ID. Based on these rules, the mapping of the XML document in Figure 1.3 results in four tables as shown in Figure 2.3. The table `bib` is created for the root element only since all its children (`article` and `book`) are connected by “*” edges. Some vertices, e.g., `title` and `price`, have no corresponding tables since they are inlined to the `book` and `article` tables. The `author` table is created since it connects edges labeled with “*”. Each row in the table corresponds to an element in the XML document. The ID and PID columns correspond to the IDs for the element and its parent, respectively, representing the primary key and foreign key in the tables. Shanmugasundaram et al. also define other inlining techniques that try to reduce the number of tables and space overhead.

The advantages of this approach are that the XML-to-relational mapping is quite simple and easy to implement. It can handle arbitrarily complex XML documents. Although during DTD simplification the ordering of siblings is lost in the relational schema, the order can be preserved by adding a position column in the table and assigning each element an appropriate position in the siblings while the document is loaded. Inserting (or deleting) elements in the XML document can be translated into inserting (or deleting) tuples in the corresponding tables as long as the updates do not make the document inconsistent with the DTD. Therefore updating can also be performed quite efficiently. However, there are some disadvantages. Firstly, this technique does not support schemaless documents. Secondly, if the DTD evolves, the relational schema needs to be updated, which is expensive. Finally, as also pointed out by the authors, current relational technology

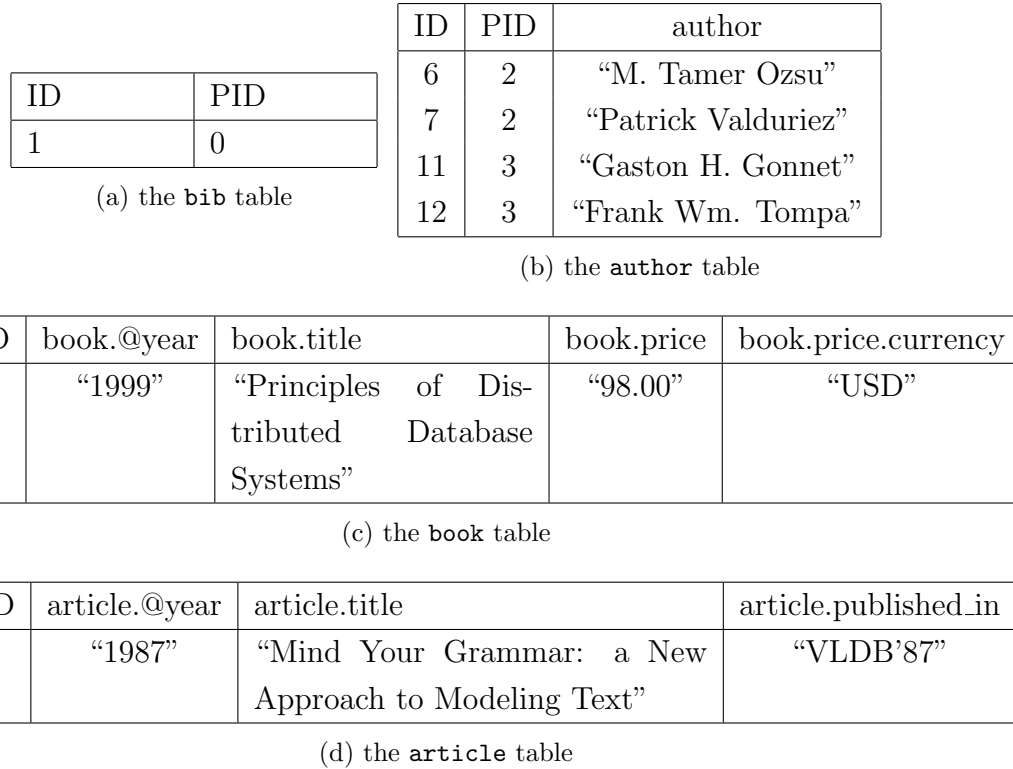


Figure 2.3: Relational tables mapped from Figure 1.3 using Shanmugasundaram et al. [122]

proves awkward or inefficient for some XML queries. For example, to answer the query `//bib//*[@year="1999"] [author="M. Tamer Ozsu"]`, one needs to join all the tables `bib`, `book`, `article`, and `author`. If there are more children for the `bib` vertex in the DTD graph, all corresponding tables must be joined.

Deutsch et al. [49] propose to deal with schemaless documents by mining the structure (which can be expressed by a fragment of DTD) from XML documents. The mapping from XML structure to relational schemata is defined in a declarative language called STORED. The basic idea of the mapping is as following: objects with different “types” are first identified. A type is similar to the *complex* element (which has child elements) definition in DTD (e.g., `article` in Figure 2.1). Each type is converted to a relational

ID	@year	title	author1	author2	price
1	"1999"	"Principles of Distributed Database Systems"	"M. Tamer Ozsu"	"Patrick Valduriez"	"98.00"

(a) the `bib.book` table

ID	@year	title	author1	author2	published_in
2	"1987"	"Mind Your Grammar: a New Approach to Modeling Text"	"Gaston H. Gonnet"	"Frank Wm. Tompa"	"VLDB'87"

(b) the `bib.article` table

Figure 2.4: Relational tables mapped from Figure 1.3 using STORED

table, where the subelements are mapped to columns. An object of a particular type is then translated into a tuple in the table. For example, the XML file in Figure 1.3 is translated into two tables `bib.book` and `bib.article` as shown in Figure 2.4.

Different from the previous mapping technique, `author` is not identified as a type, but is stored as column(s) in the table. The number of `author` columns is determined by the maximum number of authors in all `book` or `article` objects. Therefore, there are possibly many NULL values in the tables. The advantages of this mapping technique is that it can deal with schemaless documents as well as semi-structured graph data instances. However, since some structural information is coded in the relational schema, e.g., the table name `bib.book`, path query processing needs manipulation of the schema information as well. For example, answering path expression `/*/book` needs to find all tables with the name that has a postfix `".book"`. Another problem with the mapping is that updates to the XML document may cause the relational schema to change. For example, if another `author` element is inserted to the `book` element, the table `bib.book` needs to insert a new column `author3`.

Schemaless XML-to-Relational Mappings

Schemaless mappings do not require prior knowledge about DTD or XML structure. Techniques in this category design a generic relational schema for all XML documents regardless of whether or not they have a schema.

Florescu and Kossman [58] propose to store an XML tree into an *edge table*, where each node has an ID and name. In the edge table, the IDs of the parent and child nodes are recorded in one tuple, along with other information such as the type and reference of the possible text node associated with the child. In addition to the edge table, one table is created for each data type and they are referenced from the edge table. For example, the XML document in Figure 1.3 can be converted to an edge table, a V_{int} table, a V_{string} , and a V_{double} table as shown in Figure 2.5.

Each tuple in the Edge table represents an edge (either a tree edge or an IDREF edge). The **source** and **target** columns record the object ID of the source and target nodes. Specifically the document root has a special ID—0, and the value nodes (CDATA or PCDATA) have special value IDs v_i . The **name** and **ordinal** are the target's name and the order it appears in its siblings, respectively. The **flag** column indicates the type of the edge: **ref** indicates that the edge is a tree edge between two element nodes or an IDREF edge; other keywords indicate the data type of the target node if it is a value node. Each data type has a separate table (e.g, the V_{int} table) that records the value ID and the value.

The advantages of the edge table approach is that it can convert any XML document, with or without schema, to relational tables. IDREFs, which are usually omitted or need special treatment by many other storage systems, can be naturally stored in the edge table. Inserting or deleting an element can be performed efficiently by inserting or deleting edges in the edge table. The order of siblings is also preserved in the edge table; however, it is hard to determine the ordering of two arbitrary nodes that are not siblings. Furthermore, it is clear that simple path queries that contain only */*-axes can be evaluated using multiple joins on the same edge table: each */*-axis is translated into a self-join on the edge table. The deficiency of this approach, however, is the inability or inefficiency to answer queries containing *//*-axes in the path expressions using SQL only. For example, to answer query *//bib//author*, one needs to find all **author** elements as the descendant of **bib** element following paths of *any* length. Therefore, it is necessary to use the recursion

source	ordinal	name	flag	target
0	1	bib	ref	1
1	1	book	ref	2
1	2	article	ref	3
2	0	@year	ref	4
2	1	title	ref	5
2	2	author	ref	6
2	3	author	ref	7
2	4	price	ref	8
3	0	@year	ref	9
3	1	title	ref	10
3	2	author	ref	11
3	3	author	ref	12
3	4	published_in	ref	13
4	1	TEXT	int	v_1
5	1	TEXT	string	v_2
6	1	TEXT	string	v_3
7	1	TEXT	string	v_4
8	0	@currency	ref	14
8	1	TEXT	double	v_5
9	1	TEXT	int	v_6
10	1	TEXT	string	v_7
11	1	TEXT	string	v_8
12	1	TEXT	string	v_9
13	1	TEXT	string	v_{10}
14	1	TEXT	string	v_{11}

(a) the Edge table

vid	value
v_1	1999
v_6	1987

vid	value
v_5	98.00

(b) the V_{int} table (c) the V_{double} table

Figure 2.5: Relational tables mapped from Figure 1.3 using the edge-based approach

vid	value
v_2	“Principles of Distributed Database Systems”
v_3	“M. Tamer Ozsu”
v_4	“Patrick Valduriez”
v_7	“Mind Your Grammars: a New Approach to Modeling Text”
v_8	“Gaston H. Gonnet”
v_9	“Frank Wm. Tompa”
v_{10}	“VLDB’87”
v_{11}	“USD”

(d) the V_{string} table

Figure 2.5: Relational tables mapped from Figure 1.3 using the edge-based approach (continued)

mechanism defined in SQL 1999 standard which is not well supported in many RDBMS implementations.

Interval encoding is a widely adopted XML-to-relational mapping technique [139, 79, 95, 28, 48]. In interval encoding, each tree node is assigned three numbers: **begin**, **end**, and **level**, which can be obtained by depth first traversal of the XML tree. During the traversal, a counter is incremented each time the visit of a tree node is started and finished. Each tree node is visited twice, and the counter values associated with the two visits are assigned to **begin** and **end**. The **level** of a node is the depth from the root node. Figure 2.6 shows the interval encoding of the XML file in Figure 1.3.

Along with the document ID, **begin** and **end** uniquely identify the location of a node in an XML tree. The **level** is kept for efficient checking of the parent-child relationship. These four integers are sufficient for testing any structural relationship (parent-child, ancestor-descendant, etc.) between two nodes by *containment conditions*. For example, a node x is a descendant of node y if and only if: (1) $x.docID = y.docID$; and (2) $y.begin < x.begin < x.end < y.end$. Testing parent-child relationship only needs one additional condition $x.level = y.level + 1$.

The advantages of this mapping technique are that any XML documents can be trans-

docID	name	type	begin	end	level
1	bib	element	1	39	1
1	book	element	2	21	2
1	year	attribute	3	5	3
1	“1999”	int	4	4	4
1	title	element	6	8	3
1	“Principles of Distributed Database Systems”	string	7	7	4
1	author	element	9	11	3
1	“M. Tamer Ozsu”	string	10	10	4
1	author	element	12	14	3
1	“Patrick Valduriez”	string	13	13	4
1	price	element	15	20	3
1	currency	attribute	16	18	4
1	“USD”	string	17	17	5
1	“98.00”	double	19	19	4
1	article	element	22	38	3
1	year	attribute	23	25	3
1	“1995”	int	24	24	4
1	title	element	26	38	3
1	“Mind Your Grammars: a New Approach to Modeling Text”	string	27	27	4
1	author	element	29	31	3
1	“Gaston H. Gonnet”	string	30	30	4
1	author	element	32	34	3
1	“Frank Wm. Tompa”	string	33	33	4
1	published_in	element	35	37	3
1	“VLDB’87”	string	36	36	4

Figure 2.6: Interval Encoding for XML file in Figure 1.3

lated into interval encoding (except the integration of IDREFs). The interval encoding also preserves the document order since it coincides with the order on the `begin` column. Given the interval encoding of any two nodes, checking all the structural constraints can be answered in constant time. However, updating could be expensive. For example, inserting a new element will change the `begin` and `end` encodings of all elements after it in document order. Although more update-friendly interval encoding techniques are proposed (e.g., [35]), the updating cost is still $O(\log n)$, where n is the total number of elements. For a large document consisting of millions of elements, the cost is still expensive.

Yoshikawa et al. [137] proposed XRel, a path-based approach to storing XML documents in relational databases. XRel maintains four tables for any XML document: the element table, the attribute table, the text table, and the path table. The first three tables contain tuples corresponding to element, attribute and text nodes in an XML tree. The `start` (or `end`) column in these tables represents the start (or end) position (byte offset from the beginning of the document) of the corresponding element, attribute, or text, respectively. The path table contains distinct paths in the XML tree and each path is assigned a unique ID. Steps in the path are delimited by `'#/'` rather than `'/'`. The reason is to be able to evaluate queries with `//`-axes using the `LIKE` predicate in SQL (introduced later). Since many paths have common prefixes, the path table contains significant redundant information. Each tuple in the element, attribute, and text tables contains a foreign key to the path table indicating the path from root to this node. If there are multiple elements that share the same rooted paths, their `start` and `end` columns will tell the difference and their relative position. In order to expedite evaluation of path expressions containing position predicates (e.g., `/bib/book/author[2]`), the element table also keeps two extra columns—`index` and `reindex`—which indicate the element's relative position among its siblings with the same name. As an example, Figure 2.7 shows the XRel table converted from the XML file in Figure 1.3.

The relational schema of the path-based technique is similar to interval encoding except that the rooted paths of each element is kept in the path table. In order to answer queries such as `/bib//author`, one needs to find in the path table all tuples with `author` as the element name and where there is a `bib` element in the rooted path. This is implemented by the `LIKE` predicate `'#/bib#%/author'`, where `'/'` is replaced by `'#/'` and `'//'` by `'#%/'`. The

docID	pathID	start	end	index	reindex	NodeID
1	1	0	428	1	1	1
1	2	5	184	1	1	2
1	4	23	47	1	1	4
1	5	55	80	1	1	5
1	5	89	112	2	1	6
1	6	149	177	1	1	7
1	8	191	414	1	1	9
1	10	212	260	1	1	11
1	11	268	299	1	1	12
1	11	308	337	2	1	13
1	12	379	402	1	1	14

(a) The Element table

docID	pathID	start	end	value	NodeID
1	3	6	6	"1999"	3
1	7	150	150	"USD"	8
1	9	192	192	"1987"	10

(b) The Attribute table

docID	pathID	start	end	value	NodeID
1	4	30	46	"Principles of Distributed Database Systems"	15
1	5	63	79	"M. Tamer Ozsu"	16
1	5	97	111	"Patrick Valduriez"	17
1	6	170	176	"98.00"	18
1	10	219	259	"Mind Your Grammar: a New Approach to Modeling Text"	19
1	11	276	298	"Gaston H. Gonnet"	20
1	11	316	336	"Frank Wm. Tompa"	21
1	12	393	401	"VLDB'87"	22

(c) The Text table

Figure 2.7: Relational tables mapped from Figure 1.3 by XRel

pathID	pathexp
1	#/bib
2	#/bib#/book
3	#/bib#/book#/@year
4	#/bib#/book#/title
5	#/bib#/book#/author
6	#/bib#/book#/price
7	#/bib#/book#/price#/@currency
8	#/bib#/article
9	#/bib#/article#/@year
10	#/bib#/article#/title
11	#/bib#/article#/author
12	#/bib#/article#/published_in

(d) The Path table

Figure 2.7: Relational tables mapped from Figure 1.3 by XRel (continued)

regular expression ‘#%/’ is able to express the **descendant-or-self** semantics of // -axis, where ‘//’ expresses **descendant** semantics. Answering a branching path expression is much more complex. For example, the path expression `/article[author="M. Tamer Ozsu"]//title` is translated into three string pattern matching ‘#/article’, ‘#/article#/@year’, and ‘#/article#%/title’ and nine joins between element tables, path tables, and text tables:

```

SELECT e2.docID, e2.start, e2.end
FROM Path p1, Path p2, Path p3,
     Element e1, Element e2,
     Text t3
WHERE   p1.pathexp LIKE '#/article'
        AND p2.pathexp LIKE '#/article#%/title'
        AND p3.pathexp LIKE '#/article#/author'
        AND e1.pathID = p1.pathID
        AND e2.pathID = p2.pathID

```

```

AND t3.pathID = p3.pathID
AND e1.start < e2.start
AND e1.end > e2.end
AND e1.docID = e2.docID
AND e1.start < t3.start
AND e1.end > t3.end
AND e1.docID = t3.docID
AND t3.value = 'M. Tamer Ozsu'
ORDER BY e2.docID, e2.start, e2.end

```

This complex SQL statement is hard to optimize. Another disadvantage of this path-based technique is that updating the XML document is even more expensive than the interval encoding because inserting an element may cause all tables to be updated (`start`, `end`, `index`, `reindex`, and `pathexp` may be changed).

Grust [70] has developed another encoding scheme, *XPath Accelerator*, using preorder and postorder traversals to identify any node in a tree. The relational schema includes five columns: `pre`, `post`, `par`, `att`, `tag`, where `pre` and `post` are the preorder and postorder of the node, respectively; `par` represents the parent's preorder; `att` is a boolean value indicating whether the node is an attribute; and `tag` is the element/attribute tag name. Grust et al. [72] extends this schema to support other kinds of nodes (text, PI, etc.) by changing the boolean `att` column to a categorical `kind` column. The ancestor-descendant relationship can be easily tested using the `pre` and `post` values: v' is a descendant of v if and only if $\text{prev}(v) < \text{prev}(v') \wedge \text{post}(v') < \text{post}(v)$. The `par` value can be used to test the parent-child and sibling relationships. Figure 2.8 shows the table for the XML document in Figure 1.3.

This storage technique has similar characteristics to interval encoding, e.g., inefficiency in updating. However, it has an advantage that it can exploit an efficient spatial index on the `pre-post` plane to evaluate path expressions.

2.1.2 Native Storage

Native XML storage techniques treat XML trees as first class citizens and develop special purpose storage schemes without relying on the existence of an underlying database system.

pre	post	par	att	tag
1	14	0	F	bib
2	7	1	F	book
3	1	2	T	year
4	2	2	F	title
5	3	2	F	author
6	4	2	F	author
7	6	2	F	price
8	5	7	T	currency
9	13	1	F	article
10	8	9	T	year
11	9	9	F	title
12	10	9	F	author
13	11	9	F	author
14	12	9	F	published_in

Figure 2.8: The XPath Accelerator table

Kanne and Moerkotte propose the Natix storage system [86, 57] that partitions a large XML tree into subtrees, each of which is stored in a *record*. The partition is designed such that the record is small enough to fit into a disk page. By introducing *proxy* nodes and *aggregate* nodes, it is possible to connect subtrees in different records, which enables the navigation of in the original XML tree. Figure 2.9 depicts a possible record partition of the XML tree in Figure 1.3. In this figure, the XML tree is partitioned into three records R1, R2, and R3. *p1* and *p2* are proxy nodes that point to different records. *h1* and *h2* are aggregate nodes that are inner nodes of the tree and are used to connect subtrees in different records.

Natix clusters tree nodes by their structural locality. This may significantly reduce the I/O cost while navigating the tree. Updating is relatively easy since insertions and deletions are usually local operations to a record. Therefore, the update cost mainly depends on the size of the subtree that fits into one record. When a record overflows a page, what needs to

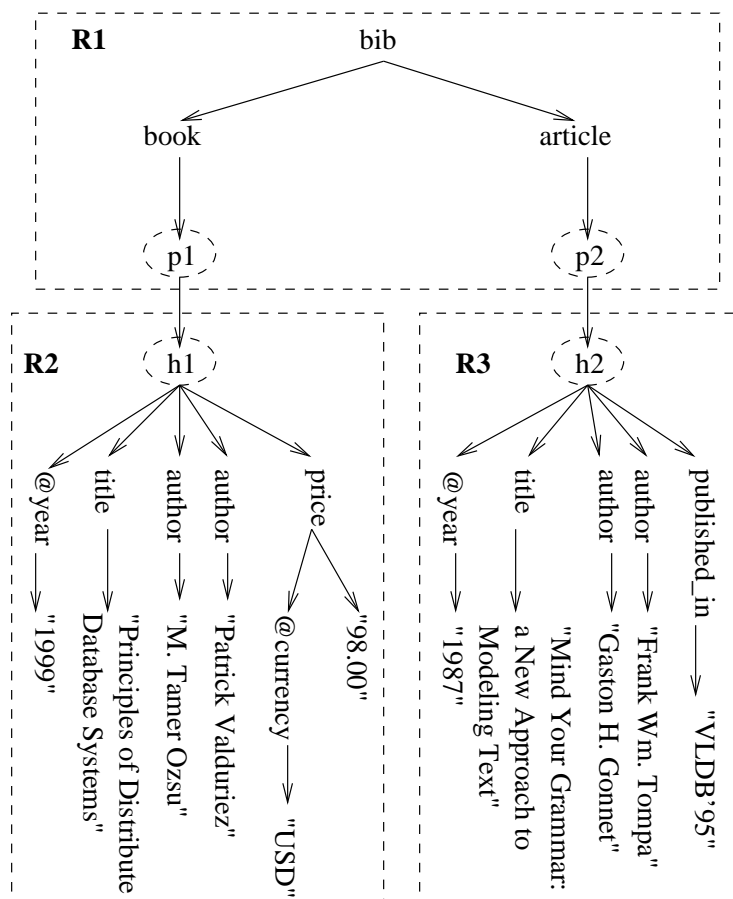
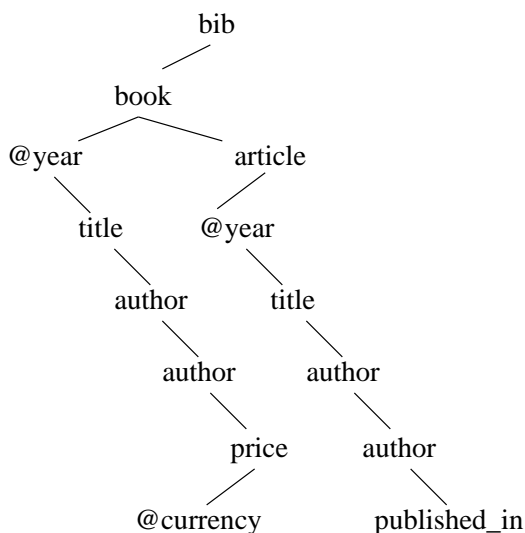


Figure 2.9: A possible Natix record decomposition of the XML tree in Figure 1.3

be done is simply splitting the record into two pages. Deletions may result in merging two adjacent records, which can also be performed efficiently. In general, this approach is more flexible than the interval encoding or preorder-postorder encoding in handling frequent updates.

Koch proposes the *Arb* storage model [90] to store a tree in its binary tree representation on disk. It is well known that any ordered tree T can be translated into a binary tree B by translating the first child of a node in T to the left child of the corresponding node in B and the following sibling in T to the right child in B . Figure 2.10a shows the binary tree representation of elements and attributes in Figure 1.3. Arb stores tree nodes in document



(a)

bib	book	@year	title	author	author	price	@currency	article	@year	title	author	author	published_in										
1	0	1	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0

(b)

Figure 2.10: Arb Storage of Figure 1.3, (a) logical binary tree representation; (b) physical storage

order. Each node uses two bits to represent whether it has a left child and/or a right child. As an example, the Arb physical storage for Figure 1.3 is shown in Figure 2.10b.

Arb efficiently supports updates since inserting (or deleting) a node is equivalent to inserting (or deleting) an item in the ordered list. The only extra work is to update the left or right child bits of the parent of the updated node. Arb also supports efficient top-down and bottom-up navigation in the tree. However, navigating between siblings, e.g., **following-sibling**, is not efficient, because jumping to the following sibling of a non-leaf node requires traversal of the whole subtree. Supporting efficient sibling level navigation is crucial to some navigational evaluation algorithms introduced in Section 2.2.

Commercial DBMS vendors have also developed storage systems for supporting XML data [23, 60, 108, 92]. However due to the lack of sufficient information on how the data are stored, updated, and queried, they are not included in this chapter.

Storage System	category	S1	S2	S3	S4	S5	S6
Shanmugasundaram et al. [122]	relational	yes	no	no	yes	no	yes
STORED	relational	yes	yes	no	no	no	no
edge-table	relational	yes	yes	partly	no	yes	moderate
interval-encoding	relational	yes	yes	yes	yes	no	moderate
XRel	relational	yes	yes	yes	no	no	no
XPath-Accelerator	relational	yes	yes	yes	yes	no	moderate
Natix	native	yes	yes	yes	yes	yes	moderate
Arb	native	yes	yes	yes	partly	yes	yes
Schkolnick [118]	native	yes	no	yes	yes	yes	moderate

Table 2.1: Feature comparison between different techniques (Recall that S1=robustness; S2=schema-independence; S3=order-preserving; S4=supporting efficient evaluation; S5=supporting efficient update; S6=succinctness)

The problem of efficiently storing hierarchical data has been studied in the context of hierarchical databases in 1970’s. Schkolnick [118] proposed a technique to store trees with types in a paged I/O model. At the logical level, Schkolnick’s technique uses a parenthesis representation for *type trees*, which is analogous to the DTD or XML Schema in the context of XML databases. For example, an example type tree is represented as: $T = (\text{NAME}, (\text{ADDRESS}), (\text{PAYROLL}), (\text{SKILL}, (\text{EXPERIENCE}), (\text{EDUCATION})))$. Each node in the type tree is a type, and if node i is the parent of node j then type i is a super type of type j . For each type tree node, there are instances associated with the type. Schkolnick proposed a clustering algorithm that is based on two principles: (1) all instances of the same type are clustered together, and (2) type tree is partitioned into subtrees and all instances of the sub type trees can be clustered together. Given a type tree, there are 2^{n-1} different ways to partition it. Therefore, the objective is to find the optimal partition. The optimality of partitioning is determined by the expected page I/Os when performing a *hierarchical scan*, which allows three types of transitions: parent-child transition (PCT), twig transition (TT), and child-parent transition (CPT). Given the frequencies of these transitions, the paper proposes a base-line algorithm in $O(2^n)$ to find the optimal parti-

tion. For a special case where every node in the type tree has a fixed k degree, the paper then proposes a more efficient algorithm in $O(nk\alpha^k)$, where n is the number of nodes in the type tree, k is the fanout for each node, and α is some constant. Compared to the other native storage schemes, Schkolnick's partitioning algorithm is schema-dependent. The optimality of the partitioning is dependent on the frequencies of the three types of transitions, which may be unavailable in some real world situations. Moreover, in the general case, the clustering algorithm is still exponential in the size of the schema.

In summary, Table 2.1 presents the feature comparison between the existing storage systems, where S1 to S6 are the six desirable features listed at the beginning of Section 2.1.

2.2 Path Query Processing

Processing of path queries can mainly be classified into two categories: *join-based approach* [137, 139, 15, 28, 48, 71] based on extended relational storage, and *navigational approach* [90, 21, 84] based on the native storage. A hybrid approach [74] is also developed to take advantages of both join-based and navigational approaches.

The rest of the section focuses on the algorithmic aspects of the query processing techniques, rather than comparing techniques based on different criteria as in the previous section. The reason is that there is only one primary interest of the processing techniques: the runtime efficiency of the algorithms.

2.2.1 Join-based Approach

Techniques in this category [137, 139, 15, 28, 48, 71] focus on designing specific relational join operators that take into account the special properties of the relational schema of XML trees.

Most of the join-based techniques are based on the same processing strategy: given a path expression, every NameTest is associated with a list of input elements with the same name that match with the NameTest. The axis between two NameTests are translated into a *structural join* operator that returns pairs of elements that satisfy the structural relationship. For example, the path expression `//article//author` can be translated into a join between two lists of `article` and `author` elements $[\text{article}_1, \text{article}_2, \dots, \text{article}_m]$

and $[\text{author}_1, \text{author}_2, \dots, \text{author}_n]$ respectively. Pairs of $(\text{article}_i, \text{author}_j)$ are returned if they have a **ancestor-descendant** relationship. Depending on different XML-to-relational mapping, different techniques may use different join conditions on the relational encodings.

As introduced in Section 2.1, XRel [137] translates a path expressions into a SQL statement, which consists of a conjunction of **LIKE** predicates and comparison ($=, <, >$) predicates. How to evaluate this statement is, therefore, left to the relational database systems. The Dynamic Interval (DI) system [48] is also built on pure relational database systems without relying on specific designed operators. DI first translates a path expression into a nested FLWOR expression, which is then compiled into an execution plan. Since DI uses interval encoding to represent documents, the translated FLWOR expression is compiled into joins on the **begin**, **end**, and **level** columns with the following condition for `//article//author`:

```

        article.start < author.start
    AND article.end   > author.end

```

Two physical join operators, nested-loop join and sort-merge join, are adopted from the relational database systems. Experiments show that sort-merge join outperforms the nested-loop join significantly.

The Multi-Predicate Merge Join (MPMGJN) algorithm [139], also based on interval encoding, modifies the merge join algorithm with the objective of reducing unnecessary comparisons during the merge process. The difference from the DI's sort-merge join is that MPMGJN considers multiple documents, therefore, an equi-predicate between **docIDs** are added to the join condition for `//article//author`:

```

        article.docID = author.docID
    AND article.start < author.start
    AND article.end   > author.end

```

When the input lists are sorted by **docID** and use **start** to break ties, the naïve merge join is not efficient since it will compare all pairs of **article** and **author** elements from the same document (because their **docIDs** are equal). The key observation of the MPMGJN algorithm is that if an item article_i in the ancestor input list satisfies the structural

relationship with the items in the range $[\mathbf{author}_j, \mathbf{author}_{j+k}]$, then, when attempting to join the next item $\mathbf{article}_{i+1}$, the algorithm only needs to compare it with \mathbf{author}_j and onward because all items before \mathbf{author}_j are guaranteed not to be the descendants of $\mathbf{article}_i$ thus $\mathbf{article}_{i+1}$.

The stack-based structural join [15] algorithm is based on a similar observation, but improves the performance of MPMGJN in the case of recursive data. For example, consider the following XML document:

```

<a>
  <a>
    <b> ... </b>
    <b> ... </b>
  </a>
</a>

```

Denoting the first and second occurrences of elements \mathbf{a} and \mathbf{b} as \mathbf{a}_1 , \mathbf{a}_2 , \mathbf{b}_1 , and \mathbf{b}_2 , respectively, the MPMGJN algorithm will compare the pairs $(\mathbf{a}_1, \mathbf{b}_1)$, $(\mathbf{a}_1, \mathbf{b}_2)$, $(\mathbf{a}_2, \mathbf{b}_1)$, and $(\mathbf{a}_2, \mathbf{b}_2)$. The stack-based structural join algorithm optimizes this case by exploiting the transitivity of the descendant relationship: the fact that \mathbf{b}_1 and \mathbf{b}_2 being descendants of \mathbf{a}_2 and the fact that \mathbf{a}_2 being descendant of \mathbf{a}_1 imply \mathbf{b}_1 and \mathbf{b}_2 being descendants of \mathbf{a}_1 .

To avoid backtracking, the stack-based structural join algorithm keeps a stack of elements \mathbf{a}_i such that any element on the stack is an ancestor of all elements on top of it. Therefore, to answer $//\mathbf{a}//\mathbf{b}$, \mathbf{a}_1 and \mathbf{a}_2 are first pushed onto the ancestor stack in that order. Then \mathbf{b}_1 and \mathbf{b}_2 only need to compare with the top of the stack \mathbf{a}_2 . Any successful join to the top of the stack indicates successful joins to all nodes in the stack. With the assumption that XML trees are usually shallow, the memory requirement for maintaining the stacks is reasonably small.

The holistic twig join [28] generalizes the stack-based structural join to allow more than two inputs. The objective is to reduce the number of intermediate results from the binary structural joins that do not contribute to the final results. The holistic twig join algorithm maintains a stack of elements for each input list and output a result only if the top elements of all stacks satisfy all the structural constraints. Therefore, there is no need to output and deal with intermediate results. The memory requirement of holistic twig join is even

larger than the binary stack-based structural join, but the former significantly improves the query performance over binary structural join and MPMGJN [28]. In fact, the holistic twig join is proven to be I/O optimal for queries containing only // -axes [28].

2.2.2 Navigational Approach

Path query processing techniques in the navigational approach are usually based on native storage systems. Basically there are two types of navigational approaches: query-driven and data-driven. In the query-driven navigational operators (e.g., Natix [27]), each location step in the path expression is translated into a transition from one set of XML tree nodes to another set. In the data-driven operators (e.g., XNav [84]), the query is translated into an automaton and the data tree is traversed according to the current state of the automaton. The query-driven approach is easier to implement, but it may need multiple scans of the input. On the other hand, the data-driven approach only needs one scan of the data, but its implementation is much complex.

The Natix query processing engine translates a path query into a native XML algebraic expression [27]. Each location step in the path expression is translated into an *Unnest-Map* operator that effectively replaces an input list with an output list satisfying structural relationship specified by the axis. A path expression is then translated into a chain of Unnest-Map operators connected by *dependency joins (d-joins)*. Since each Unnest-Map operator is translated into a physical operator that follows the links in the Natix storage system, navigation can be determined statically by the query. Kanne et al. [85] optimize the Natix I/O performance by considering multiple Unnest-Map operators as a whole and schedule I/O accordingly (introduced in more detail in Section 2.4). New physical operators optimize I/O for both asynchronous and sequential access patterns by dynamically determining the navigation in the data storage.

XNav [21, 84] is a navigational processing technique based on finite state automata. Since XNav is proposed in the context of streaming XML processing², the algorithm requires only one pass of the input XML data, possibly skipping some tree nodes. Therefore, it is analogous to the sequential scan operator in relational database systems. The difference is that XNav has a more complex data access pattern. The automaton can be

²There is nothing to prevent XNav from being used in the stored-and-query XML processing context.

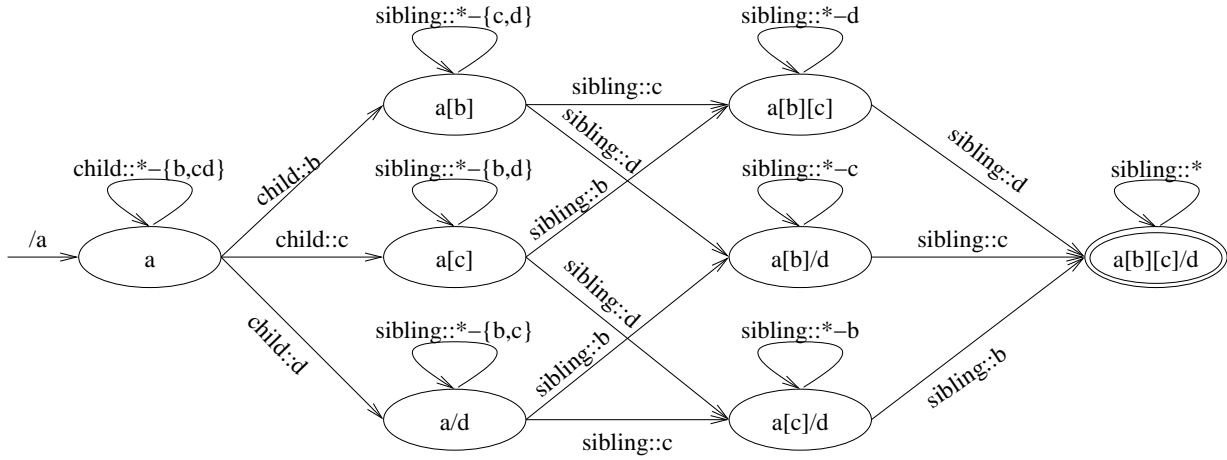


Figure 2.11: A static automaton for path expression $/a[b][c]/d$

constructed from the path expression, but it is built dynamically to reduce memory consumption. The reason is that the number of states in a statically constructed automaton is exponential in the number of branches in the pattern tree (just as the deterministic finite automaton of a regular expression can take exponential space with respect to the query size [78]). This can be best illustrated by the following example. Figure 2.11 depicts a static automaton that tests the *existence* of the tree pattern specified by $/a[b][c]/d^3$. The states of the automaton are labeled with partial path expressions that have been satisfied up to that point. The transition edges are labeled with the structural relationship to the previous state as well as the element name, separated by the symbol ‘::’. It is clear from the automaton that for a simple two-level pattern tree with n branches ($n = 3$ in this example), the number of states in the automata equals to $\sum_{i=0}^n \binom{n}{i} = 2^n$ since every combination of branches corresponds to a state.

As in most automata implementations, XNav traverses the data tree and generates events that trigger the state transitions. After each state transition, a set of possible states that are reachable from the current state is dynamically constructed. Besides state transition, the automaton also navigates the traversal in the tree. For example, depending on

³An automaton that produces results will be much more complex.

the current state, the automaton can instruct the traversal to jump from the current node to its following sibling. Therefore, navigation in this technique is determined dynamically while reading the data tree.

Koch [90] proposes a navigational processing approach based on tree automaton—Selecting Tree Automaton (STA)—to process path queries based on the Arb storage model. This approach scans the XML tree twice for each path expression. Each scan can be modeled as a deterministic tree automaton. The first scan is by means of a bottom-up deterministic tree automaton to determine which states are reachable. The second scan is by means of a top-down tree automaton to prune the reachable states and compute predicates that occur in all remaining states. Based on the Arb secondary storage, both scans can be implemented efficiently. However, two scans of the input is too expensive for some simple queries, such as `/bib/book`, which can be evaluated by XNav with a single scan.

2.2.3 Hybrid Join-based and Navigational Approach

Halverson et al. [74] proposed a mixed mode query processing technique for path expressions. A navigational operator *Unnest* and a join-based operator *Zig-Zag Join* are proposed to evaluate the path expression. While the Unnest operator can handle a general path expression, the Zig-Zag join operator can only deal with one location step similar to the binary structural join. The Unnest operator is a direct translation from a path expression to a finite state automaton, where each state is associated with a cursor of two types: child axis (CA) cursor and descendant axis (DA) cursor. The CA and DA cursors take an XML node as input and enumerate all children or descendants, respectively. This Unnest operator is in essence a static automaton version of XNav, but it does not guarantee a single-pass of the input.

The Zig-Zag join is a natural extension to the MPMGJN algorithm by making use of the index on the interval encodings (a similar indexing technique is proposed in [38]). The basic ideas of Zig-Zag join are two-fold: (1) it enables advancing the pointers in both input lists; and (2) it utilizes the interval encoding of the last item that failed the join condition to gauge the step to obtain the next input item. When advancing the pointers, an index is used to skip the input lists in sublinear time.

The mixed model query processing technique is to combine these two operators for evaluating a single path expression. To be able to compare the costs of different evaluation plans, Halverson et al. [74] define analytical cost models for both Unnest and Zig-Zag Join operators. An optimizer is also developed to enumerate left-deep plans from which to choose the optimal one based on the cost models. An interesting observation from the experiments is that the optimal plan coincides with the BlossomTree evaluation heuristic proposed in Chapter 3.

2.3 XML Indexing

Many XML indexing techniques have been proposed over the past few years. These indexes can also be roughly classified as extended relational approach and native approach. The extended relational indexes are usually developed for the extended relational storage, since it is natural to extend existing relational indexes, such as B⁺ tree and R-tree, on the relational tables to which the XML documents are mapped. The native indexing techniques (which are further classified as string-based and structural similarity based approaches) are usually independent from other XML storage systems (regardless whether they are extended relational or native). To some extent, native indexes can be treated as native storage systems themselves. The difference is on the design goal: native indexes mostly concentrate on efficient evaluation of usually a small fragment of the path expression, while the storage systems need to balance between multiple criteria, such as ease-of-update and maintaining orders among nodes. As a consequence of storage independence, evaluating path expressions using native indexes usually requires new index operators. The rest of this section concentrates on the algorithmic aspects of the techniques in each of these approaches.

2.3.1 Extended Relational Index

Many mature relational indexing techniques can be applied directly or extended easily to the XML context. These indexing techniques are usually based on the extended relational storage systems. For example, a B⁺ tree can be built on relational tables based on interval

encoding to perform an indexed nested-loop structural join [139, 105]. A B⁺ tree or a R-tree can be built on the table based on pre-order and post-order encoding [70] to retrieve child or descendant nodes efficiently. System RX [23] uses a path index and a value index (both are B⁺ trees) to efficiently answer queries, e.g., `//name="Maggie"`, that have both path and value constraints. The advantages of using these existing indexing techniques are straightforward: they are implemented by most database systems and they support various desired functionalities—scalability, concurrency control, and recovery, just to name a few.

Native path query processing techniques introduced in the previous section can also be extended with existing indexing techniques to boost their performance [28, 81, 82, 20]. These indexes, built with the deep understanding of the underlying path processing techniques, prune the input lists of the join operators. For example, XB-tree [28], a variant of B⁺ tree, improves the performance of holistic twig join on parent-child axes. XB-tree extends B⁺ tree by adding a bounding segment $[N.min, N.max]$ to every non-leaf index node N . The bounding segment is selected such that $N.min$ and $N.max$ are the minimum `begin` and maximum `end` encodings, respectively, of all descendant index nodes of N . With the help of XB-tree, the holistic twig join algorithm can skip some input items by only selecting non-leaf nodes whose bounding segment contains a certain `[begin, end]` interval.

XR-tree [81] also extends B⁺ tree with stab lists and bookkeeping information in the internal nodes. Stab lists are selected integers in the range $[b_{min}, e_{max}]$, where b_{min} and e_{max} are the minimum `begin` and maximum `end` over all indexed elements, respectively. An integer s “stabs” an interval $[b, e]$ if $b \leq s \leq e$. In the XR-tree, stab lists are the keys in the non-leaf index nodes and interval encoded elements are at the leaf index nodes. The stab list effectively partitions the intervals into a tree. Jiang et al. [82] proposed to modify the holistic twig join based on the knowledge of XR-tree. In addition to the index evaluation operator, one of the key issues is the heuristics for selecting the next pattern tree edge to “join” in order to skip as many input elements as possible.

Chen et al. [34] proposed to partition the holistic twig join input lists based on more structural information (e.g., level, and path prefix) in addition to element names. The refinement of the input lists brings potential benefits for queries containing parent-child constraints.

All these indexing techniques are tightly coupled with the supporting processor and

usually require modifications of both the existing indexes and the path query processing operators.

2.3.2 String-based Index

Another line of XML indexing techniques reduce the tree pattern matching problem to string pattern matching problem [129, 116, 128]. Wang et al. propose ViST index [129] to convert an XML tree into a sequence of tree nodes represented by the element name and its rooted path prefix. This encoding scheme is very similar to XRel [137] in that they both record the rooted path of each node as a string to match against a regular expression representation of a path expression. The difference is that ViST translates a branching path query into a regular expression on strings so that branching path queries can also be evaluated without joins. However, due to the unordered nature of branching predicates (i.e., $//a[b][c]$ has the same semantic meaning as $//a[c][b]$), the number of regular expressions translated from a multi-branch query is exponential in the number of branches. This may significantly deteriorate the overall query performance.

Rao and Moon propose PRIX [116] to use Prüfer sequence [115] as a more compact representation for XML trees. Prüfer sequence can be generated by iteratively deleting the least labeled (the authors use post-order) leaf and appending its parent label to the list. This approach still needs to enumerate all possible sequences for a branching path query.

Wang and Meng [128] propose another sequencing technique by incorporating more structural information (e.g., the interval encoding) with the tree node and adding pointers between siblings. By adding more information to the index, query performance is improved. However, the query processing algorithm grows very complex and it is unclear whether the main memory based storage is scalable to large documents.

2.3.3 Structural Similarity-based Index

A large body of XML indexing research focuses on *structural clustering indexes*, which group the XML tree nodes by their structural similarity. Although they may be based on different notions of similarity (e.g., common path prefix [64], bisimilarity [99], and F&B bisimilarity [87]), the basic idea is the same: similar tree nodes are clustered into

equivalence classes (or *index nodes*), which are connected to form a tree or graph. For example, [87] defines the forward bisimilarity relation on the set of XML tree nodes X as follows: $u, v \in X$ is bisimilar ($u \approx v$), if and only if

1. u and v have the same name;
2. $\text{parent}(u) \approx \text{parent}(v)$;
3. if there is an IDREF reference edge from some node u' to u , then there is a reference node from some node v' to v such that $u' \approx v'$.

Forward bisimilar vertices are clustered together into an equivalence class called *bisimulation vertex*. Two bisimulation vertices are connected by directed edges if some vertices from each of the two equivalence classes have a parent-child relationship in the XML tree. The result is a *bisimulation graph* which is the structural index itself. The F&B bisimilarity can be obtained by refining the bisimulation graph: splitting a bisimulation vertex into multiple ones if XML tree nodes in the equivalence class are not *backward bisimilar*. The backward bisimilarity is defined similarly as the forward bisimilarity by reversing the directions of all edges in the XML tree first.

Since the structural clustering indexes are graphs, existing XML query processing techniques cannot be directly applied. Therefore, new index evaluation operators need to be developed for each of the index graphs. Different navigational operators (e.g., those based on DFS, BFS [131]) are proposed for the structural indexes. These index processing techniques can be thought of as navigational operators on indexed graphs. Therefore, they bear similar properties of the navigational operators on XML trees, e.g., it is efficient to evaluate path expressions with only $/$ -axes, but relatively inefficient for expressions containing $//$ -axes, particularly on large index graphs. Although structural indexes are reasonably small for regular data sets, they could grow very large for structure-rich data. Various techniques are developed to cope with this situation: e.g., materializing the index on disk [131], or limiting the similarity definition by tree depth to tradeoff the covered query set against the space requirement (e.g., $A(k)$ -index [88], $D(k)$ -index [33] and $M(k)$ - and $M(k)^*$ -index [75]). However, index evaluation still requires complex pattern matching on the whole graph.

2.4 Path Query Optimization

There are many fundamental problems in database query optimization, e.g., query rewrite, execution plan enumeration, and plan cost estimation. Since the accuracy of cost estimation is usually heavily dependent on the accuracy of cardinality estimation of subqueries, many techniques are proposed to deal with cardinality estimation. Section 2.4.1 introduces prior work on plan enumeration, and Section 2.4.2 focuses on the cardinality estimation.

2.4.1 Plan Enumeration and Selection

In the extended relational approach, where XML queries are translated into SQL statements, query optimization mostly relies on the underlying relational database systems. Previous research focuses on query mappings that preserve the semantics rather than efficiency [137, 48].

In the join-based native query processing approach, execution plan enumeration usually amounts to enumerating the join orders. Wu et al. [133] examine multiple strategies of exploring the space of execution plans with different join orders. A dynamic programming with pruning algorithm and a heuristics that only considers fully pipelined plans are proposed to quickly find the optimal join order.

Kanne et al. [85] compare the performance of different navigational primitives: a *Simple* method that translates each location step to an Unnested-Map operator, a more I/O friendly *XSchedule* operator that optimizes inter-cluster navigation, and a *XScan* operator that sequentially scans all the data. The last operator usually involves sequential I/O while the other two involve asynchronous I/O. Experiments show that: (1) XSchedule almost always outperforms the Simple method due to the better I/O locality, and (2) XSchedule and XScan can outperform one another depending on the selectivity of the queries. If most of the data need to be examined during the query evaluation, XScan is better due to its sequential access pattern. XSchedule wins if the query is highly selective.

As introduced in Section 2.2.3, another plan enumeration and selection framework is developed by Halverson et al. [74]. In this framework, plans consisting of a navigational operator and a join-based operator are enumerated and costed. However, the cardinality or selectivity estimation is a missing piece in that work.

2.4.2 Cardinality Estimation

A large body of research focus on cardinality estimation [64, 13, 37, 59, 132, 110, 111, 14, 113, 112, 130], where most of them concentrate on queries that consist of structural constraints only [64, 13, 37, 59, 110, 113, 112]. All these techniques first summarize the XML trees into a small synopsis that contains structural information and statistics. The synopsis is usually stored in the database catalog and is used as the basis for estimating cardinality. Depending on how much information is reserved, different synopses cover different types of queries. To be able to compare different synopses, the following criteria are considered:

- C1: Does the synopsis support branching path queries as well as simple path queries?
- C2: Does the synopsis produce accurate estimation results for the queries that they support?
- C3: Is the synopsis efficient to construct?
- C4: Is the synopsis adaptive to memory budget?
- C5: Does the synopsis support structural and value constraints?
- C6: Is the synopsis recursion-aware?
- C7: Does the synopsis support incremental update if the XML documents are updated?

These features are all important for a synopsis to be practical in real world database systems.

Chen et al. [37] proposed to reduce the problem of cardinality estimation on twigs to the estimation problem on substrings. Their definition of twig matching, however, is different than the semantics of path expressions. Therefore, their work cannot be directly used for estimating cardinality of path expressions.

DataGuide [64] was first designed for semistructured data (graph-based OEM data model [109]). It records all distinct paths from a data set and compresses them into a compact graph. Aboulnaga et al. [13] proposed two synopses—path trees and Markov tables—to compress many XML documents to small synopses. Path tree, similar to DataGuide,

also captures all distinct paths in the XML trees, along with the cardinality statistics associated with the synopsis vertices. When the path trees are too large, vertices are compressed into one specially labeled wildcard (*) vertex, with the aggregated cardinality statistics. Markov tables, on the other hand, capture the sub-paths under a certain length limit. Selectivity of longer paths are calculated using fragments of sub-paths similar to Markov process. Markov tables can also be compressed using suffix wildcards (*). Freire et al. proposed StatiX [59] to keep statistics—children count and value histograms—with the XML schemata. Their focus is on transforming the schema such that statistics with multiple granularities can be collected. Wu et al. [132] proposed a two-dimensional histogram synopsis based on the `start` and `end` positions in the interval encoding. Wang et al. [130] proposed a Bloom histogram synopsis that supports efficient cardinality estimation as well as update of the underlying XML data. All of these synopsis structures only support simple linear path queries that may or may not contain `//`-axes. Therefore, estimating cardinality for branching path queries remained a challenge until the recent development of structural clustering techniques, reviewed next.

XSketch [110] and TreeSketch [112] are two synopses that are based on structural clustering techniques. XSketch is based on forward- and backward-stability, and TreeSketch is based on count-stability, which is a refinement of forward-stability. The clustered graph is too large for structure-rich data for the same reason that F&B index is too large. Therefore, XSketch and TreeSketch develop different heuristics to summarize these graphs under a memory budget. Since obtaining the optimal summarization is NP-hard for both types of clustered graphs, the estimation accuracy is largely dependent on the how well the heuristics preserves the structural information in the original XML trees. XSketch employs a bottom-up heuristics and TreeSketch embraces a top-down heuristics.

XSketch [110] starts from a small label-split graph, which is generated from the XML tree by merging all XML nodes with the same label to one synopsis vertex and keeping the tree edges (if there are multiple edges between two synopsis vertices after the merging, just keep one) in the graph. Therefore, a vertex in the XSketch synopsis graph is associated with a set of XML nodes, called *extent* of the synopsis vertex. Since a vertex implies a unique path from the root, every vertex is labeled with the count of elements that can be reached by the corresponding rooted path in the XML tree. Furthermore, the edge

of XSketch synopsis is labeled with two boolean variables indicating whether this edge is forward and/or backward stable. An edge (U, V) in the XSketch synopsis is backward stable (B-stable) if and only if for every $v \in extent(V)$, there exists a $u \in extent(U)$ such that edge (u, v) is in the XML tree. Forward stability (F-stable) is defined similarly by reversing the synopsis edge direction. Forward and Backward stability properties are critical to estimate cardinalities of branching path expressions. The original XML tree and the label-split graph are at the two extremes of the spectrum of F- and B-stability. It is not hard to see that all edges in the XML tree are F- and B-stable, but many edges in the label-split graph may not be F- and B-stable. Therefore, the heuristics developed in [110] is to decide, under the memory constraints, how to split (or refine) vertices in the label-split graph with the objective to eliminate the uniformity and independence assumptions that are the basis for cardinality estimation, or at least make such assumptions more realistic. Three vertex refinement operations (**b-stabilize**, **f-stabilize**, and **b-split**) are developed for the heuristics to choose which one to apply for splitting a specific vertex.

TreeSketch [112] is based on the notion of count-stability: a synopsis edge (U, V) is k -stable if and only if for every $u \in extent(U)$, there are exactly k nodes $v_1, \dots, v_k \in V$ such that $(u, v_i), i \in \{1, \dots, k\}$ is an edge in the XML tree. Count stability is a refinement of F-stability in that every count stable edge is also an F-stable edge, but not the other way around. A synopsis is count-stable if every edge is k -stable for some k . Different from XSketch, TreeSketch first constructs a count-stable graph from an XML tree and merge (or summarize) vertices until the memory budget is met. The objective of the merging is to minimize the overall squared error introduced by the merging. Performance study shows that TreeSketch has orders of magnitude better estimation accuracy with even less time for construction time than XSketch. However, the construction time for TreeSketch is still prohibitive for structure-rich data. Another disadvantage with XSketch and TreeSketch is that updating the synopsis is also expensive if the XML documents are updated.

In summary, Table 2.2 listed the synopses and their features. One problem that is associated with all these techniques is that they are not recursion-aware. Recursive XML documents abound in real life data sets [39], and they represent the hardest case for cardinality estimation. Making the synopses recursion-aware may greatly improve the estimation accuracy.

Synopses	C1	C2	C3	C4	C5	C6	C7
DataGuide	no	yes	yes	no	no	no	yes
Path tree	no	yes	yes	yes	no	no	no
Markov table	no	yes	yes	yes	no	no	no
StatiX	no	yes	yes	partly	yes	no	no
Wu et al. [132]	no	yes	yes	yes	no	no	no
Bloom Histogram	no	yes	yes	no	no	no	yes
Chen et al. [37]	yes	yes	yes	no	no	no	no
XSketch	yes	yes	no	yes	yes	no	no
TreeSketch	yes	yes	no	yes	no	no	no

Table 2.2: Comparisons of different synopses (Recall that C1=supporting branching path; C2=accurate estimation; C3=ease-of-construction; C4=adaptivity to memory budget; C5=supporting value constraints; C6=recursion-aware; C7=updatability)

Chapter 3

Processing of Path Expressions and Succinct Native Storage Scheme

3.1 Introduction

This chapter presents a path expression evaluation strategy that exploits the advantages of several existing approaches. This strategy first rewrites a general path expression into an expression using a minimum subset of axes. Then a heuristic evaluation technique decomposes the path expression into subexpressions and applies a navigational approach to evaluating the subexpressions followed by a join-based approach to merging the intermediate results. The chapter then focuses on a specific problem of how to design the XML storage system that supports the evaluation strategy and balances query and update costs.

As introduced in Chapter 2, previous research on the evaluation of path expressions mainly fall into two approaches. The *navigational* approach traverses the tree structure and tests whether a tree node satisfies the constraints specified by the path expression [125, 27, 21, 84, 90]. The *join-based* approach first selects a list of XML tree nodes that satisfy the node-associated constraints for each pattern tree node, and then pairwise joins the lists based on their structural relationships (e.g., *parent-child*, *ancestor-descendant*, etc.) [139, 15, 28, 123, 66, 71]. Using proper labeling techniques [43, 36, 126], tree pattern matching (TPM) can be evaluated reasonably efficiently by various join techniques (multi-predicate merge joins [139], stack-based structural joins [15], or holistic twig joins [28]).

Compared to the navigational techniques, join-based approaches are more scalable and enjoy optimization techniques from the relational database technology. However, there are inevitable difficulties:

1. Since choosing the optimal structural join order is NP-hard, the query optimizer relies heavily on heuristics [133]. When the query size is reasonably large, the optimization time may dominate the execution time. Thus, it is hard for the query optimizer to compromise between optimization and execution.
2. The selection-then-join methodology is not adaptive to streaming XML data (e.g., SAX events) where the input streams could be considered as infinite and selection on the infinite input will not terminate.

In this chapter, a novel path expression processing approach is proposed to combine the advantages of both navigational and join-based approaches [143]. The rationale is based on the observation that some of the structural relationships imply *higher degree of locality* in the XML document than others, and thus may be evaluated more efficiently using the navigational approach. On the other hand, others represent more *global* relationships, and thus may be evaluated more efficiently using the join-based approach. For example, **parent-child** is a more local relationship than **ancestor-descendant** since finding the parent or child of a node requires only one navigation along the edge, but finding ancestor or descendant requires traversing a path or the whole subtree. If XML elements are *clustered* at the physical level based on one of the “local” structural relationships (say **parent-child**), the evaluation of a subset of the path expression consisting of only those local structural relationships can be performed more efficiently using a navigational technique without the need for structural joins. Therefore, clustering by structural relationship is an important requirement for native storage systems.

Based on this idea, a special type of path expression, called the *next-of-kin* (NoK) pattern tree, is identified to reflect the locality concept. A NoK pattern tree is a pattern tree whose nodes are connected by **parent-child** and **following-/preceeding-sibling** (abbreviated by \triangleleft) relationships only. These axes are called *local axes*. It is straightforward to partition a general pattern tree into NoK pattern trees, which are interconnected by arcs labeled with **//** or other “global” structural relationships such as **following/preceeding**. The evaluation

strategy based on this idea is as follows: a general path expression is first rewritten as a pattern tree that consists of a minimum subset of axes. The pattern tree is then partitioned into interconnected NoK pattern trees, to which the navigational pattern matching operators are applied. Finally, the intermediate results of the NoK pattern matching are joined based on their structural relationships, just as in the join-based approach. Note that the complexity for evaluating a path expression consisting of $/$, $//$, $[]$, and $*$ is P-hard with respect to combined data and query complexity [67]. Gottlob et al. [66] proposed an $O(mn)$ algorithm, where m and n are the sizes of the query and data, respectively. The NoK pattern matching can also be evaluated in $O(mn)$, but it only needs a single scan of the document while the Gottlob's algorithm needs multiple scans.

The effectiveness of this approach depends on the answers to the following two questions: (1) How many local relationships are there compared to global relationships in the actual queries? (2) How to efficiently evaluate NoK pattern matching so that its performance is comparable to or better than structural joins? The first question is hard to answer since it depends on the actual usage domain of the query, but a simple statistical analysis of the queries in the XQuery Use Cases [32] reveals that approximately 66% of structural relationships are $/$'s, 33% are $//$'s, and the rest are **self** axes ($“.”$) [144]. Figure 3.1 shows the detailed distribution of axes in each sample query from the XQuery Use Cases. This fact partly justifies that using NoK pattern matching first will significantly reduce the number of structural joins later.

The answer to the second question relies on how well the physical storage system satisfies the clustering criteria. To justify this conjecture, a simple and succinct physical storage scheme is proposed to support efficient navigational NoK pattern matching. Since the storage scheme has the locality property, an update of the XML document (insertion/deletion of an element) only affects part of the whole structure, making it more amenable to update than other techniques (e.g., the interval encoding [48]).

The rest of the chapter is organized as follows: Section 3.2 introduces the rewriting rules that translate the whole set of axes into a minimum subset. Section 3.3 presents a decomposition-based heuristic approach to evaluating path expressions. Section 3.4 presents the algorithm for NoK pattern matching at a logical level. Section 3.5 presents the design of the physical storage scheme. Section 3.6 introduces how to realize the logical

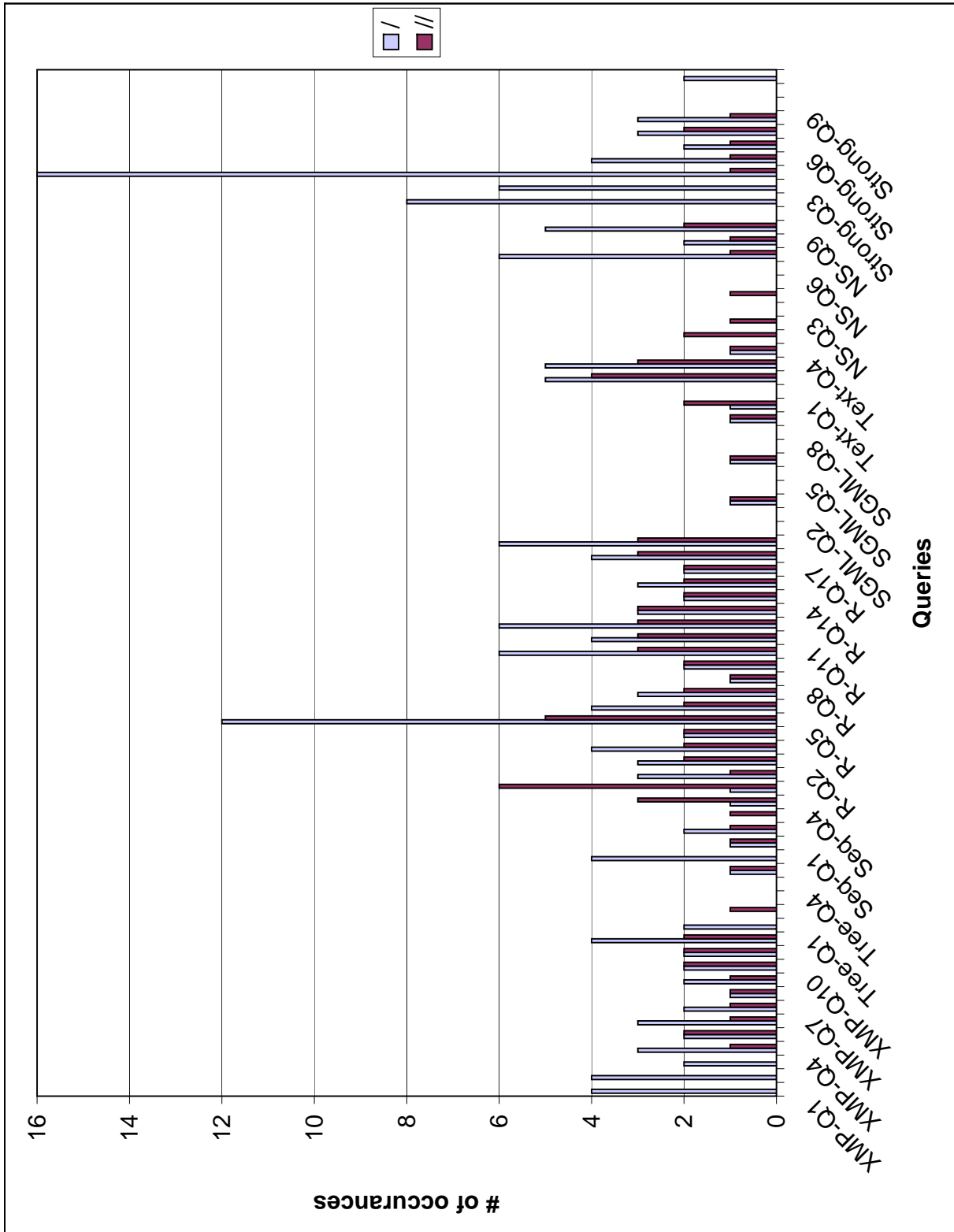


Figure 3.1: Number of / and // axes in the XQuery Uses Cases sample queries

level NoK pattern matching algorithm over this physical storage. Section 3.7 presents the implementation and experimental results. Section 3.8 compares related work to the NoK approach.

3.2 Rewriting Axes

Given a general path expression consisting of any types of axes, the query compiler first rewrites the axes using a minimum subset. There are many benefits for the rewriting. For example, the rewriting reduces the types of axes that need to deal with. More importantly, *backward axes*, such as **parent**, can be rewritten using *forward axes* such that an one-pass evaluation algorithm, such as the one introduced in Section 3.4, can be applied. Olteanu et al. [106] proposed a similar technique to rewrite backward axes to forward axes. The rewriting technique introduced in this thesis goes one step further to rewrite the forward axes to a minimum subset of axes. A more detailed comparison is presented in Section 3.8.

As discussed in Chapter 1, path expressions have thirteen axes. Among these, **attribute** and **namespace** are two that specify types of the nodes rather than the structural relationships between two steps. Therefore, they are not included in the following rewriting. In the rest of the chapter, let \blacktriangleleft and \triangleleft denote **following** and **following-sibling** axes, respectively.

The set of axes defined in the path expression is redundant in the sense that all of them can be rewritten using a small subset. In fact, the minimum set is not unique. For example, both $\{., /, //, \blacktriangleleft\}$ and $\{., /, //, \triangleleft\}$ can be used to rewrite the other axes. The following theorem presents the rewrite rules for each of the two sets.

Theorem 3.1 (Rewriting Rules for Axes I) *Given any pattern tree $G(V, E)$ as defined in Definition 1.5, it can always be converted to a graph whose edge labels are in the set $\{., /, //, \blacktriangleleft\}$. Assume that (p, c) is an edge and $\lambda(p, c)$ denotes that the axis associated with the edge is “ λ ”, and $label_e$ and $label_n$ represent edge label and node label, respectively. The rewriting rules are as follows:*

$$(a) \text{ child}(p, c) \iff label_e(p, c) = "/"$$

$$(b) \text{ parent}(p, c) \iff label_e(c, p) = "/"$$

$$(c) \text{ descendant}(p, c) \iff \exists x \text{ label}_e(p, x) = "/" \wedge \text{label}_e(x, c) = "//" \wedge \text{label}_n(x) = "*"$$

$$(d) \text{ ancestor}(p, c) \iff \exists x \text{ label}_e(x, p) = "/" \wedge \text{label}_e(c, x) = "//" \wedge \text{label}_n(x) = "*"$$

$$(e) \text{ self}(p, c) \iff p = c$$

$$(f) \text{ descendant-or-self}(p, c) \iff \text{label}_e(p, c) = "/"$$

$$(g) \text{ ancestor-or-self}(p, c) \iff \text{label}_e(c, p) = "/"$$

$$(h) \text{ following-sibling}(p, c) \iff \exists x \text{ label}_e(x, p) = "/" \wedge \text{label}_e(x, c) = "/" \wedge \text{label}_e(p, c) = "\blacktriangleleft" \wedge \text{label}_n(x) = "*"$$

$$(i) \text{ preceding-sibling}(p, c) \iff \exists x \text{ label}_e(x, p) = "/" \wedge \text{label}_e(x, c) = "/" \wedge \text{label}_e(c, p) = "\blacktriangleleft" \wedge \text{label}_n(x) = "*"$$

$$(j) \text{ following}(p, c) \iff \text{label}_e(p, c) = "\blacktriangleleft"$$

$$(k) \text{ preceding}(p, c) \iff \text{label}_e(c, p) = "\blacktriangleleft"$$

Their graphical representations are in Figure 3.2(a)–(k) in that order.

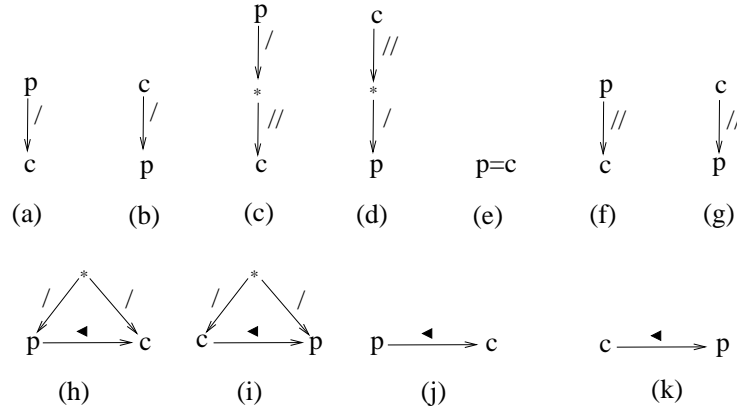


Figure 3.2: Converting axes to $\{., /, //, \blacktriangleleft\}$. (a) $\text{child}(p, c)$ (b) $\text{parent}(p, c)$ (c) $\text{descendant}(p, c)$ (d) $\text{ancestor}(p, c)$ (e) $\text{self}(p, c)$ (f) $\text{descendant-or-self}(p, c)$ (g) $\text{ancestor-or-self}(p, c)$ (h) $\text{following-sibling}(p, c)$ (i) $\text{preceding-sibling}(p, c)$ (j) $\text{following}(p, c)$ (k) $\text{preceding}(p, c)$

PROOF The child, parent, descendant-or-self, ancestor-or-self, self, following, and preceding axes are straightforward based on the semantics of edge labels “/”, “//”, “.” and “◁”. Since ancestor is the reverse of descendant, and preceding-sibling is the reverse of following-sibling, we prove descendant and following-sibling only. For rule (c), by the semantics of descendant-or-self axis, $\text{descendant-or-self}(x, c) \equiv \text{descendant}(x, c) \vee x = c$. Therefore, the right-hand-side of (c) can be rewritten to

$$\begin{aligned} & \exists x \text{ child}(p, x) \wedge \text{descendant-of-self}(x, c) \vee \text{child}(p, x) \wedge x = c \\ \iff & \text{descendant}(p, c) \vee \text{child}(p, c) \\ \iff & \text{descendant}(p, c) \end{aligned}$$

■

which is the left hand side.

Rule (h) is straightforward since the right-hand-side is exactly the semantics of following-sibling axis: p and c have the same parent and c is following p in document order.

Theorem 3.2 (Rewriting Rules for Axes II) *All of the eleven axes in Theorem 3.1 can be rewritten to the set $\{., /, //, \triangleleft\}$. The rewriting for child, parent, descendant, ancestor, self, descendant-or-self, and ancestor-of-self are exactly the same as the corresponding rewrite rules in Theorem 3.1 since they do not use \triangleleft . The rewrite rules for the remaining axes are as follows:*

$$(a) \text{ following-sibling}(p, c) \iff \text{label}_e(p, c) = “ \triangleleft ”$$

$$(b) \text{ preceding-sibling}(p, c) \iff \text{label}_e(c, p) = “ \triangleleft ”$$

$$(c) \text{ following}(p, c) \iff \exists y, z \text{ label}_n(y) = “ * ” \wedge \text{label}_n(z) = “ * ” \wedge \text{label}_e(y, p) = “ // ” \wedge \text{label}_e(z, c) = “ // ” \wedge \text{label}_e(y, z) = “ \triangleleft ”$$

$$(d) \text{ preceding}(p, c) \iff \exists y, z \text{ label}_n(y) = “ * ” \wedge \text{label}_n(z) = “ * ” \wedge \text{label}_e(y, c) = “ // ” \wedge \text{label}_e(z, p) = “ // ” \wedge \text{label}_e(y, z) = “ \triangleleft ”$$

Their graphical representations are in Figure 3.3(a)–(d) in that order.

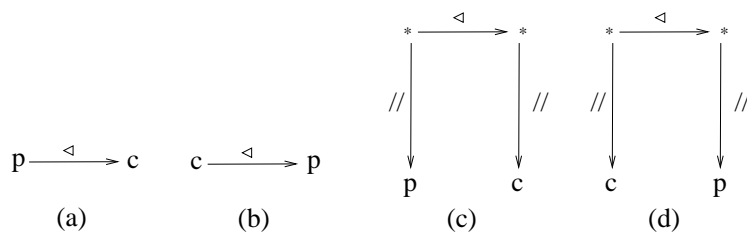


Figure 3.3: Converting axes to $\{., /, //, \triangleleft\}$. (a) **following-sibling**(p,c) (b) **preceding-sibling**(p,c) (c) **following**(p,c) (d) **preceding**(p,c)

PROOF (a) and (b) are straightforward and (d) is just reverse of (c). So a proof of (c) is sufficient.

From the formal semantics of path expression [53], if two nodes p and c satisfy **following**(p, c), there must exist two nodes y and z such that they are ancestor-or-self of p and c , respectively, and z is a following sibling of y . Therefore, the result of the rewriting is a union of two pattern trees. ■

Since the above two sets are possible targets for the rewrite, the query compiler can rewrite a path query into two equivalent pattern trees (corresponding to logical plans) and let the query optimizer choose the pattern tree that leads to the optimal physical plan. Section 3.3 presents a heuristic evaluation strategy that takes a pattern tree and translates it into a physical plan. This heuristic is based on the observations of the advantages and disadvantages of different physical operators. A better solution is to develop a cost model for each of the operator. In Chapter 5, a synopsis structure and cardinality estimation algorithm for a cost-based optimizer is presented. Its possible application in a cost model is discussed in Chapter 6.

3.3 BlossomTree Decomposition

Given a pattern tree after rewriting, the hybrid BlossomTree heuristics [143, 141] first decomposes it into interconnected NOK pattern trees. Each NOK pattern tree is evaluated by a navigational pattern matching operator. The intermediate results are then joined together using the join-based approach.

Algorithm 1 gives the pseudo-code to decompose a pattern tree based on depth-first traversal. In the parameters, pt is a pattern tree, S is a set containing the roots of the decomposed NoK pattern trees, and T is a set containing the non-root nodes in the current NoK pattern trees. Initially, S is a singleton set containing the root of pt (line 1). From line 6 to 14, the algorithm adds child nodes reached by a local (respectively global) axis to T (respectively S). Since every vertex in T is a non-root node, the algorithm calls the DFS function (line 15) to build a complete NoK tree. This function traverses the pattern tree in depth-first search from vertices in T . During the traversal, it separates non-root nodes and NoK root nodes in T and S , respectively (lines 6–8). The function DFS calls itself recursively to traverse the whole subtree (line 12). Line 12 in function DECOMPOSE and line 9 in function DFS mark the incident nodes to a global-axis-labeled edge as *join nodes*, which indicates that they belong to different NoK pattern trees and their matched nodes need to be joined later (explained latter). It is straightforward to see that the pattern tree decomposition algorithm is $O(n)$ where n is the number of vertices of the pattern tree.

Example 3.1 Figure 3.4a illustrates the BlossomTree decomposition of the expression $/a[b//d][c//e]$ into three interconnected NoK pattern trees: N_1 corresponding to $/a[b][c]$, and N_2 and N_3 corresponding to $//d$ and $//e$, respectively. When N_1 is matched to an XML document, the intermediate results must include XML nodes that matched with b and c since they need to be joined with d and e , respectively. Therefore, a relation $N_1(a, b, c)$ is introduced to store intermediate results generated by the NoK pattern matching. Similarly, the matching results for N_2 and N_3 are both relations containing one column $N_2(d)$ and $N_3(e)$. Figure 3.4c shows the relations as the matching results of N_1 , N_2 , and N_3 on the XML document in Figure 3.4b. Relational operators, such as projection, selection, and joins, can be applied to the relational intermediate results. A logical structural join operator that is similar to the ones discussed in Chapter 2 can combine the intermediate results into the final result. In this example, the final result can be obtained by joining the three relations: $((N_1 \bowtie_{b//d} N_2) \bowtie_{c//e} N_3)$ or $((N_1 \bowtie_{c//e} N_3) \bowtie_{b//d} N_2)$. Therefore, the remaining question is how to efficiently evaluate NoK pattern matchings. \square

Algorithm 1 Decomposing a pattern tree into NoK pattern trees

DECOMPOSE($pt : \text{PatternTree}, S : \text{Set}, T : \text{Set}$)

```

1   $S \leftarrow$  root of  $pt$ ;
2  while  $S \neq \emptyset$  do
3      extract an item  $u$  from  $S$ ;
4       $T \leftarrow \emptyset$ ;
5      initialize  $t$  as empty NoK tree;
6      for each out-edge  $(u, v)$  s.t.  $v$  has not been visited do
7          if the label of  $(u, v)$  is a local axis then
8              set  $v$  as visited;
9               $T \leftarrow T \cup \{v\}$ ;
10             add  $v$  and edge  $(u, v)$  in  $t$ ;
11         else  $S \leftarrow S \cup \{v\}$ ;
12             mark  $u$  and  $v$  as join nodes;
13         end
14     end
15     DFS( $t, S, T$ );
16     output  $t$ ;
17 end

```

DFS($t : \text{NoKBlossomTree}, S : \text{Set}, T : \text{Set}$)

```

1  while  $T \neq \emptyset$  do
2      extract an item  $u$  from  $T$ ;
3      for each out-edge  $(u, v)$  s.t.  $v$  has not been visited do
4          if the label of  $(u, v)$  is a local axis then
5              set  $v$  as visited;
6               $T \leftarrow T \cup \{v\}$ ;
7              add  $v$  and edge  $(u, v)$  in  $t$ ;
8          else  $S \leftarrow S \cup \{v\}$ ;
9              mark  $u$  and  $v$  as join nodes;
10         end
11     end
12     DFS( $t, S, T$ );
13 end

```

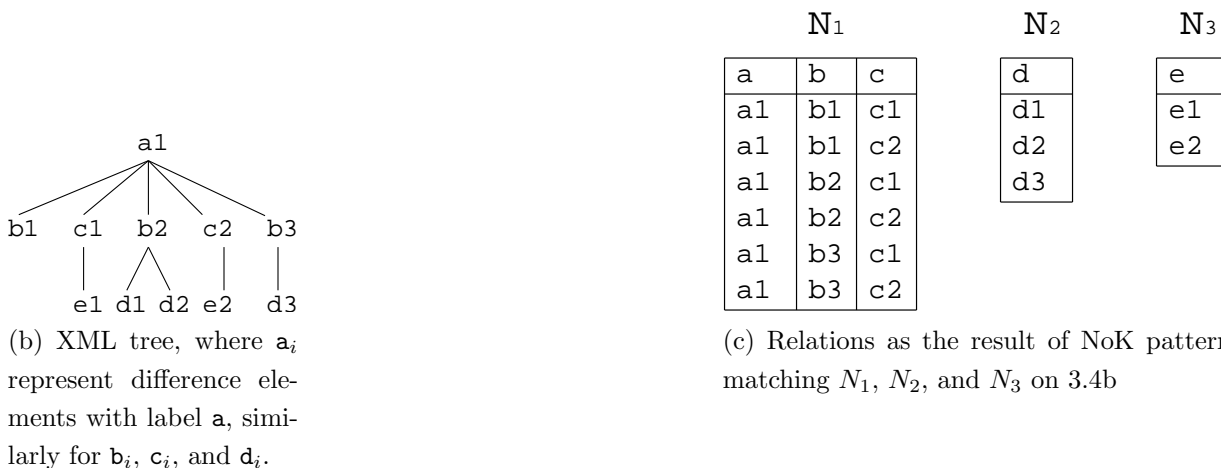
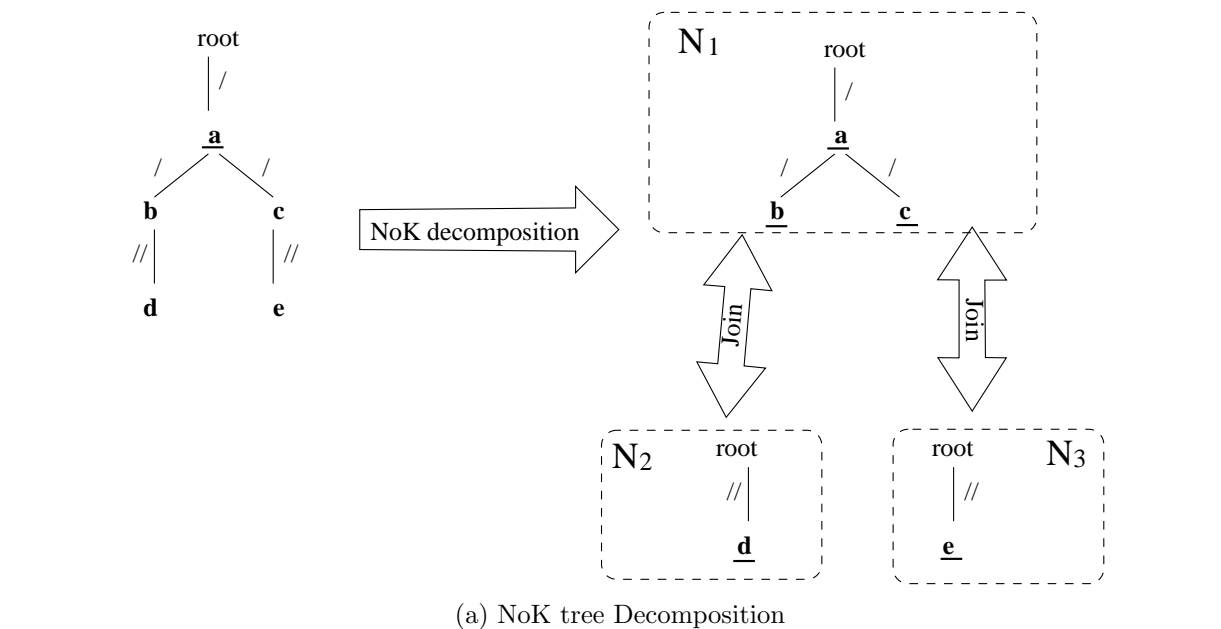


Figure 3.4: NoK Decomposition and relation of NoK pattern matching

3.4 Logical Level NoK Pattern Matching

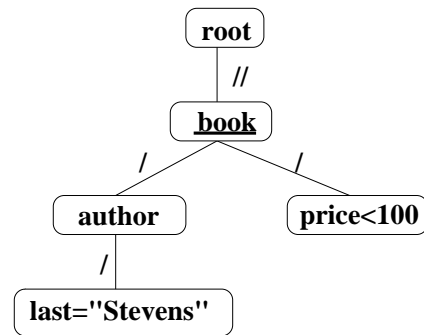
Throughout the rest of the chapter, the XML document in Figure 3.5a and the path expression `//book[author/last="Stevens"][price<100]` (its pattern tree is shown in Figure 3.5b) are used to illustrate the storage and query processing.

```

<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="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>
<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="1999">
  <title>The Economics of Technology and Content for Digital TV</title>
  <editor>
    <last>Gerbarg</last><first>Darcy</first>
    <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>
</bib>

```

(a) An XML bibliography file



(b) An example pattern tree for //book[author/last="Stevens"] [price<100]

Figure 3.5: An XML file and a pattern tree

To efficiently store the tree, tag names are first mapped to the characters—the short representations of tag names—in an alphabet Σ . For example, one possible mapping of

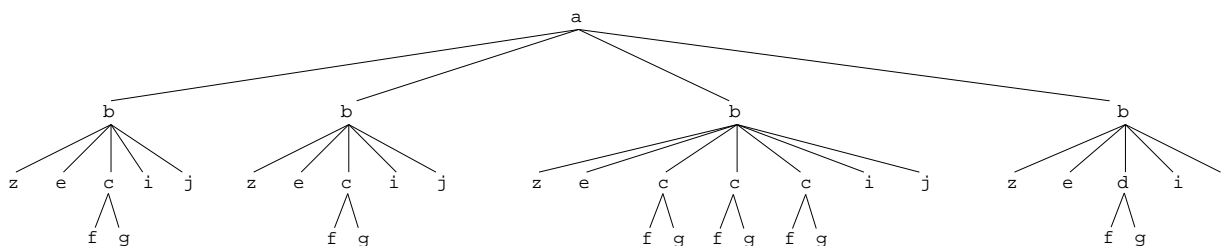


Figure 3.6: Subject tree representation of the bibliography XML document

tag names in Figure 3.5a to the alphabet $\Sigma = \{a, b, c, e, f, g, i, j, z\}$ could be as follows:

<code>bib</code> \rightarrow <code>a</code>	<code>book</code> \rightarrow <code>b</code>	<code>@year</code> \rightarrow <code>z</code>
<code>author</code> \rightarrow <code>c</code>	<code>title</code> \rightarrow <code>e</code>	<code>publisher</code> \rightarrow <code>i</code>
<code>price</code> \rightarrow <code>j</code>	<code>first</code> \rightarrow <code>f</code>	<code>last</code> \rightarrow <code>g</code>

Following this mapping, the XML document in Figure 3.5a can be represented as a tree, called *subject tree* or *XML tree* (Figure 3.6). In the subject tree, only tag names and their structural relationships are preserved. The value of each tree node is detached from the structure and stored separately. The reason is discussed in Section 3.5.

There are two steps in the process of matching NoK pattern trees to the subject tree: locating the nodes in the subject tree to start pattern matching, and NoK pattern matching from that starting node. The first step is needed since a NoK pattern tree `b/c` could be obtained from the path expression `/a//b/c` in which case any descendant of `/a` could be a starting point for NoK pattern matching.

In string pattern matching, the major concern is how to efficiently locate the starting points, while matching the string itself is straightforward. In the case of NoK pattern matching, both steps are nontrivial. There could be many options to locate the starting point:

Naïve approach: Traverse the whole subject tree in document order and try to match each node with the root of the NoK pattern tree. If a matching node is found, then

start the NoK pattern tree matching process from that node. This is exactly what might be done in the streaming XML context.

Index on tag names: If a B⁺ tree on tag names is available, an index lookup for the root of the NoK pattern tree will generate all possible starting points.

Index on data values: If there are value constraints in the NoK pattern tree (such as `last="Stevens"` in Figure 3.5b), and a B⁺ tree is constructed for all values in the XML document, the value-based index can be used to locate all nodes having the particular value. For each node, a root-lookup operation is performed to find the XML node that matches with the root of the pattern tree. Each of these nodes is then used as a starting point for a NoK pattern matching.

All three strategies are implemented and evaluated in Section 3.7. In the experiments, the best method is manually chosen assuming a perfect optimizer. Note that there is another evaluation strategy that traverses the tree from bottom-up. This strategy is very similar to the bottom-up tree automaton technique [90] that is introduced in Section 2.2.

The function NOK-MAIN in Algorithm 2 is the main function that takes a pattern tree node *pnode* and a subject tree node *snode* as inputs and produces a list of subject nodes *R* as output. As described above, the *pnode* may or may not be the root of the pattern tree. In either case, the function ROOT-LOOKUP is invoked to find the subject tree nodes that matches with the root of the pattern tree. ROOT-LOOKUP is a straightforward recursive function: it first checks whether the labels of the pattern tree node and the subject tree node matches (lines 1–3). If they match, the same function is recursively called with the parameters of the parents of the pattern tree node and the subject tree nodes, respectively. The PARENT function in line 7 is an interface to the storage system, which is introduced in Section 3.6. When the function ROOT-LOOKUP returns true, the *snode* is the starting point for the NoK pattern matching, which is codified by the function NPM.

Having established the starting points, NoK pattern matching needs to deal with the unordered nature of siblings. That brings up the complexity that there could be more than one pattern tree node that matches a subject tree node. Moreover, the partial order constraints on siblings specified by the **following/preceding-sibling** axes also need to be taken care of (recall that, in general, the children of a pattern tree node is a DAG connected by \triangleleft arcs). We call the children of a *pattern* tree node *frontiers* if their sibling-indegree is 0, i.e.,

no sibling occurs before them according to the following/preceding-sibling constraints. The frontiers represent the current ready-to-match nodes, and the set should be dynamically maintained since a matched frontier should be deleted (if it is not the returning node) and its “following siblings” in the pattern tree should be added if their sibling-indegree is now zero. This process is codified in Algorithm 2, which is a logical-level NoK tree pattern matching algorithm that returns `true` if the pattern tree rooted at *proot* matches the subject subtree rooted at *snode* (the starting node) in the subject tree. Initially, the third parameter *R* is set to \emptyset , and it will contain a list of subject tree nodes (in document order) that match the returning node. The precondition of the algorithm is that the label of *proot* matches that of *snode*.

In lines 1–3 of Algorithm 2, if *proot* is found to be the returning node in the pattern tree, its matching *snode* is put in the result list *R*. Since there could be multiple subject tree nodes that match the returning node in different recursive calls, *snode* must be appended to the resulting list. Lines 6 and 16 contain the only two operations on the subject tree. Together they implement the traversal of all children of *snode* from left to right. During the traversal, if a subject tree node *u* matches a frontier node *s*, satisfying both tag-name and value constraints, it recursively matches the subtrees rooted at *u* and *s* (line 8–9). If the whole subtrees match, *s* should not be considered as a candidate for matching subsequent subject tree nodes and its following-siblings in the pattern tree should be inserted into the frontier set if they qualify when deleting *s* and its incident arcs (line 10–14). The rest of the pseudo-code cleans up the resulting list *R* if only part of the pattern tree was matched when traversing the children of *snode* is exhausted—FOLLOWING-SIBLING returns NIL in line 16.

Note that Algorithm 2 accesses subject tree nodes in a depth-first manner. This means that subject tree nodes are accessed in the document order. This property is crucial to the proof of Theorem 3.3 given in Section 3.6.

Example 3.2 Consider the subject tree in Figure 3.6 and the NoK pattern tree in Figure 3.5b with tag names properly translated (`b[c/g="Stevens"][j<100]`). Suppose the starting point *snode* is the first `b` in the subject tree, which matches the *proot* and is appended to *R*, the algorithm iterates over `b`’s children to check whether they match with any node in the frontier set $\{c,j\}$. When the third child of *snode* matches with `c`, a re-

Algorithm 2 NoK Pattern Matching

NOK-MAIN($pnode, snode$)

```

1  $R \leftarrow \emptyset$ ;
2 if ROOT-LOOKUP( $pnode, snode$ ) then
3     NPM( $pnode, snode, R$ );
4 end
5 return  $R$ ;

```

ROOT-LOOKUP($pnode, snode$)

```

1 if label( $pnode$ )  $\neq$  label( $snode$ ) then
2     return false;
3 end
4 while  $pnode$  is not the root of pattern tree do
5      $pnode \leftarrow pnode.parent$ ;
6      $snode \leftarrow PARENT(snode)$ ;
7     ROOT-LOOKUP( $pnode, snode$ );
8 end
9 return true;

```

NPM($pnode, snode, R$)

```

1 if  $pnode$  is the returning node or join node then
2     construct or update a relation for the candidate result;
3     append  $snode$  to  $R$ ;
4 end
5  $S \leftarrow$  all frontier children of  $pnode$ ;
6  $u \leftarrow FIRST-CHILD(snode)$ ;
7 repeat
8     for each  $s \in S$  that matches  $u$  with both tag name and value constraints do
9          $b \leftarrow NPM(s, u, R)$ ;
10        if  $b = true$  then
11             $S \leftarrow S \setminus \{s\}$ ;
12            delete  $s$  and its incident arcs from the pattern tree;
13            insert new frontiers caused by deleting  $s$ ;
14        end
15    end
16     $u \leftarrow FOLLOWING-SIBLING(u)$ ;
17 until  $u = NIL$  or  $S = \emptyset$ 
18 if  $S \neq \emptyset$  and  $pnode$  is a returning node then
19     remove all matches to  $pnode$  in  $R$ ;
20     return false;
21 end
22 return true;

```

recursive call is invoked to match the NoK pattern $c/g="Stevens"$ with the subtree rooted at $snode/c$. When the recursive call returns `true`, the algorithm continues to check the other children and eventually j is matched, causing the frontier set to be \emptyset . After that, the result R contains the starting point b . \square

Complexity It is clear from the algorithm that every $snode$'s child will be visited exactly once, but in some special cases, its grandchildren (and great-grandchildren and so on) could be visited multiple times through multiple recursive calls. For example, in $/a[b/c][b/d]$, a has two children b 's and they should be both in the frontier when a is matched with $snode$. Since every $snode/b$ node matches both b 's in the frontier, two recursive calls will be invoked to match the two branches (b/c and b/d), so every grandchild of $snode$ will be visited exactly twice for matching c and d . In the worst case, there will be $|S|$ recursive calls at each level (when all frontier nodes match with the current node of subject tree). Assume there are l levels in the pattern tree, s_i and p_i denote the number of nodes at level i in the subject tree and pattern tree, respectively, the maximum number of recursive calls at each level will be $O(s_i \cdot p_i)$. The worst case complexity of the whole algorithm is simply the sum of the number of recursive calls at each level $\sum_{i=1}^l O(s_i \cdot p_i) = O(mn)$, where $\sum_{i=1}^l s_i = m$ and $\sum_{i=1}^l p_i = n$, and m and n are the number of nodes in the pattern tree and subject tree, respectively.

3.5 Physical Storage

The design desiderata for the native physical storage scheme are listed as follows:

1. The XML structural information (subject tree) should be stored separately from the value information. The reason for this is explained in Section 3.5.1.
2. The subject tree should be “materialized” to fit into the paged I/O model. Here materialization means the two-dimensional tree structure should be represented by a one-dimensional “string”. The materialized string representation should be as succinct as possible, yet still maintain enough information for reconstructing the tree structure. The justification for this is given in Section 3.5.2.

3. The storage scheme should have enough auxiliary information (e.g., indexes on values and tag names) to speed up NoK pattern matching.
4. The storage scheme should be adaptable to support updates.

The subsequent two subsections introduce how to manage the value information and structural information, respectively.

3.5.1 Value Information Storage

The first issue in the desiderata is based on two observations: Firstly, an XML document is a mixture of schema information (tag names and their structural relationships) and value information (element contents). The irregularity of contents (variability of lengths) makes it hard (inefficient) for the query engine to search for certain schema/content information. Secondly, any path query can be divided into two subqueries: pattern matching on the tree structure and selection based on values. For example, the structural constraints and value constraints in the path expression in Figure 3.5b are `//book[author/last][price]` and `last="Stevens" ^ price<100`, respectively. The final result could be joined by the results returned by the subqueries. Separating the structural information and the value information allows us to separate different concerns and address each appropriately. For example, a B⁺ tree can be built on the value information and a path index (suffix tree, for example) can be built on the structural information without worrying about the other part.

After the separation, one needs to somehow maintain the connection between structural information and value information. Dewey ID [73] is used here as the key of tree nodes to reconnect the two parts of information, e.g., the Dewey IDs of the root `a` and its second child `b` are 0, and 0.2, and so on. The reason to use Dewey ID instead of giving each node a permanent ID is that Dewey ID contains the structure information and can be derived automatically during the tree traversal. That eliminates the need to keep the ID information in the tree structure (cf. Section 3.5.2). Given a Dewey ID, another B⁺ tree keyed on Dewey ID can quickly locate the value of the node in the data file. This data file and its auxiliary data are shown in Figure 3.7.

Each text node also has a Dewey ID. For example, in the following

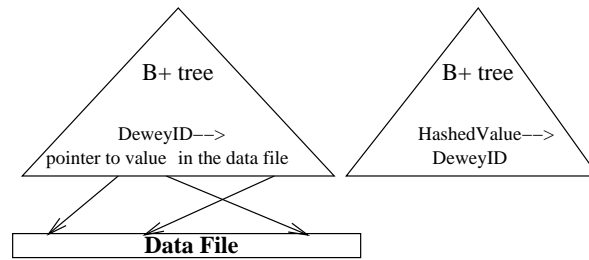


Figure 3.7: Data file and auxiliary indexes

```
<section> text1 <figure @url="figure.pdf"/> test2 </section>
```

the `section` element two text nodes which are separated by a `figure` element. Suppose the Dewey ID of `section` is 1, then the Dewey IDs for the two text nodes are 1.1 and 1.3, respectively. The Dewey ID and value pairs of the text nodes are stored sequentially in a *data file* ordered by their Dewey IDs. To evaluate the value-based constraints efficiently, a B⁺ tree is constructed on the data file keyed on the *hashed* data values and a lookup in the index will return a set of Dewey IDs whose nodes contain that value. The purpose of the hash function here is to map any data value (could be variable length string) to an integer that can be compared quickly. Different values that are hashed to the same key can be distinguished by looking up the data file directly. Careful selection of the hash function would significantly reduce this type of conflict.

Example 3.3 In the data file, each element content could be represented by a binary tuple $(len, value)$, where len is the length of the value. The value information for the XML document in Figure 3.5a can be organized as a list of records: (4, “1994”), (18, “TCP/IP Illustrated”), (14, “Addison-Wesley”), (7, “Stevens”), (5, “65.95”), and so on. The position of these records in the data file are kept in the Dewey ID B⁺ tree. If there are more than one node with the same value, only one copy is kept and all nodes point to the same position in the data file. □

If the XML file is updated, the value can be easily appended to the end of the data file. However, both indexes need to be updated. The value-based B⁺ tree can be updated

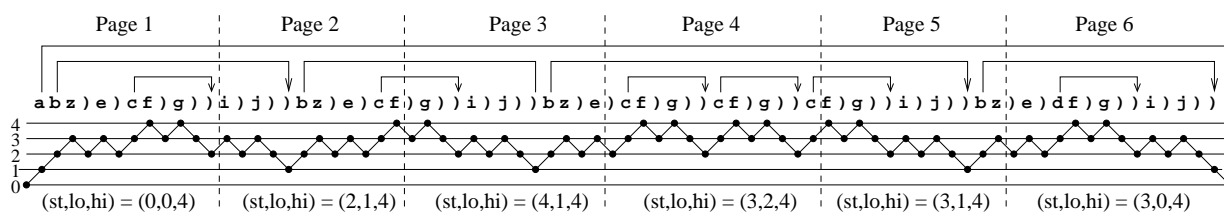


Figure 3.8: The string representation of an XML tree

incrementally based on insertion/deletion of keys. Due to the nature of Dewey IDs, the node ID B⁺ tree may need to be reconstructed if many IDs have been updated.

3.5.2 Structural Information Storage

One way to materialize the tree is to store the nodes in pre-order and keep the tree structure by properly inserting pairs of parentheses as introduced in [89]. For example, `(a(b)(c))` is a string representation of the tree that has a root `a` and two children `b` and `c`. The “(” preceding `a` indicates the beginning of a subtree rooted at `a`; its corresponding “)” indicates the end of the subtree. It is straightforward that such a string representation contains enough information to reconstruct the tree. However, it is not a succinct representation because each node (a character in the string) actually implies an open parenthesis. Therefore, all open parentheses can be safely removed and only closing parentheses are retained as in `ab)c)`. Note that this representation can be further compressed by replacing any series of closing parentheses with a number indicating how many of those closing parentheses there are. However, this introduces the difficulty that it is unknown how many bits are needed for encoding the number, unless the XML document is parsed beforehand. However, parsing beforehand is impossible in the context of streaming XML where we have no knowledge of the upcoming events (closing tag or deeper nesting). Thus the closing parentheses “)” is kept.

Example 3.4 Figure 3.8 shows the string representations of the subject tree in Figure 3.6 (At the physical level, the pointers in the figure are not stored. They only serve to easily identify the end of a subtree to the reader.). If the string is too long to fit in one page,

it can be broken up into substrings at any point and stored in different pages. Assume each character in Σ is 2 bytes long, “)” is 1 byte long, and each page is 20 bytes long (the number is chosen for illustration only), the string can be divided into six pages separated by the dashed lines in the figure. \square

Extra information is stored in each page to speed up the query process. The most useful information for locating **children**, **siblings** and **parent** is the node level information, i.e., the depth of the node from the root. For example, assuming the level of the root is 1 in Figure 3.8, the level information for each node is represented by a point in the 2-D space under the string representation (the x -axis represents nodes and the y -axis represents level). For each page, an extra tuple (st, lo, hi) is stored, where st is the level of the last node in the previous page, lo and hi are the minimum and maximum levels of all nodes in that page, respectively (Note that st could be outside the range $[lo, hi]$). This tuple can be thought of as a feather-weight index for guessing the page where the following sibling or parent is located. Its usage is introduced in Section 3.6. In order to expedite the speed for loading the page headers, all $[lo, hi]$ tuples can be extracted from the pages into a separate file, so that it is not necessary to scan all pages to find the header at the query processing phase.

Note that when streaming XML (e.g., SAX events) are parsed so that every open tag of an element is translated to a character in Σ and every closing tag is translated to a “)”, the result is exactly the same as the NoK physical string representation. Therefore, the single-pass NoK pattern matching algorithm (presented in Section 3.6) based on this physical string representation can be adapted to the streaming XML context, except that page headers (which help to skip page I/O’s) are not necessary in the streaming context since each page needs to be read into main memory anyway.

In addition to these advantages, it is also fairly easy to insert and delete nodes from the string representation of the tree. Attaching a subtree to a leaf can be done by inserting the string representation of the subtree between the left character and its corresponding “)”. For example, if $ab)c))$ is inserted as a subtree of the first f node in page 1, one can allocate a new page with the content $ab)c))$, cut-and-paste the content after f in page 1 to the end of content of the new page, and insert the new page into the page link between pages 1 and 2. The (st, lo, hi) information for page 1 should be changed accordingly.

Inserting a subtree to a non-leaf node is slightly more complicated. For example, if **a** is inserted in between the root **a** and its second child **b**, this requires the insertion of an additional “)” after the rightmost descendant of **b**. This can be handled by controlling the load factor of each page, thereby reserving some of the page for insertion and by keeping a next page pointer in the header in case a new page is inserted. The page layout is shown in Figure 3.9.

According to the page layout, the number of nodes in each page can be calculated easily: assume that each page is 4KB, of which 20% of the space is reserved for update; each character in Σ is 2 bytes long and “)” is 1 byte long. Then each node occupies 3 bytes (because each node consists of a character in Σ and a “)” character); each parameter in the vector (st, lo, hi) occupies 1 byte, and the page index occupies 4 bytes. Consequently, the number of nodes in a page is around 1000. This number is called the *capacity*, \mathcal{C} , of the page. It can be calculated by the formula: $\mathcal{C} = \frac{B \times (1-r) - V - I}{S + P}$, where B is the page size, r is the ratio for space reserved for update, V is the size of vector (st, lo, hi) , I is the page index length, S is the length of character in Σ , and P is the length for encoding of “)”. As calculated above, the value of \mathcal{C} is around 1000 to 3000 by substituting reasonable values to these parameters. The experiments show that the string representation of the tree structure is only about 1/20 to 1/100 of the size of the XML document.

Now assume that the subject tree has 10 billion nodes (the size of the original XML document is about 200GB to 1TB according to the statistics), then one needs about 3 to 10 million pages to store the string representation of the tree structure. If loading the page headers (assuming each is 7 bytes long) to main memory, the system only needs 21 to 70 MB. In modern computer systems, this is reasonably small for handling up to 1 terabyte of data.

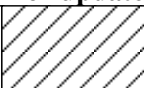
header	string representation	reserved for update
(st,lo,hi)	abz)e)cf)g))i)j))bz)e)cf	
nextpage		

Figure 3.9: Page layout for structural information

The subsequent section answer the natural question: if the header information is loaded into main memory beforehand, how does it help in speeding up the path queries?

3.6 Physical Level NoK Pattern Matching

In the NoK pattern matching algorithm (Algorithm 2), the only operation on the subject tree is the iteration over children of a specific node in their document order. Using the physical storage technique proposed in the previous section, this operation is divided into two primitive operations: `FIRST-CHILD` to find the first child of a node, and `FOLLOWING-SIBLING` to find the following sibling of a node. The physical level NoK pattern matching algorithm simply uses the physical level `FIRST-CHILD` and `FOLLOWING-SIBLING` operations to perform the iteration in lines 6 and 16 in Algorithm 2.

According to the pre-order property of the string representation, these two operations can be performed by looking at the node level information of each page from left to right without reconstructing the tree structure. The basic idea is illustrated in the following example.

Example 3.5 Consider the string representation in Figure 3.8. Suppose the query is to find the first child of character `b` in the first page. Since the nodes are pre-ordered, the first child of `b` must be the next character if it's not a `)`. This condition is equivalent to saying that the first child of a node at level l is the next character if its level is $l + 1$. In Figure 3.8 the answer is `b`'s immediate right neighbor `z`.

Now, suppose the query wants to find `b`'s following sibling. Again, since the nodes are pre-ordered, the following sibling must be located to the right of `b` in the string and its level must be the same. Moreover, the target character must not be too far to the right since, in this case, it could be `b`'s cousin (share the same grandparent but not the parent). Therefore, there must be another constraint: no intermediate character (i.e., cousin) whose level is 2 less than `b`'s level should be in the string between `b` and `b`'s following sibling. In Figure 3.8, the answer is `b` in page 2, but there is no following sibling of `j` in the second page. □

Given a page, it is straightforward to calculate the level information for each node: initially the level is set to `st` in the page header (`st` in the first page is always 0), the

string is scanned from left to right, if the character is in Σ , its level is incremented by 1, otherwise (i.e., a closing parenthesis) its level information is decremented by 1. For example, the levels for the nodes in the first page are 0123232343432.

Algorithm 3 gives a straightforward implementation of the FIRST-CHILD and FOLLOWING-SIBLING operations.

The READ-PAGE subroutine reads a page from disk to main memory and calculates the level information described above. It takes the page number p as the input parameter, and returns the page content and level information to the next two parameters \mathbf{A} and \mathbf{L} , which are two-dimensional arrays, where $\mathbf{A}[p]$ and $\mathbf{L}[p]$ are strings (e.g., `abz)e)cf)g))` and 0123232343432 for page 1) representing the content and level information of page p , respectively.

The FIRST-CHILD and FOLLOWING-SIBLING subroutines call READ-PAGE to read a page and calculate the level information when necessary. They take two parameters p and o that are the page number and the offset in the page, respectively. They check the string representation and level information stored in \mathbf{A} and \mathbf{L} and return a character representing the tag name of first child or following sibling.

The I/O complexity of the FIRST-CHILD is straightforward: two page I/O's in the worst case to get the next character in the string. The FOLLOWING-SIBLING operation may scan the whole file before finding the next character with the same level information. In fact, this is the case for finding the following sibling of root `a` in Figure 3.8. To avoid unnecessary page I/O's, the algorithm should exploit the maximum and minimum level information in each page as described in the page header. The idea is based on the fact that if the current node u with level l has a following sibling, the page that contains this following sibling must have a character “)” with level $l - 1$ (this is the closing parenthesis corresponding to u). If $l - 1$ is not in the range $[lo, hi]$ of a page, it is clear that this page should not be loaded. As described in the previous section, all the page headers can be kept in main memory with very low cost, and greatly reduce the number of page I/O's. In the case of locating `a`'s following sibling, only two page I/O's are needed (pages 1 and 6). This optimization can be easily implemented by modifying the READ-PAGE subroutine in Algorithm 3.

The FIRST-CHILD and FOLLOWING-SIBLING subroutines correspond to the `child` and

Algorithm 3 Primitive Tree Operations

READ-PAGE($p, \mathbf{A}, \mathbf{L}$)

```

1  if page  $p$  is invalid then
2      return false;
3  end
4  if page  $p$  is not in main memory then
5      read page  $p$  in array  $\mathbf{A}[p]$ ;
6      calculate level array  $\mathbf{L}[p]$  for page  $p$ ;
7  end
8  return true;

```

FIRST-CHILD(p, o)

```

1  if READ-PAGE( $p, \mathbf{A}, \mathbf{L}$ ) = false then
2      return NIL;
3  end
4  if  $o = \mathbf{A}.len$  then
5      return FIRST-CHILD( $p + 1, 0$ );
6  elseif  $\mathbf{L}[p][o + 1] = \mathbf{L}[p][o] + 1$  then
7      return  $\mathbf{A}[p][o + 1]$ ;
8  else return NIL;
9  end

```

FOLLOWING-SIBLING(p, o)

```

1   $l \leftarrow \mathbf{L}[p][o]$ ;
2   $j \leftarrow o + 1$ ;
3  while READ-PAGE( $p, \mathbf{A}, \mathbf{L}$ ) = true do
4      while  $j < \mathbf{A}.len$  do
5          if  $\mathbf{L}[p][j] = l - 2$  then
6              return NIL;
7          elseif  $\mathbf{L}[p][j] = l$  and  $\mathbf{A}[p][j] \neq \mathbf{A}[p][o]$  then
8              return  $\mathbf{A}[p][j]$ ;
9          end
10          $j \leftarrow j + 1$ ;
11     end
12      $p \leftarrow p + 1$ ;
13      $j \leftarrow 0$ ;
14 end
15 return NIL;

```

following-sibling axes in a path expression. Other axes (e.g., `parent`, `//` and `following`) can be easily composed by using these two operations. For example, given a node u in the string representation, its descendants are those characters located in between u and its following sibling (more precisely it should be all characters in between u and its first right-side character whose level is $level(u) - 1$). This implies that the interval $\langle p_1 * \mathcal{C} + o_1, p_2 * \mathcal{C} + o_2 \rangle$, where p_1, p_2, c_1, c_2 are the page number (p_i) and offset (c_i) of a character and its corresponding “)”, respectively, can be used in the condition for structural joins just as in the interval encoding approach.

Theorem 3.3 *Given a string representation of the subject tree \mathcal{S} and a NoK pattern tree \mathcal{P} , suppose the maximum number of descendants of the second level nodes (e.g., nodes labeled with \mathbf{b} in Figure 3.6) in \mathcal{S} is n . The physical level NoK pattern matching algorithm reads every page at most once (single-pass), and requires only n/\mathcal{C} pages in main memory (where \mathcal{C} is the capacity of the page).*

PROOF From the analysis of Algorithm 2, it is known that in a special case, the algorithm might access a subject tree node u more than once if $level(u) > 2$. In the NoK physical storage scheme, the descendants of u are stored before its following sibling. Since Algorithm 2 matches subject tree nodes in a depth-first manner (matches all of u 's descendants first before following sibling and never reads back), in the worst case one only needs to read all the pages that contain u 's descendants in main memory, which requires a buffer size of n/\mathcal{C} pages, and match them against all pattern tree branches. After the FOLLOWING-SIBLING is called, this buffer can be freed and those pages are read only once. ■

Since usually XML files are flat and the page capacity is around 1000, the number of pages needed in main memory is small in practice.

3.7 Experimental Evaluation

To assess the effectiveness of the proposed approach, extensive experiments are conducted and the performance is compared with existing systems or prototypes that are based on interval encoding or other native physical storage schemes. Both the data and the queries

data set	size	#nodes	d_{avg}	d_{max}	tags	tree	$ \mathbf{B}_t^+ $	$ \mathbf{B}_v^+ $	$ \mathbf{B}_i^+ $
author	1.2	15,006	3	3	8	0.035	0.18	0.33	0.4
address	17	403,201	3	3	7	0.5	5	12	11
catalog	30	620,604	5	8	51	1.2	8	15	13
TreeBank	82	2,437,666	8	36	250	5.3	58	80	72
dblp	133	3,332,130	3	6	35	8	62	150	180

Table 3.1: Statistic information of data sets . Note: d_{avg} and d_{max} represent average and maximum depths, respectively. The sizes of data set, |tree|, $|\mathbf{B}_t^+|$, $|\mathbf{B}_v^+|$, and $|\mathbf{B}_i^+|$ are all in MB.

are classified into categories so that the efficiency of all approaches in different environments can be tested.

3.7.1 Experimental Settings

The algorithms and physical storage prototype are implemented in Java with JDK 1.4. All the experiments were conducted on a PC with Pentium III 997MHz CPU, 512MB RAM, and 40GB hard disk running Windows XP.

Both synthetic and real data sets are tested. The synthetic data sets (author, address, and catalog) are selected from the Xbench benchmark [136] in the data-centric category. The real data sets (Treebank and dblp) are selected from University of Washington XML Data Repository [7]. These data files are selected because they are either bushy (author, address, dblp) or deep (catalog, Treebank). Table 3.1 shows the statistical information of the data sets and B^+ tree indexes, in which **tree**, \mathbf{B}_t^+ , \mathbf{B}_v^+ , \mathbf{B}_i^+ denote the string representation of the tree structure, the B^+ trees for tag names, values, and Dewey IDs, respectively.

Queries were carefully selected for the experiments to cover different aspects of path queries on the XML data. The selection is based on the following three properties of path expressions:

Selectivity: A path expression returning a small number of results should be evaluated faster than those returning a large number of results. To evaluate whether the NoK

Query	Category	Example query
Q1	hpy	/a/b[c="hi"]
Q2	hpn	/a/b/c/d
Q3	hby	/a/b[c="hi"] [d="hi"]/e
Q4	hbn	/a/b[c] [d] [e] [f]
Q5	mpy	/a/b[z="mod"]/d/e
Q6	mpn	/a/b/e
Q7	mby	/a/b[c="mod"] [d="mod"]
Q8	mbn	/a/b[c] [d] [e]
Q9	lpy	/a/b[c="low"]/d
Q10	lpn	/a/b/c
Q11	lby	/a/b[c="low"] [d="low"]
Q12	lbn	/a/b[c] [d]

Table 3.2: Query categories

pattern matching algorithm is sensitive to selectivity, queries are divided into three categories based on their selectivity: high (several results), moderate (greater than 10 but less than 100 results), and low (greater than 100 results).

Topology: The shape of the pattern tree could be a single-path or bushy (two or more leaf nodes) and may contain `//`-arcs. Some systems may have different performance in these cases, but the I/O cost of the NoK algorithm should be the same, except that the main memory operations in the bushy case could be greater.

Value constraints: The existence of value constraints and index on values may be used for fast locating the starting point for NoK pattern matching, especially when the selectivity is high. Therefore, queries having value constraints may be used to justify the effectiveness of value-based indexes.

With these three criteria, queries are classified in twelve categories shown in Table 3.2. Each category is denoted by a string of length three, where each position denotes one of the above criterion. The character in each position stands for: low (l), moderate (m), or high (h) for selectivity; path (p), or bushy (b) for topology; and yes (y), or no (n) for

existence of value constraints. The tag names and constants in the example queries are dummy and they should be replaced by appropriate values in different test files. Queries with // are also tested by randomly substituting // for a / axis.

3.7.2 Performance Evaluation and Analysis

The NoK pattern matching algorithm is tested against two join-based algorithms based on interval encoding—dynamic interval (DI) [48] and TwigStack [28], as well as a state-of-the-art native XML database system X-Hive/DB version 4.1.1. For each data set, a representative path expression is chosen in each of the twelve categories. The performance evaluation results are shown in Table 3.3. Each running time is the average over three executions. Some categories are not applicable (denoted as “NA” in the table) to the data sets (e.g., author and address data sets do not have moderate selectivity queries without value constraints), or the selected queries for the category contain some features that were not implemented (denoted as “NI” in the table) by a particular system (e.g., DI does not support value comparisons other than equality as of the date the experiments were performed).

All the indexes (ID, tag-name, and value) are created for the data sets. To conduct fair comparisons, indexes (ID, tag-name, and value) for X-Hive are also created. The TwigStack algorithm is implemented in a way such that different tree nodes with different tag names are stored separately in a file sorted by document order. Each file contains the nodes constituting an input stream associated with a node in the twig. In order to speed up value comparisons, a B⁺ tree for the value nodes is also created. DI has only limited support for tag-name index at this time, so the tests do not use index for DI. This is one of the reasons that DI did not perform as well as other systems. A very simple rule-based heuristic is applied to choose which index to use:

1. Whenever there are value constraints, the value index is always chosen to locate the starting point for NoK pattern matching.
2. If there are more than one value constraints, the most selective one is chosen. The selectivity is estimated by “peeking” at the cardinalities of distinct values maintained in the value-based B⁺ tree index.

file	systems	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
author	DI	0.25	0.24	0.6	N/A	0.24	N/A	NI	N/A	NI	0.29	NI	0.87
	X-Hive	0.34	0.28	0.45		0.38		0.48		0.78	0.43	0.85	0.23
	TwigStack	0.25	0.21	0.28		0.34		1.5		1.6	0.38	2.3	0.48
address	NoK	0.2	0.23	0.29		0.3		0.29		0.58	0.17	0.69	0.17
	DI	2.53	2.6	6.6	N/A	2.49	N/A	8.9	N/A	2.57	2.73	NI	4.81
	X-Hive	0.3	1.7	0.47		1.03		0.72		0.79	2.3	0.87	4.73
catalog	TwigStack	0.3	0.31	0.26		1.2		1.74		1.79	1.9	101	4.2
	NoK	0.32	0.05	0.27		0.83		0.77		1.5	3.2	1.39	3.8
	DI	8.8	3.25	14	N/A	9.4	N/A	NI	N/A	8.9	2.87	NI	13
Treebank	X-Hive	1.3	2.16	0.67		1.4		1.4		1.7	4.9	1.6	5.4
	TwigStack	1.01	1.2	0.81		2.01		1.3		0.8	0.9	2.4	1.1
	NoK	0.37	0.1	0.43		1.2		1.4		2.5	1.1	2.4	1.3
dblp	DI	26.3	11.7	45.5	13	N/A	27.4	N/A	43	N/A	14.2	N/A	43.6
	X-Hive	0.61	8.2	0.6	7.2	19.7		14		2.9			14.2
	TwigStack	0.8	18.3	0.45	20.1	1.62		1.9		1.8			4.2
dblp	NoK	0.35	0.77	0.37	0.74	0.51		0.65		2.1			1.2
	DI	27	18.5	17.5	18.3	60.8	26	18.8	17.6	17.4	18	75.2	26.6
	X-Hive	0.97	19.4	12	11.36	10.8	8.6	12.9	8.85	2.78	16.2	9.7	10.7
dblp	TwigStack	2.19	1.2	7.3	2.2	8.9	0.32	9.2	0.45	1.2	1.4	12.9	13.8
	NoK	0.6	0.1	2.8	3.2	1.22	0.57	10.8	0.5	1.5	0.89	0.62	0.99

Table 3.3: Running time (in sec) for DI, X-Hive, TwigStack, and NoK on different queries and data sets

3. If there are no value constraints and the selectivity of the query is “high”, the tag name which has the highest selectivity is chosen as the key to search in the tag-name index.
4. If all the above rules do not apply, a sequential scan is used.

This heuristic allows us to test the effectiveness of value and tag-name indexes. Experiments show that sometimes a value index is more effective than a tag-name index (e.g., in Treebank high selective categories), and sometimes a tag-name index is more effective (e.g., in catalog). This is because values in Treebank were randomly generated and has higher selectivity than tag names. Without doubt, this heuristic may not choose the optimal method. A cost model that estimates the cost of each method based on pre-collected statistics is more desirable. In Chapter 5, one of the most important components of cost-based optimization, cardinality estimation, is presented; and Chapter 6 introduces how to use the cardinality estimation technique to modeling cost using an established cost modeling technique.

Another reason for the good performance of NoK (as well as TwigStack) is that it does not need to materialize intermediate results for multiple joins. Materializing intermediate results or recomputing partial results is inevitable in bushy path expressions for DI. For example, in the path expression $/a/b[c][d]/e$, element b needs to be tested for children c and d , and then return children e . Each of the three operations need a join with nodes returned by $/a/b$, while in a single-path query, DI could use a pipelined plan and avoid materialization. Therefore, DI is topology sensitive, but NoK is not, as shown in Table 3.3.

Moreover, since both materialization and re-computation are expensive operations when the intermediate result is large, DI has to perform the same amount of work disregarding the result size, i.e., DI is not sensitive to the selectivity. Generally, the running time of NoK decreases when the selectivity of the *starting points* increases. The experiment shows that the selectivity of starting points can be a fairly good approximation for selectivity of final results if the most selective index (value or tag-name-based) is chosen for locating the starting points.

In comparison to the TwigStack algorithm, the NoK algorithm performs fewer fruitless scans because it does not need to traverse a subtree if its root does not match a pattern tree node. However, TwigStack has to scan all streams associated with leaf nodes in the

pattern tree. Although XB-tree [28] or other index structures [82] might compensate for this problem, the storage basis (interval encoding) lacks the flexibility for update and for processing streaming XML.

In summary, the NoK system is comparable to DI, TwigStack, and X-Hive in some cases and outperforms them in most cases. In particular, NoK is sensitive to the selectivity, insensitive to the topology of pattern tree, and could take advantage of the existence of value constraints.

3.8 Comparisons to Related Work

Rewriting backward axes to forward axes makes possible an one-pass evaluation algorithm. This is particularly important for streaming XML processing [21] where the input XML document is scanned once. Barton et al. [21] proposed a technique to convert an X-tree (which is a representation of XPath path expression) consisting of forward and backward axes (parent and ancestor) to an X-dag which consists of only forward axes. The translation is simple: each edge labeled with a backward axis is replaced by a reversed edge labeled with a corresponding forward axis (parent to child, and ancestor to descendant), and a descendant labeled edge is introduced to any vertex that has no incoming edge. For example, the path expression `/descendant::n/parent::m` is rewritten to `/descendant::m[child::n == /descendant::n]`. This paper, however, focuses on only four axes: ancestor, parent, child, and descendant.

Olteanu et al. [106] proposed a comprehensive set of rewrite rules for converting *all* backward axes to forward axes. The basic idea is the same as above: rewriting a backward axis to its corresponding forward axis, and adding a descendant edge from the root to any vertex without an incoming edge. Moreover, this paper introduces more rules for special cases. For example, the earlier path expression `descendant::n/parent::m` can be rewritten to a simpler expression: `/descendant-or-self::m[child::n]`.

The rewriting techniques introduced in this thesis complement previous techniques [21, 106] in that the forward axes after the initial rewriting can be further rewritten to a minimum subset. This can further reduce the complexity of the processing algorithms.

In general, tree pattern matching can be classified into the ordered tree pattern match-

ing (OTPM) problem and the unordered tree pattern matching (UTMP) problem depending on the ordering of siblings in the pattern tree. The OTPM problem can be solved very efficiently— $O(n\sqrt{m} \text{ polylog } m)$, where m and n are respectively the sizes of pattern tree and data tree, by using suffix trees or other data structures [77, 50]. However the pattern tree generated by path expressions are generally unordered since branchings in the pattern tree are caused by multiple predicates that are unordered. To be more precise, partially ordered pattern tree should be considered since two nodes can be connected by the **following-sibling** or **preceding-sibling** axes. The UTPM problem is generally tackled by the join-based approach introduced in Chapter 2. Early implementations follow the formal semantics and treat a path expression as a sequence of steps, each of which takes input from the previous step and produces an output to the next step. This can be thought of as a special case of join-based approach that uses nested-loop join instead of merge join-like algorithms. Experimental results show that implementations following this approach suffer from exponential runtime in the size the of path expressions in the worst case [66]. The XNav algorithm [21, 84] has similar features as NoK pattern matching algorithm does: they both are single-pass algorithms and both are suitable for streaming XML processing. The XNav processor also support a fragment of path expressions that only contain **child**, **parent**, **ancestor** and **descendant** axes, as well as value constraints. XNav algorithm is very similar to NoK pattern matching in the streaming XML context, but these two algorithms are based on different storage systems. Both XNav and NoK can be considered as dynamically converting a pattern tree to a finite state automata. But since XNav support global axes, the implementation of XNav navigational operator is more complex than that of NoK.

As introduced in Chapter 2, a number of storage systems have been proposed, including using flat file systems (e.g., Kweelt [117]), extending mature DBMS technologies such as relational DBMSs or object-oriented DBMS [93], and building native XML repositories (e.g., Tamino [120], Natix [56], X-Hive, and Xyleme). Natix [56] is a well-known native storage system. It fits into the paged I/O model by partition the big tree into subtrees, each of which is small enough to fit into a page. Natix needs extra artificial tree nodes in order to “glue” subtrees together. NoK does not require artificial nodes, therefore the space requirement is slightly smaller. Furthermore, since NoK translates a tree into a

string, which can be cut at any position to fit into a page, NoK is more flexible in that the string in each page is not required to be a subtree. Update in both Natix and NoK are easy since they both result in local update: updating a subtree in the case of Natix and updating substring for NoK. The Arb storage system [90] share many features with NoK as well: the space it uses is comparable to NoK, but it requires more page I/Os in the FOLLOWING-SIBLING operation if the tree is very deep since Arb does not maintain the level information. Interestingly, the Arb storage system can also be extended to the streaming XML context but the Arb processing algorithm requires two sequential scans. The NoK pattern matching algorithm uses only one sequential scan in the worst case since NoK pattern tree is less expressive than the path expression introduced in Arb. The idea of Schknolnick’s tree partition technique [118] is similar to Natix storage, but with theoretical guarantees on optimality. However, Schknolnick’s technique requires DTD or XML schema knowledge and the optimal partitioning algorithm is still exponential in the size of schema (or type tree in their terminology). In contrast, the NoK storage partitioning algorithm is simply the problem of partitioning a string into substrings, whose length are under the page limit. This can be easily performed while parsing the XML documents and storing it into the string. However, optimality in terms of minimizing I/O is not guaranteed. Succinct representations for trees are also studied in the data structure and algorithms community [83, 101]. In these papers, a binary tree is represented by parenthesized string of length $2n + o(n)$ bits to support constant time operations to find the left/right child and the parent. These techniques are, however, based on the RAM model rather than the paged I/O model as the NoK storage is. Therefore, the random and sequential access of data are treated with equivalent cost in the RAM model, while they are different in the paged I/O model.

3.9 Conclusion

In this chapter, a special type of pattern tree—NoK pattern tree, is identified and a novel approach is proposed for efficiently evaluating path expressions by NoK pattern matching. The result of NoK pattern matching greatly reduces the number of structural joins in the later step. NoK pattern matching can be evaluated highly efficiently (only need a single

scan of input data) using the native physical storage scheme. Performance evaluation has shown that the proposed pattern matching algorithm has better or comparable performance than the existing extended-relational (based on interval encoding) and native XML database management systems.

Chapter 4

Feature-based XML Index

4.1 Introduction

The previous chapter presented three ways to locate the starting points for NoK pattern matchings: sequential scan, tag-name index, and value index. Since the NoK pattern matching operation is usually expensive (it needs to scan the whole document in the worst case), indexes are crucial to minimize the number of fruitless NoK pattern matchings and thus to improve the overall query performance. Among the tag-name and value indexes, the former can be thought of as the simplest structural index in that the index discriminates subtrees only by the root node label. In some structure-rich data sets, however, it is possible to have many distinct (non-isomorphic) subtrees with the same root. Therefore, an index with more discriminative structural information will provide more pruning power. Moreover, since values usually exhibit higher degree of heterogeneity than structures [23], value constraints usually entail higher selectivity. Therefore, an index combining both structural and value information is even more desirable. This chapter presents such a unified *feature-based index for XML* (FIX) [147], that handles both values and tree structures.

As discussed in Chapter 2, existing *structural indexes* [99, 88, 87, 131] cluster XML element nodes based on their structural similarity, with the objective of obtaining better locality and, hence, better query performance. While structural clustering is effective for data sets that conform to a regular schema (e.g., an `order` always has an `order_id` and `ship_date`), the index could grow remarkably large for structure-rich data sets. To

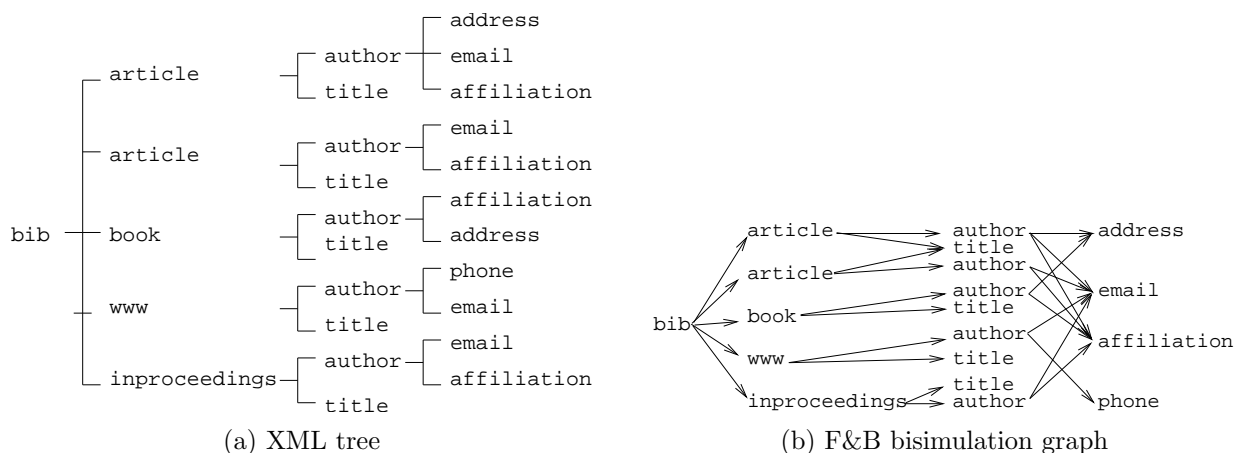


Figure 4.1: An XML document and its F&B bisimulation graph

illustrate the problem, Figure 4.1 shows a bibliography XML document and its clustering index—F&B bisimulation graph [87]. In this data set, all types of publications (**article**, **book**, etc.) have a child element **author**, which may have any combination of subelements **address**, **email**, **phone**, and **affiliation**. Since each **author** element has a different parent or set of children, the **author** elements are incompressible in this particular example. In the case of a structure-rich data set such as Treebank [7], the F&B bisimulation graph has more than 3×10^5 vertices and 2×10^6 edges. Although particular storage structures are developed to materialize F&B bisimulation graphs on disk [131], updating as well as searching in such a large graph could be very expensive. This problem is not unique to the F&B index, but common to all structural clustering techniques.

The key insight of the FIX index is to break a large document into small pieces of substructures (which we call *twig patterns*) to achieve high pruning power without searching the entire graph. A FIX index is constructed by enumerating all twig patterns in the document and mapping each of them into a vector of *structural characteristics* (or *features*). The feature vector is a signature of a twig pattern and serves as a key to record the twig pattern in a mature index such as B⁺ tree. In the query phase, the features of the query are computed and candidate twig patterns that conform to these features can be quickly retrieved from the index without exploring the whole search space. A following refinement step may be required to obtain the final results.

Using this approach, answering a twig query amounts to looking up a vector in the B^+ tree. However, two challenges arise: (1) what are the appropriate set of features of the twig patterns, and (2) how to deal with the fact that the number of twig patterns is exponential in the number of vertices in the graph? These two questions are correlated in that if the number of patterns is small, the index can record all of them and use their string representations (or the hash codes thereof) as the keys. However, in the general case, when the number of patterns is large, the index has to choose a subset of them. In this general case, the string representation of a twig pattern is no longer a valid key, since when index lookup for a query pattern fails, it is unknown whether the pattern is in fact not in the database, or just missed in the index. Accordingly, the critical issue in this approach relies on finding the desired features.

FIX employs a set of features based on spectral graph theory [47, 41]. These features are proven to satisfy the *no-false-negative* requirement: by examining only the query, the index is able to fix a *complete* set of candidate twig patterns that may produce results. The *no-false-positives*, i.e., all candidates produce results, is not as important since this can be handled by a further refinement step. Therefore, FIX is a pruning index that can be built on top of any existing XPath query processor to achieve better query performance.

The rest of the chapter is organized as follows: Section 4.2 provides the background that is specific to this chapter. Section 4.3 introduces the translation of a twig pattern into a matrix and proves certain properties of the eigenvalues of the matrix. Sections 4.4 and 4.5 present the index construction algorithm and index query algorithm, respectively. Section 4.6 present an experimental evaluation of FIX. Comparisons to related work is presented in Section 4.7.

4.2 Background

FIX can handle a subset of path expressions called *twig queries*. The term “twig query” is defined slightly differently in different papers in the literature. The following definition defines the usages of the term in this thesis.

Definition 4.1 (Twig Query) A *twig query* is a path expression whose axes could only be /, except the axis for the first location step could be //. Moreover, there is no KindTest

in the expression and no value-based comparison inside the predicates. \square

The restriction in this definition to exclude $//$ -axes and value predicates is not a limitation of FIX, but makes the presentation easier. Sections 4.4.6 and 4.5 show how to handle value equality conditions and $//$ -axes in the middle. Section 4.5 also explains how to answer queries with **following-sibling** axes, which may be contained in the pattern tree after the rewrite as discussed in Section 3.2.

A twig query can be thought of as a tree in which each step corresponds to a node in the tree, and the first step is connected to a special *root* node. The axes are translated into edges in the tree. Based on the tree representation, the notion of *existential match* (or simply *match*) between a twig query and an XML tree can be defined.

Definition 4.2 (Existential Matches) A twig query Q *matches* an XML tree X if there exists a mapping f from the NameTests of Q to the nodes in X such that the following hold:

- the root of the twig query always matches the document node (parent of the root node in the document).
- for any NameTest $q \in Q$, $label(q) = label(f(q))$.
- if two NameTests u and u' are connected by an axis $\alpha \in \{"/", "//"\}$, then $f(u)$ is a parent (or ancestor) of $f(u')$ if $\alpha = "/"$ (or $"//"$). \square

Match does not specify which XML nodes should be returned, therefore it is used for existential testing.

4.2.1 Bisimulation Graph

Given an XML tree, there is a unique (subject to graph isomorphism) minimum bisimulation graph that captures all structural constraints in the tree. The bisimulation graph defined in this chapter is based on the bisimilarity¹ notion defined by Henzinger et al. [76].

¹*Bisimilarity* is used to denote the relation between XML nodes and index vertices; and *bisimulation* graph is used to denote the resulting index graph after the bisimilarity mappings.

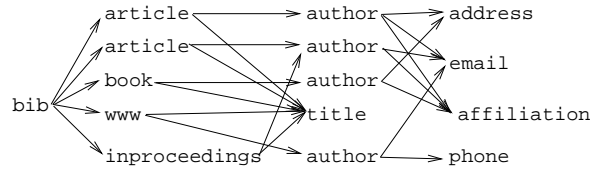


Figure 4.2: A bisimulation graph of the bibliography document in Figure 4.1a

Definition 4.3 Given an XML tree $T(V_t, E_t)$ and a labeled graph $G(V_g, E_g)$, an XML tree node $u \in V_t$ is *bisimilar* to a vertex $v \in V_g$ ($u \cong v$) if and only if all the following conditions hold:

- u and v have the same label.
- if there is an edge (u, u') in E_t , then there is an edge (v, v') in E_g such that $u' \cong v'$.
- if there is an edge (v, v') in E_g , then there is an edge (u, u') in E_t such that $v' \cong u'$.

Graph G is a bisimulation graph of T if and only if G is the smallest graph such that every vertex in G is bisimilar to a vertex in T . □

It is easy to see that the bisimulation graph of a tree is a directed acyclic graph (DAG). Otherwise, if the bisimulation contains a cycle, the tree must also contain a cycle based on the definition.

The bisimulation graph of the XML tree in Figure 4.1a is shown in Figure 4.2. The difference between the bisimulation graph and the F&B bisimulation graph is that the former requires that two nodes in the XML tree belong to the same equivalence class if their subtrees are structurally equivalent. The bisimulation graph does not require that the two indexing vertices have similar ancestors, but the F&B bisimulation graph does. Consequently, the bisimulation graph clusters the two `author` vertices from `book` and `inproceedings` into one equivalence class.

The tree representation of a twig query can always be translated into a bisimulation graph, which is called *twig pattern*. Similarly to the twig query, matching twig patterns can also be defined on a bisimulation graph of an XML tree.

4.2.2 Matrices and Eigenvalues

An undirected unlabeled graph G with n vertices can be represented as an $n \times n$ matrix (e.g., adjacency matrix or Laplacian matrix). Given an $n \times n$ matrix \mathbf{M} , there exists a column n -vector \mathbf{v} such that

$$\begin{aligned}\mathbf{M} \cdot \mathbf{v} &= \lambda \mathbf{v} \\ \langle \mathbf{v}, \mathbf{v} \rangle &= 1\end{aligned}$$

where λ is a scalar, and $\langle \mathbf{v}, \mathbf{v} \rangle$ is the inner product of two vectors, which is defined as $\langle \mathbf{v}, \mathbf{v} \rangle = \mathbf{v}^T \mathbf{v} = \sum_{i=1}^n v_i * v_i$, for $\mathbf{v} \in \mathbb{R}^{n \times 1}$; or $\sum_{i=1}^n \bar{v}_i * v_i$ for $\mathbf{v} \in \mathbb{C}^{n \times 1}$, where \bar{v}_i is the complex conjugate operator. The \mathbf{v} and λ are called the normalized eigenvector (or simply eigenvector) and eigenvalue of \mathbf{M} , respectively. The eigenvectors need to be normalized since otherwise there are an infinite number of eigenvalues that are obtained by scaling the eigenvectors. For an $n \times n$ matrix, there are a total of n such eigenvector and eigenvalue pairs, but they may not be distinct. The eigenvalues are usually denoted by $\lambda_1, \dots, \lambda_n$, ordered by their magnitude in descending order. Throughout the rest of this chapter, λ_{max} and λ_{min} denote the maximum and minimum eigenvalues, respectively, and $\lambda_i(G)$ denotes the i^{th} eigenvalue of the matrix representation of G whenever there is no possibility of confusion.

There is a well-know property about two graphs and their eigenvalues [26], and it is the basis for the structural feature selection.

Theorem 4.1 *Let G and H be two undirected unlabeled graphs, and \mathbf{M}_G and \mathbf{M}_H be their adjacency matrices. If H is an induced subgraph of G , then $\lambda_{min}(\mathbf{M}_G) \leq \lambda_{min}(\mathbf{M}_H) \leq \lambda_{max}(\mathbf{M}_H) \leq \lambda_{max}(\mathbf{M}_G)$. \square*

Section 4.3.3 presents a proof to a similar theorem for labeled directed graphs after a certain translation from the graph to matrix.

4.3 Features and Their Properties

Given a twig pattern (labeled directed graph), it is desirable to identify the distinctive characteristics of the structures contained in it. These characteristics are called *features*

of the pattern. Features can be used as a *key* to index and retrieve those instances that match a pattern. In FIX, the features are based on a subset of eigenvalues of the matrix representation of a pattern. Eigenvalues have the desired property that they enable the pruning of the index search space without losing any results.

4.3.1 Structure Preservation

Before discussing the extraction of features, it is important to understand how to use a bisimulation graph to test the existence (match) of a pattern. This is necessary because bisimulation graphs are the input to calculating the features — eigenvalues. The following theorem guarantees that the match of a twig pattern on a bisimulation graph is equivalent to the match of its twig query on the XML tree. In other words, bisimulation graph preserves all the structural information required for existential matching. The reason for using twig patterns and bisimulation graphs rather than their corresponding tree structures is that the trees contain many structural repetitions and are too large to extract their features (eigenvalues).

Theorem 4.2 *A twig query Q matches an XML tree X if and only if the twig pattern Q' matches the bisimulation graph X' . □*

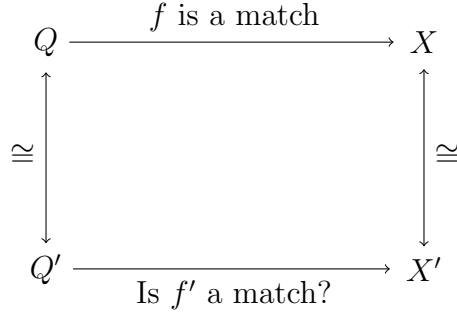
PROOF The proof is quite straightforward after realizing that matching and bisimilarity are homomorphisms on the edge relation.

The sufficient condition:

Given a twig query Q and an XML tree X , assuming their bisimulation graphs are Q' and X' and the mapping $f : Q \rightarrow X$ is a match, one can define a mapping $f' : Q' \rightarrow X'$ and prove that f' is a match as depicted in the following diagram.

Let us first prove that for any two vertices u and v in Q , if u and v are bisimilar, i.e., they are grouped into one vertex in Q' , then $f(u)$ and $f(v)$ are also bisimilar. This actually follows directly from the fact that matching and bisimilarity are homomorphisms on the edge relation. The following is the formal proof.

By definitions of bisimilarity and match, it follows that $label(f(u)) = label(u) = label(v) = label(f(v))$. Furthermore, for any $u', v' \in Q$, if (u, u') and (v, v') are edges in Q , then $(f(u), f(u'))$ and $(f(v), f(v'))$ are edges in X . Since Q' is a bisimulation graph



of Q and $u \cong v$, it follows that $u' \cong v'$. Similarly, since X' is a bisimulation graph of X , it follows that $f(u') \cong f(v')$. Therefore, by definition of bisimilarity, $f(u)$ and $f(v)$ are bisimilar.

Based on the above result, one can define the mapping $f' : Q' \rightarrow X'$ as follows: for any $q' \in Q'$, there always exists $q_i \in Q$ such that $q_i \cong q'$. Based on the previous result, for all such q_i , there is a unique $q'' \in X'$ such that $q'' \cong f(q_i)$. Therefore, let us define $f'(q) = q''$.

Next let us prove that f' is a match from Q' to X' . Given any $q' \in Q'$, $f'(q')$ satisfies the following conditions:

- $label(q') = label(f'(q'))$. This directly follows from the definition of f' .
- for any edge $(p', q') \in Q'$, there is an edge $(f'(p'), f'(q')) \in X'$. This is because the edge relation is preserved in the match and bisimilarity mappings and f' is defined to be the composition of the two mappings.

Therefore, based on the definition of match, f' is a match between Q' and X' .

The necessary condition:

This direction follows from the fact that bisimulation is an onto mapping, i.e., bisimulation graph is the minimum graph such that *every* vertex in Q' has a bisimilarity vertex in Q . The proof is similar to the other direction. ■

This theorem seems contradictory to the fact that the F&B bisimulation graph is the smallest covering index for twig queries [87] and bisimulation graph is smaller than the F&B bisimulation graph. The reason it holds is that here the “structural preservation”

is defined for testing pattern existence (the notion of match) and the “covering” in F&B bisimulation is defined in terms of query answering (which needs more information than existential testing). In fact, the bisimulation graph shown in Figure 4.2 cannot answer the query `//inproceedings[author]` since two authors from `inproceedings` and `book` are grouped into one equivalence class. But this graph is sufficient to answer the existence of `authors` under `inproceedings`.

With the structure preserving property, the twig pattern and bisimulation graph of an XML document can be used as the subject of querying and indexing instead of twig query and XML tree.

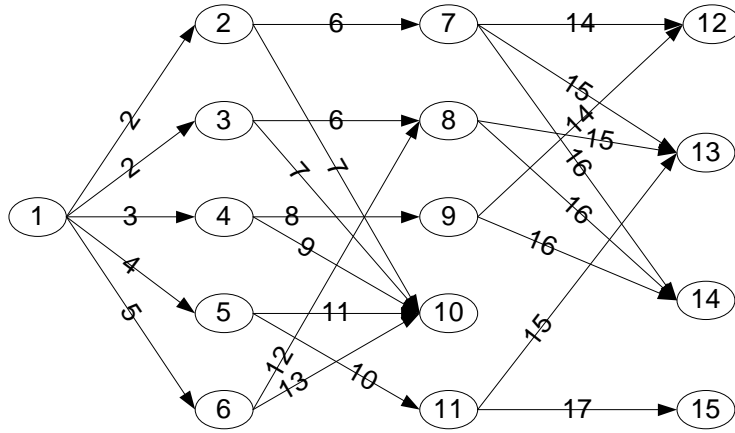
4.3.2 Anti-symmetric Matrices for Twig Patterns

A labeled directed graph (twig pattern) can be translated into a matrix such that the matrix preserves all structural information in the graph. The structural information refers to the labels of the vertices and the edge relations (here the orientations of the edges are important). Ignoring either of them makes the matrix unrepresentative, and, therefore, reduces the pruning power of any method based on this matrix representation.

To record the vertex label information in the matrix, a distinctive weight is assigned to each distinctive edge according to the labels of the two incident vertices. This is a one-to-one mapping, therefore it is always possible to translate the weighted directed graph back to the original labeled directed graph. For example, the following is one possible weight assignment for the bisimulation graph in Figure 4.2.

(bib, article) → 2	(bib, book) → 3	(bib, www) → 4
(bib, inproceedings) → 5	(article, author) → 6	(article, title) → 7
(book, author) → 8	(book, title) → 9	(www, author) → 10
(www, title) → 11	(inproceedings, author) → 12	(inproceedings, title) → 13
(author, address) → 14	(author, email) → 15	(author, affiliation) → 16
(author, phone) → 17		

By keeping the above mapping, the vertex labels can be removed safely without loss of structural information. For example, if the vertices in Figure 4.2 is numbered from one in breadth-first order, the bisimulation graph can be converted to the unlabeled, weighted graph as shown in Figure 4.3a.



(a) unlabeled, weighted graph corresponding to Figure 4.2

$$\mathbf{M} = \begin{bmatrix}
 0 & 2 & 2 & 3 & 4 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 -2 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\
 -2 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\
 \vdots & & & & & & \dots & & & & & & & & \\
 0 & 0 & 0 & 0 & 0 & 0 & -15 & -15 & 0 & 0 & -15 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & -16 & -16 & -16 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -17 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

(b) A 15×15 matrix representation of 4.3a

Figure 4.3: Edge-weighted and matrix representations of bisimulation graph in Figure 4.2

To preserve the direction information, the directed weighted graph is represented as an *anti-symmetric* matrix (a.k.a. *skew-symmetric* matrix [51]) as follows: each vertex v is numbered arbitrarily from 1 to n and it is mapped to a dimension in the $n \times n$ matrix \mathbf{M} . The reason for this is that any assignment can be permuted to some other assignment (and the permutation results in an isomorphic graph), which is equivalent to a permutation of the matrix. It is well known that the eigenvalues of a matrix remain invariant under matrix permutation [47].

If an edge (v_i, v_j) has weight $w_{i,j}$ after the above edge-label-to-integer translation, the mapping assigns $\mathbf{M}[i, j] = w_{i,j}$ and $\mathbf{M}[j, i] = -w_{i,j}$. If (v_i, v_j) is not an edge, $\mathbf{M}[i, j] =$

$\mathbf{M}[j, i] = 0$. In this anti-symmetric matrix, the diagonal elements $\mathbf{M}[i, i]$ are always 0 for an acyclic graph. For example, the matrix representation of the graph in Figure 4.3a is shown in Figure 4.3b. The reason for the negative weight at $\mathbf{M}[j, i]$ is that triangle matrices with all $\mathbf{M}[i, i] = 0$ have the same set of eigenvalues $[0, 0, \dots, 0]$ (matrices having the same eigenvalues are called *isospectral*). A non-zero anti-symmetric matrix is guaranteed to have at least one non-zero eigenvalue [47]. Two anti-symmetric matrices are isospectral, if one can be transformed to the other by a non-singular transformation, that is one anti-symmetric matrix can be obtained by multiplying the other anti-symmetric matrix by a non-singular matrix (a matrix that has an inverse). If it is common that two anti-symmetric matrices are isospectral but non-isomorphic, the pruning power will be small. Given that the number of distinct edge label encodings is small in most XML databases and given the requirement of $\mathbf{M}[i, j] = -\mathbf{M}[j, i]$ for anti-symmetric matrices, the probability of two anti-symmetric matrices being isospectral but non-isomorphic is expected to be very small.

4.3.3 Eigenvalue Containment Property

Given the pairs of λ_{min} and λ_{max} of two anti-symmetric matrices, the similar result to Theorem 4.1 is proven as follows.

Theorem 4.3 *Let G and H be two DAGs, and \mathbf{M}_G and \mathbf{M}_H be the anti-symmetric matrix representations of G and H respectively. If H is an induced subgraph of G (which means H is isomorphic to a subgraph of G with the isomorphic mapping f and for every edge (u, v) in H , there is an edge $(f(u), f(v))$ in G such that their weights are the same), then $\lambda_{min}(G) \leq \lambda_{min}(H) \leq \lambda_{max}(H) \leq \lambda_{max}(G)$. \square*

PROOF Since a similar theorem holds for a symmetric matrix (adjacency matrix for undirected graphs), the idea of the proof is to convert the anti-symmetric matrix to a (somewhat) symmetric matrix and use the same proof idea for symmetric matrix in the anti-symmetric case.

The rationale of the conversion is based on the fact that the anti-symmetric matrix has some degree of “symmetry” in that $M[i, j]$ and $M[j, i]$ only differ by a negation. In fact, if the imaginary unit $i = \sqrt{-1}$ is multiplied with the matrix, the result is a *Hermitian* matrix $i\mathbf{M}$, which is a symmetric matrix equivalent in the complex domain $\mathbb{C}^{n \times n}$. A

Hermitian matrix \mathbf{H} is a matrix in $\mathbb{C}^{n \times n}$ such that \mathbf{H} is equal to its conjugate transpose, i.e., $\mathbf{H}[i, j] = \overline{\mathbf{H}[j, i]}$, where $\overline{a + bi} = a - bi$ is the complex conjugate for real numbers a and b .

It is well known that the eigenvalues of a Hermitian matrix are all real numbers and all eigenvalues of an anti-symmetric matrix are all pure imaginary numbers [51]. It follows that in order to compare the magnitude of the eigenvalues, it is sufficient to compare the imaginary part (real numbers) only. Furthermore, one needs to prove that the set of eigenvalues of the anti-symmetric matrix is the same set of eigenvalues of the transformed Hermitian matrix by multiplying a negative imaginary unit, i.e., $\lambda(\mathbf{M}) = -i \lambda(i\mathbf{M})$. Therefore, the magnitudes of λ_{min} and λ_{max} of \mathbf{M} is the same as those of $\lambda_{min}(i\mathbf{M})$ and $\lambda_{max}(i\mathbf{M})$. Thus what remains is to prove that the theorem holds on a Hermitian matrix.

(1) $\lambda(\mathbf{M}) = -i\lambda(i\mathbf{M})$: for any eigenvector \mathbf{x}_i of $i\mathbf{M}$, the corresponding eigenvalue λ_i satisfies: $i\mathbf{M} \cdot \mathbf{x}_i = \lambda_i \mathbf{x}_i$. It follows that $\mathbf{M} \cdot \mathbf{x}_i = (-i\lambda_i)\mathbf{x}_i$. Based on the definition of eigenvalue, $-i\lambda_i$ is an eigenvalue of \mathbf{M} .

(2) Eigenvalue containment property holds for the Hermitian matrix: This proof is very similar to the proof for symmetric matrix. Since H is an induced subgraph of G , it is sufficient to prove that the property holds for the largest induced subgraph (denoted as H') of G , i.e., H' can be obtained from G by removing an arbitrary vertex and its incident edges. It can be proven that the same property holds for smaller induced subgraphs by induction on the number of vertices.

Given a graph G with n vertices and its largest induced subgraph H' with $(n - 1)$ vertices, one can always permute $\mathbf{M}_{H'}$ such that $\mathbf{M}_{H'}$ is the sub-matrix of \mathbf{M}_G in dimensions 1 to $(n - 1)$ without changing its eigenvalues. Suppose \mathbf{x} is the eigenvector of $\mathbf{M}_{H'}$ corresponding to $\lambda_{max}(\mathbf{M}_{H'})$, i.e., $\mathbf{x} = [x_1, x_2, \dots, x_{n-1}]$ and $\langle \mathbf{x}, \mathbf{x} \rangle = \sum_{i=1}^{n-1} \overline{x_i} * x_i = 1$, it is proven that $\mathbf{y} = [x_1, x_2, \dots, x_{n-1}, 0]$ is a normalized eigenvector of \mathbf{M}_G and its corresponding eigenvalue is $\lambda_{max}(\mathbf{M}_{H'})$. It is clear that $\langle \mathbf{y}, \mathbf{y} \rangle = 1$ and $\langle \mathbf{M}_G \cdot \mathbf{y}, \mathbf{y} \rangle = \langle \mathbf{M}_{H'} \cdot \mathbf{x}, \mathbf{x} \rangle = \lambda_{max}(\mathbf{M}_{H'})$.

Therefore, the following hold:

$$\begin{aligned}
 (\mathbf{M}_G \cdot \mathbf{y})^T \mathbf{y} &= \lambda_{max}(\mathbf{M}_{H'}) \\
 (\mathbf{M}_G \cdot \mathbf{y})^T \mathbf{y} &= \lambda_{max}(\mathbf{M}_{H'}) \mathbf{y}^T \mathbf{y} \\
 (\mathbf{M}_G \cdot \mathbf{y})^T &= \lambda_{max}(\mathbf{M}_{H'}) \mathbf{y}^T \\
 \mathbf{M}_G \cdot \mathbf{y} &= \lambda_{max}(\mathbf{M}_{H'}) \mathbf{y}
 \end{aligned}$$

From the last equation, $\lambda_{max}(\mathbf{M}_{H'})$ is an eigenvalue of \mathbf{M}_G by definition. Therefore, it follows that $\lambda_{max}(\mathbf{M}_{H'}) \in [\lambda_{min}(\mathbf{M}_G), \lambda_{max}(\mathbf{M}_G)]$. Similarly, $\lambda_{min}(\mathbf{M}_{H'})$ can also be proven to fall in the range $[\lambda_{min}(\mathbf{M}_G), \lambda_{max}(\mathbf{M}_G)]$. ■

This eigenvalue containment property allows us to choose λ_{min} and λ_{max} as two features to index. Testing for possible matching amounts to checking eigenvalue containment.

Computational Cost: Eigenvalue computation for a Hermitian matrix is $O(n^3)$, where n is the number of vertices in the bisimulation graph [114]. Since the twig patterns are usually very small and the large bisimulation graph for XML tree is broken into small ones in the index construction step, the real-world computation cost is very efficient—sub-millisecond for a dense 10×10 matrix and sub-second for a dense 300×300 matrix on a Pentium 4 3GHz PC. Eigenvalue calculation for sparse matrices (which are generated by most bisimulation graphs) should be even more efficient.

4.3.4 Other Features

In addition to eigenvalues of patterns, there are other possible features that can further increase the pruning power, such as the root label of the twig pattern or bisimulation graph. These can easily be included in the key to be indexed in the B^+ tree. Any bisimulation graph in the index that satisfies the eigenvalue range containment requirement but whose labels do not match with the twig pattern will also be pruned.

Other features may qualify as well, but FIX currently uses the set of $\{\lambda_{min}, \lambda_{max}, rl\}$ as features, where rl is the root label. These features are the keys of the B^+ tree index

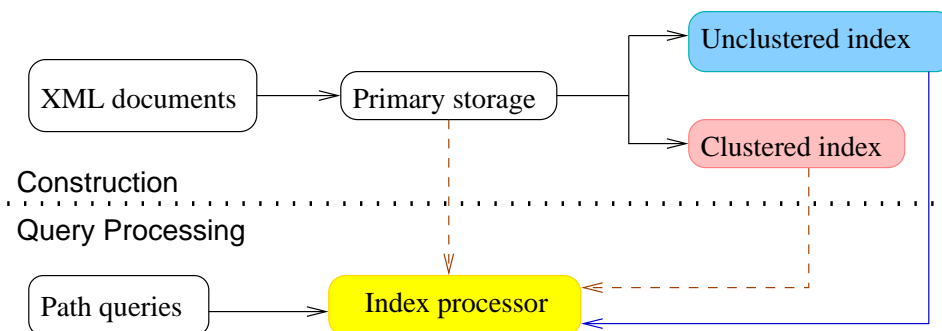


Figure 4.4: Building and querying indexes

described in the next section. The pruning criteria is that the indexed eigenvalue range does not contain the query eigenvalue range, or the root labels do not match. This is formally described in the following theorem.

Theorem 4.4 (Necessary conditions) . *Given a query Q and an XML tree D , if Q matches D , then $\lambda_{\min}(D) \leq \lambda_{\min}(Q) \leq \lambda_{\max}(Q) \leq \lambda_{\max}(D) \wedge lr(D) = lr(Q)$, where $\lambda_{\min}, \lambda_{\max}, rl$ represent the minimum, maximum eigenvalues and root label.*

PROOF This theorem is straightforward since the first conjunct holds due to Theorem 4.3 and the second conjunct holds by definition of existential match (Definition 4.2). ■

This theorem lays the foundation for the FIX pruning index: all subtrees that does not satisfy the necessary conditions will be pruned out, and the remaining will be the candidates and subject to the refinement phase.

4.4 Index Construction

During the construction phase, all subtrees under a certain depth limit are first enumerated. A vector of features is calculated for each subtree. The feature vector and the subtree forms the key-value pair for inserting to the B^+ tree. The overall architecture of constructing and querying the index is depicted in Figure 4.4. This section concentrates on the construction of FIX; query processing discussion is left to the next section.

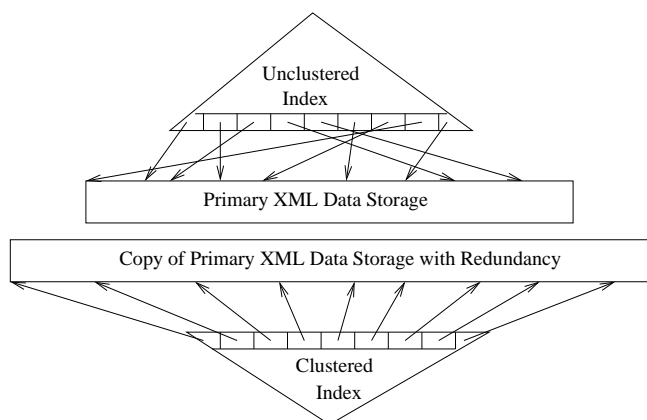


Figure 4.5: Clustered and Unclustered FIX Indexes

4.4.1 Types of Indexes

A clustered or unclustered index can be constructed similar to what is done in relational systems. Unlike relational databases, the clustered index for FIX incurs storage overhead due to the redundant storage of subelements as explained later. In both cases, the keys of the B^+ tree are the features but the “values” are different. In the unclustered index, the values are the references/pointers to the primary data storage (see Figure 4.5). The advantage of an unclustered index is that the primary storage does not need to be changed, and there is very small overhead for building the B^+ tree with pointers as the data entries. However, query processing may suffer from the fact that it needs to follow many pointers to perform the query refinement phase, which usually incurs random I/O.

On the other hand, a clustered index can be constructed by copying the contents of the primary storage pointed by the pointers and storing them sequentially according to their feature keys (see Figure 4.5). This is different from the relational case since reordering data units in place is impossible. The reason is that the data units in the XML case are subtrees and one may contain another as a descendant. Therefore, in order to make the value sorted in the same order as the keys, the clustered index has to copy each subtree to another storage, which may incur large space overhead. Thus, there is a tradeoff between the storage overhead and performance in the query refinement step.

Unclustered indexes are easier to build, and they are the only choice if data has to be

ordered on other criteria. They may be useful when the selectivity of the typical queries is high so that few pointers are produced as candidates. On the other hand, a clustered index could provide better performance because the I/O is essentially sequential. In the case where the database consists of a large collection of relatively small documents and each of them are inserted into the database as an entry, the clustered index may be the right choice because it is possible to reorder the documents so that their order coincides with the order of their feature keys. Furthermore, there is no redundancy in the storage since every document is treated as a unit. Therefore, the clustered index does not need to keep a copy of the primary storage and incurs no space overhead.

4.4.2 Index Construction for Collections of Documents

The index construction algorithm takes a collection of XML documents as input, and constructs a B⁺ tree index for them. The algorithm works in two phases: in the first phase, it generates *indexable units* that are small enough to efficiently extract features from. An indexable unit could be a small document in the collection, or a substructure of a large document. In the second phase, the features of the indexable units are computed and inserted into the B⁺ tree.

The index construction procedure is codified as the method CONSTRUCT-INDEX in Algorithm 4, where input C is a collection of XML documents (possibly singleton), L is the depth limit, and I is a B⁺ tree that holds the index entries. The depth limit is a parameter for a document being qualified as an indexable unit. The following subsection first introduces how to index an indexable unit, and the subsequent subsection introduces how to handle large documents.

4.4.3 Construction of an Index Entry for a Small Document

Each small document whose depth is no larger than the depth limit (an application-dependent threshold) is treated as a unit and converted into a bisimulation graph, which, in turn, is translated into an anti-symmetric matrix. Eigenvalues for each of these matrices are calculated and the λ_{max} and λ_{min} together with the root label of the document are used as the key to be inserted into the B⁺ tree. The value of the entry inserted into the B⁺

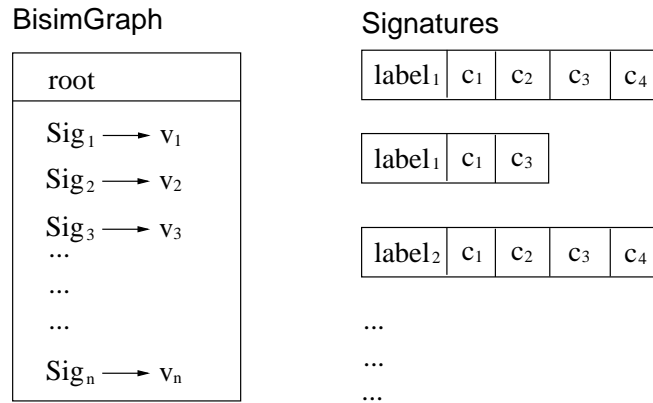


Figure 4.6: Data Structures: BiSimGraph and Signatures. v_i and c_i are references to vertices in the BiSimGraph.

tree is the document itself when building a clustered index, or the pointer to the primary storage for an unclustered index. This process is codified in the `CONSTRUCT-ENTRIES` method in Algorithm 4. The third parameter of the methods has to be set to 0 indicating that the document does not need to be partitioned.

In the input to the `CONSTRUCT-ENTRIES` method, X is the input event stream, and H is a B^+ tree index. Parameter L is the pattern depth limit and is set to 0 in this case. The variable G (line 1) is of type `BiSimGraph`, which is a data structure that contains two substructures: the root of the bisimulation graph and a mapping from a *signature* to a vertex in the bisimulation graph (see Figure 4.6 for an example). It also maintains the maximum depth of the bisimulation graph. The signature is a data structure that uniquely identifies a vertex. It consists of the vertex label and a set of child vertices. Two XML nodes are in the same equivalence class (bisimulation vertex) if and only if their signatures, namely, labels and children are the same by the definition of bisimilarity.

`CONSTRUCT-ENTRIES` works as follows: Whenever an open event (corresponding to encountering an open tag when parsing the XML document) is received, a new signature is created and initialized with its label and an empty set of child vertices (line 5). The pair of signature and pointer to the primary storage corresponding to the event is pushed onto a stack *PathStack* (line 6). This pair is popped whenever the corresponding closing event (corresponding to a closing tag) is received (line 8). Since at this time, all children

Algorithm 4 Constructing FIX for a Collection of Documents

CONSTRUCT-INDEX($C : \text{Collection}, L : \text{int}, I : \text{BTree}$)

```

1  for each XML document  $d \in C$ 
2  doif the depth of  $d \leq L$  then
3      CONSTRUCT-ENTRIES( $I, d, 0$ );
4      else CONSTRUCT-ENTRIES( $I, d, L$ );
5  end
6  end

```

CONSTRUCT-ENTRIES($H : \text{BTree}, X : \text{EventStream}, L : \text{int}$)

```

1   $G \leftarrow$  empty graph;  $\triangleright G$  is of type BisimGraph
2   $PathStack \leftarrow$  empty stack;
3  while  $X$  generates more event  $x$ 
4  doif  $x$  is an open event then
5       $sig \leftarrow \langle x.label, \emptyset \rangle$ ;  $\triangleright c\_set$  initialized to  $\emptyset$ 
6       $PathStack.push(\langle sig, x.start\_ptr \rangle)$ ;
7      elseif  $x$  is a closing event then
8           $\langle sig, start\_ptr \rangle \leftarrow PathStack.pop()$ ;
9           $u \leftarrow$  lookup  $sig$  in  $G$ ;
10         if  $sig$  is not in  $G$  then
11             create vertex  $u$  with label  $x.label$ ;
12             create edge  $(u, v_i)$  for each  $v_i \in sig.c\_set$ ;
13             create mapping  $sig \Rightarrow u$  in  $G$ ;
14         else release  $sig$ ;
15         end
16         if  $PathStack$  is not empty then
17              $p\_sig \leftarrow PathStack.top().first$ ;
18              $p\_sig.c\_set \leftarrow p\_sig.c\_set \cup \{u\}$ 
19         else  $G.root \leftarrow u$ ;
20         if  $L = 0$  then
21             BTREE-INSERT( $H, u, G.dep, start\_ptr$ );
22         end
23         end
24         if  $L > 0$  then
25             GEN-SUBPATTERN( $H, v, L, start\_ptr$ );
26         end
27     end
28 end

```

BTREE-INSERT($H : \text{BTree}, u : \text{BisimVertex}, L : \text{int}, ptr : \text{StoragePointer}$)

```

1  if  $u.eigs$  is not set then
2      convert  $u$  into matrix  $\mathbf{M}$  up to depth  $L$ ;
3       $\langle \lambda_{max}, \lambda_{min} \rangle \leftarrow \text{EIG-PAIR}(\mathbf{M})$ ;
4       $u.eigs \leftarrow \langle \lambda_{max}, \lambda_{min} \rangle$ ;
5  end
6   $k \leftarrow \langle u.eigs, u.label \rangle$ ;
7  if  $H$  is a clustered index then
8       $v \leftarrow$  pattern instance from the primary storage following  $ptr$ ;
9      insert  $v$  in  $H$  with key  $k$ ;
10 else insert  $ptr$  in  $H$  with key  $k$ ;
11 end

```

GEN-SUBPATTERN($H : \text{BTree}, v : \text{BisimVertex}, L : \text{int}, ptr : \text{StoragePointer}$)

```

1  if  $v.eigs$  is set then
2      BTREE-INSERT( $H, v, 0, ptr$ );
3  else  $Tr \leftarrow \text{BISIM-TRAVELER}(v, L, ptr)$ ;
4      CONSTRUCT-ENTRIES( $H, Tr, 0$ );
5  end

```

(and their descendants) corresponding to the current event have been visited and their corresponding bisimulation vertices are recorded in the signature that is popped from the stack (line 18), the algorithm needs to look up the mapping maintained in G to see if the signature already exists (line 9). If it is not in the mapping, then the algorithm needs to create a new bisimulation vertex u and insert all bisimulation vertices maintained in the signature into u 's children list, and then record the new mapping from the signature to u in G (lines 11–13). If the signature is already in the bisimulation graph, the algorithm only needs to release the memory acquired for the signature. If the *PathStack* is not yet empty (which means the whole tree has not yet been traversed), the algorithm needs to update the children list of u 's parent in the *PathStack* (lines 17–18); otherwise, the algorithm sets u as the root of graph G and calls BTREE-INSERT to update the database. $G.dep$ is the maximum depth of the bisimulation graph, which indicates that the whole graph should be indexed.

The BTREE-INSERT method is fairly straightforward: it first checks whether the bisimulation vertex has an $\langle \lambda_{max}, \lambda_{min} \rangle$ pair associated with it. If not, it converts the graph into an anti-symmetric matrix under the depth limitation, calculates the eigenvalue thereof, and associates the $\langle \lambda_{max}, \lambda_{min} \rangle$ pair with u (lines 2–4). Then it uses the pair and the root label as a key and inserts the pointer in the B^+ tree for the unclustered index. If the index is a clustered index, the algorithm needs to retrieve the XML documents from the primary storage and store them as values of the B^+ tree.

Complexity: CONSTRUCT-ENTRIES is a single-pass algorithm that reads each incoming event once. For each closing event, the algorithm searches the bisimulation graph for a signature, which could be $O(1)$ using an efficient hashing method. Therefore, the CPU cost of the construction algorithm is $O(n + m)$, where n is the number of events generated from the input event stream (in case of XML SAX-event stream, it is the number of XML elements in the whole collection), and m is the number of vertices in the bisimulation graph.

The major cost of Algorithm 4 is the I/O cost, which depends on the number of B^+ tree insertions and number of reads from the primary storage. In the unclustered case, the number of B^+ tree insertions is the same as the number of documents in the collection since only one bisimulation graph is generated for each document. In the clustered case,

the B^+ tree I/O is the same as the unclustered case but there is additional I/O cost for retrieving documents from the primary storage, which is proportional to the number of documents in the collection as well. In summary, the I/O cost is $O(N)$ where N is the number of documents in the collection.

4.4.4 Constructing Entries for a Large Document

The bisimulation graph of a large document could be very large. Furthermore, no substructures in the large document can be pruned if it is indexed as one entry. Therefore, it is necessary to enumerate subpatterns inside the document tree and populate the instances into the B^+ tree. If the database consists of multiple large documents, it is necessary to enumerate subpatterns for each of them.

First, it is necessary to restrict the subpattern size before enumerating its instances in the XML tree. Based on the same idea of local similarity in prior works [88, 33], the depth of subpatterns is limited to a small number k (k -patterns). With this construct, however, the index loses some expressive power: it can only answer a twig pattern up to depth k . The tradeoff between expressive power and efficiency is common [88] and does not invalidate the benefit of building the index. It is easy for the query optimizer to test whether a twig query is covered by an index.

The method for index construction with limited pattern depth is the `CONSTRUCT-ENTRIES` method in Algorithm 4, with a positive argument L as the depth limit. The `CONSTRUCT-ENTRIES` needs to call `GEN-SUBPATTERN` to enumerate subpatterns given the root of the subpattern and depth L . The `GEN-SUBPATTERN` method is based on the idea that a bisimulation graph “traveler” (`BISIM-TRAVELLER`) can be created to traverse the bisimulation graph in depth-first order within the depth limit L . During the traversal, it generates an open event when traversing to another vertex, or a closing event when it finishes traversing the subtree of the node or when it traverses to a depth of L . This stream of events can, in turn, be fed to the `CONSTRUCT-ENTRIES` method. The depth limit in the call to `CONSTRUCT-ENTRIES` is set to 0 whenever the whole subpattern is indexed. The method will generate a new bisimulation graph that is a subgraph of the original one, and store it into the B^+ tree as described in Section 4.4.3. To guarantee that the subpattern enumeration process is performed only once for each bisimulation vertex, the algorithm also

associates the bisimulation vertex with the $\langle \lambda_{max}, \lambda_{min} \rangle$ pair of the subpattern, indicating that this vertex has already been enumerated and the eigenvalues are calculated (line 1).

The reason that it is necessary to go all the way to define a traveler and call CONSTRUCT-ENTRIES again instead of using the subgraph beginning at the current vertex v is that the subgraph itself usually is *not* a bisimulation graph. The limit on the depth causes the subgraph to contain some repetitions such that the subgraph is no longer a bisimulation graph. For example, in Figure 4.2, the subgraph of depth 2 rooted at `bib` is not a bisimulation graph since `article` is repeated twice.

The following theorem derives the cost of the enumeration algorithm and is used to prove the completeness of the index.

Theorem 4.5 *For an index with positive depth limit, the number of subpattern instances that are enumerated by Algorithm 4 is exactly the same as the number of elements in the document.* □

PROOF With the depth limit $L > 0$ in Algorithm 4, the function GEN-SUBPATTERN is called once for each closing event. For each invocation of GEN-SUBPATTERN a new entry corresponding to a subpattern instance is inserted into the B^+ tree. Since the number of closing events equals the number of elements in an XML document, the inserted subpattern instances equals the number of elements. ■

Complexity: The CPU cost of building the index with positive depth limit is the same as the cost for building the index on the collection of small documents, except that there is the additional cost for enumerating subpatterns. For each vertex in the bisimulation graph, the subpattern rooted at this vertex is enumerated once, therefore the additional CPU cost is the same as the number of vertices in the bisimulation graph. Therefore, the CPU cost is $O(n + m)$ where n is the number of XML elements and m is the number of vertices in the bisimulation graph.

The I/O cost is dependent on the number of pattern instances generated, i.e., number of elements in the XML document. For each pattern instance, there is a B^+ tree insertion operation, and for clustered index, there is an additional read operation in primary storage. Since the B-tree insertion complexity is $\log_d(n)$ [44], where d is the fanout and n is the number of elements, the I/O cost is $O(n \log_d(n))$.

4.4.5 Completeness of Index Construction

The index constructed in the previous subsections is *complete* for any k -pattern query, if the depth limit of the index is at least k .

Theorem 4.6 *If the index is built with depth limit at least k (in the case where depth limit is 0 for collection of small documents, k is the maximum depth of the all documents in the collection), a k -pattern is not contained in the XML document, if it is not contained in the index.* □

PROOF It is straightforward to show that completeness holds for the collection of small documents case. If the twig pattern is of depth k , and it is contained in any of the documents, the document will be matched in accordance to Theorem 4.3.

In the large document case, since a subpattern for each XML element is generated (Theorem 4.5), the indexed pattern instances cover all subtrees of depth k . Therefore, if there is any XML node in the result of the k -pattern, the pattern instance of this node is already indexed and will be returned as a candidate result. ■

4.4.6 Supporting Value Equality Predicates

FIX supports value-based equality predicates such as in the query `//article[author = "John Smith"]/title`. Note that the PCDATA in the XML documents, as well as the atomic value “John Smith” in the query, can be thought of as “labels” of the text nodes, which are children of element nodes. However, the values cannot be directly used in the same way as the element node labels are in indexing and querying. The reason is that the bisimulation graph is converted to a matrix by mapping an edge (identified by the labels of the two incident vertices) to an integer. If the domain of one of the vertex labels is infinite, the edge will be mapped to an infinite domain as well, making the matrix computation impractical.

The solution to this problem is to map or hash the PCDATA or atomic value to an integer in a small range $(\alpha, \alpha + \beta]$, where α is the maximum of the element labels, and β is a small integer parameter. After the mapping, the hashed integer can be treated as the *label* of a value node, then the FIX index can be constructed based on the new document

tree with value nodes. It is straightforward to see that after the value-to-label mapping, all the properties (including the completeness) still holds for the index with value nodes. Therefore, FIX index uniformly supports structure and value matching.

One thing to note here is that it may be necessary to carefully choose the β value to trade off between query processing time and size of the index. With a large β , the values can be mapped to a large domain, and the bisimulation graph is large. Since the substructures are enumerated for each vertex in the bisimulation graph, there will be many substructures enumerated and inserted into the B^+ tree. This will result in a much larger B^+ tree compared to the B^+ tree containing only structures. On the other hand, with a small β , the B^+ tree will be small, but many different values will be hashed into the same label. This will introduce more false-positives because of the collisions in hashing. How to choose a proper β for a given data set is an interesting problem left for the future work.

4.5 Query Processing and Optimization using FIX

Using FIX for query processing has two steps: the *pruning* phase prunes the input and produces candidate results, and the *refinement* phase takes the candidate results and validates them using a query processor.

Given a twig query of depth k , it is relatively straightforward to perform query processing using FIX (Algorithm 5): one needs to first check whether the index covers the twig query by comparing the depth limit of the index and the depth of the twig query. If it does, the query tree is converted into a bisimulation graph (twig pattern), then the pattern is converted into an anti-symmetric matrix, and the λ_{max} and λ_{min} are computed. This pair of eigenvalues and the root label of the twig pattern are used as a key to perform range query in the index. For each candidate returned by the range query, the path query processor is invoked to refine the candidate and get the final results. Before the query processor takes over, the algorithm needs to replace the leading $//$ -axis with $/$ -axis. This is because any descendants of the root of an indexed pattern instance are also indexed. They will be visited eventually if they are returned by the index as candidates. For value predicates, it is straightforward to see that they can be answered without false-negatives.

The query tree corresponding to a general path expression that contains $//$ -axes in the

Algorithm 5 Index Query Processing

INDEX-PROCESSOR($Q : \text{TwigQuery}, Idx : \text{FIX}$)

```

1  check the  $Idx$  depth limit is no shorter than  $Q$ 's depth;
2   $Q' \leftarrow \text{CONVERT-TO-BISIM-GRAPH}(Q)$ ;
3   $M \leftarrow \text{CONVERT-TO-MATRIX}(Q')$ ;
4   $\langle \lambda_{max}, \lambda_{min} \rangle \leftarrow \text{EIG-PAIR}(M)$ ;
5   $k \leftarrow \langle \lambda_{max}, \lambda_{min}, \text{root label of } Q' \rangle$ ;
6   $C \leftarrow Idx.\text{search}(k)$ ;
7  if  $Idx$  has non-zero depth limit then
8      replace the leading  $//$ -axis with  $/$ -axis from  $Q$ ;
9  end
10 for each  $c \in C$ 
11 doif  $Idx$  is clustered then
12     run refinement query processor with  $Q$  on  $c$ ;
13 else run refinement query processor with  $Q$  following the pointer  $c$ ;
14 end

```

middle can be decomposed into multiple twig queries that are connected by $//$ -edges. For example, the query `//open_auction[.//bidder[name][email]]/price` can be decomposed into two sub-queries: `//open_auction/price` and `//bidder[name][email]`. If the database consists of small documents and the depth limit is set to unlimited, the document whose $[\lambda_{min}, \lambda_{max}]$ range contains the $[\lambda_{min}, \lambda_{max}]$ ranges of both twig queries should be returned as candidates. If the index is built with a non-zero depth limit on large documents, only pattern instances that contain the top sub-twig query (`//open_auction/price` in the above example) are returned as candidates, otherwise even if the candidate may match the descendant sub-twig query (`//bidder[name][email]` in the above example), the top sub-twig query will not be matched thus the whole query is not matched. In this case, the descendant sub-twig query does not provide any pruning power.

After rewriting a path expression into a pattern tree, as discussed in Section 3.2, the resulting pattern tree may also contain edges labeled with **following-sibling** axes. Since the **following-sibling** axes specify ordering constraints which are not supported in the index, the FIX index cannot handle such constraints. Therefore, the evaluation of a pattern tree Q that contains **following-sibling** edges consists of three steps: (1) remove the **following-sibling**

edges from Q to Q' ; (2) evaluate Q' using the FIX index operator to produce a set of candidates; and (3) evaluate Q using the NoK pattern matching operator, which is able to process a pattern tree with **following-sibling** edges, on the candidates to obtain the final results. The only difference between the above evaluation procedure and the index operator in Algorithm 5 is that the first step is added before the index operator is invoked. This will not change the completeness property as proved in Theorem 4.6, since after removing the **following-sibling** edges, the result of Q' will be a superset of the original query Q . Therefore, there are no false-negatives.

The cost of FIX index processing consists of three parts: CPU cost of converting a twig query into its bisimulation graph, converting the graph into a matrix, and computing the eigenvalues of the matrix. The cost of the first two components is $O(m)$ each, where m is the size of the query, and the eigenvalue computation is $O(m'^3)$, where m' is the size of the bisimulation graph and $m' \leq m$. For a reasonable sized query, these costs are negligible. The I/O cost includes searching the B^+ tree and retrieving the document from B^+ tree (for the clustered index) or from the primary storage (for the unclustered index). The cost of searching the B^+ tree is well studied and the missing part in cost estimation is the number of candidate results. This can be estimated if further knowledge (e.g., histograms on λ_{max} , λ_{min} , and root labels of pattern instances) is available. A good practice is to build a histogram on the primary sorting key (e.g., λ_{max}) in the B^+ tree. The rest of the cost is that of refinement of the candidate results. Although the number of candidates may be the same, clustered and unclustered index may have much different cost due to different degree of randomness in I/O.

4.6 Experimental Evaluation

This section first reports the performance of structural FIX indexing with respect to three implementation-independent metrics, as well as its actual run-time speedup against the state-of-the-art indexing techniques. Then the integrated value and structural index is evaluated. While the wall clock time speedup is the “net effect” of the benefit of using FIX index over a specific algorithm implementation, implementation independent metrics reveal more insights into the design decisions of the FIX index and provide a general

data sets	size	# elements	ICT	UIdx	CIIdx
XBench	27.9 MB	115306	17.8 sec	0.2 MB	6.1 MB
DBLP	169 MB	4022548	32.5 sec.	2 MB	77.9 MB
XMark	116 MB	1666315	86 sec.	5.6 MB	143.3 MB
Treebank	86 MB	2437666	375 sec.	37.3 MB	310.6 MB

Table 4.1: Characteristics of experimental data sets, the index construction times (ICT), and the sizes of the unclustered index (UIdx) and clustered index (CIIdx)

guideline of how much improvement the FIX index can achieve for any implementation.

The FIX index is implemented in C++ and uses Berkeley DB for the B⁺ tree implementation. The NoK processor [143] is used to perform the refinement step. To compare with the unclustered FIX index, the implementation of NoK operator is extended with the support for // -axes. To compare with the clustered FIX index, the disk-based F&B index [131] is chosen, whose implementation is obtained from the authors. The disk-based F&B index has been reported to have superior performance over several other indexes, so it is chosen as a representative state-of-the-art clustering index. All the tests are conducted on a Pentium 4 PC with 3GHz CPU and 1GB memory running Windows XP.

4.6.1 Test Data Sets and Index Construction

Both synthetic and real data sets are tested. In the category of large collection of small documents, XBench [136] TCMD (text-centric multi-document) data set is used. The XBench TCMD data set models the real world text-centric XML data sets such as the Reuters news corpus and the Springer digital library. This data set contains 2,607 documents with various sizes from 1KB to 130KB. The document structures have a small degree of variations, e.g., an `article` element may or may not have a `keywords` subelement. Since all documents in the collection are small, their substructures are not enumerated in each document when the index is constructed, i.e., the depth limit parameter in Algorithm 4 is set to zero.

Tests are also conducted with non-zero depth limit on large XML documents: DBLP [94], XMark [119] with scale factor 1, and Treebank [7], where DBLP and Treebank are the same

data sets that are used for experimental evaluation in Chapter 3. They are chosen because of their different structural characteristics. The structure in DBLP is very regular and the tree is shallow, so the same structure is repeated many times, making each structural pattern less selective. The XMark data set is structure-rich, fairly deep and very flat (fan-out of the bisimulation graph is large), therefore, the structures are less repetitive. The Treebank data set represents highly recursive documents. It is very deep but not as flat as XMark, and the structures are also very selective.

The basic statistics, index construction time, and index sizes for these data sets are listed in Table 4.1. The constructed index for XBench TCMD data has no depth limit, and the indexes for the other data sets are constructed by enumerating subpatterns of depth limit 6. The construction times for indexes with smaller depth limits are slightly faster. This depth limit is chosen so that the index can cover fairly complex twig queries. Depending on the complexity of the bisimulation graph of the document and the depth limit, the enumerated subpattern could be too large for calculating eigenvalues (e.g., number of edges is larger than 3000). In this case, the eigenvalues are not calculated, but an artificial $[\lambda_{min}, \lambda_{max}]$ range $[0, \infty]$ is used to guarantee that the instances of this subpattern will always be returned as a candidate result. This may lose pruning power, but fortunately, there are very few such cases in all the test data sets for reasonable depth limit of 6 (1 for DBLP, 11 for XMark, and 1 for Treebank).

4.6.2 Implementation-independent Metrics

Three metrics are defined to evaluate the effectiveness of FIX: pruning power (pp), selectivity (sel), and false-positive ratio (fpr) as follows:

$$\begin{aligned} sel &= 1 - rst / ent \\ pp &= 1 - cdt / ent \\ fpr &= 1 - rst / cdt \end{aligned}$$

where cdt is the number of entries returned by the index as candidate results, ent is number of all entries in the index, and rst is the number of entries that actually produce at least one final result. Note that selectivity is defined differently in some literature as rst / ent . There are two other metrics that are widely used in the information

retrieval literature: precision and recall. Precision is defined as $\{\text{relevant documents}\} \cap \{\text{retrieved documents}\} / \{\text{retrieved documents}\}$. Recall is defined as $\{\text{relevant documents}\} \cap \{\text{retrieved documents}\} / \{\text{relevant documents}\}$. In terms of FIX index, the retrieved documents and relevant documents correspond to *cdt* and *rst* respectively. Therefore, precision for FIX index is $1 - fpr$ and recall is 100%. For the index with depth limit 0 on a large collection of small documents, *pp* is the ratio of number of documents pruned by the index over the total number of documents in the collection. For the index with non-zero depth limit *k*, since each element corresponds to an entry in the index (the subtree of depth *k* starting from that element), *pp* is the ratio of elements pruned over the total number of elements as a starting pointer for further refinement.

In order to evaluate the real effectiveness of the index, the pruning power metric should be combined with the *selectivity* of the query. The low pruning power of a query does not mean that the index is ineffective if the selectivity is also low (i.e., the query is not selective). The only bad case is when the selectivity is high but the pruning power is low.

The metric *fpr* is another indicator of the effectiveness of the pruning of the FIX index against the “perfect” index, which produces no false-positives.

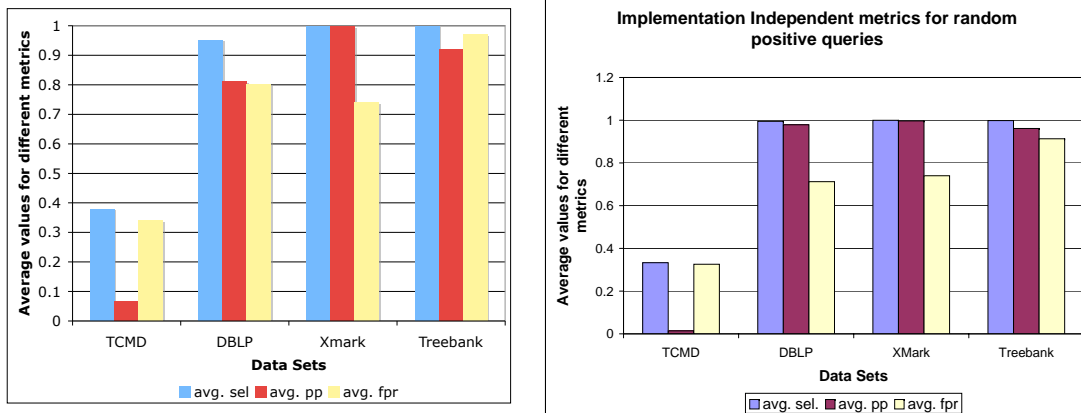
For each data set, 1000 test queries are randomly generated. Representative queries are also selected based on their selectivities: low, medium, and high. However, depending on the characteristics of the data sets (i.e., the distribution of the substructures), these queries may not cover all 3 selectivity criteria. For example, since each document in the Xbench TCMD have very similar structure, the queries are more likely to fall into the category of low selectivity. On the other hand, XMark and Treemark data sets are structure-rich, thus almost all queries fall into the high selectivity category. For these cases, the representative queries are selected with *relatively* high or low selectivity².

```
TCMD_hi : /article/epilog[acknowledgements]/references/a_id
TCMD_md : /article/prolog[keywords]/authors/author/contact[phone]
TCMD_lo : /article[epilog]/prolog/authors/author
DBLP_hi : //proceedings[booktitle]/title[sup][i]
DBLP_md : //article[number]/author
```

²Queries that have selectivity 0 and 1 are eliminated since they do not reveal much information about the index.

query	<i>sel</i>	<i>pp</i>	<i>fpr</i>
TCMD_hi	79.31%	26.12%	71.99%
TCMD_md	49.23%	5.62%	46.21%
TCMD_lo	16.85%	0.35%	16.29%
DBLP_hi	99.97%	99.79%	84.91%
DBLP_md	72.59%	70.85%	5.91%
DBLP_lo	47.36%	47.35%	0.002%
XMark_hi	99.96%	99.87%	75.13%
XMark_md	99.10%	98.71%	30.14%
XMark_lo	98.89%	98.43%	30.01%
TrBnk_hi	99.97%	95.37%	99.45%
TrBnk_md	99.81%	85.97%	98.67%
TrBnk_lo	97.48%	95.36%	45.79%

Table 4.2: Implementation-independent metrics for representative queries for each data sets in each category



(a) Average implementation independent metrics (b) Average implementation independent metrics for positive queries

Figure 4.7: Average selectivity, pruning power, and false-positive ratio of 1000 random queries on different data sets

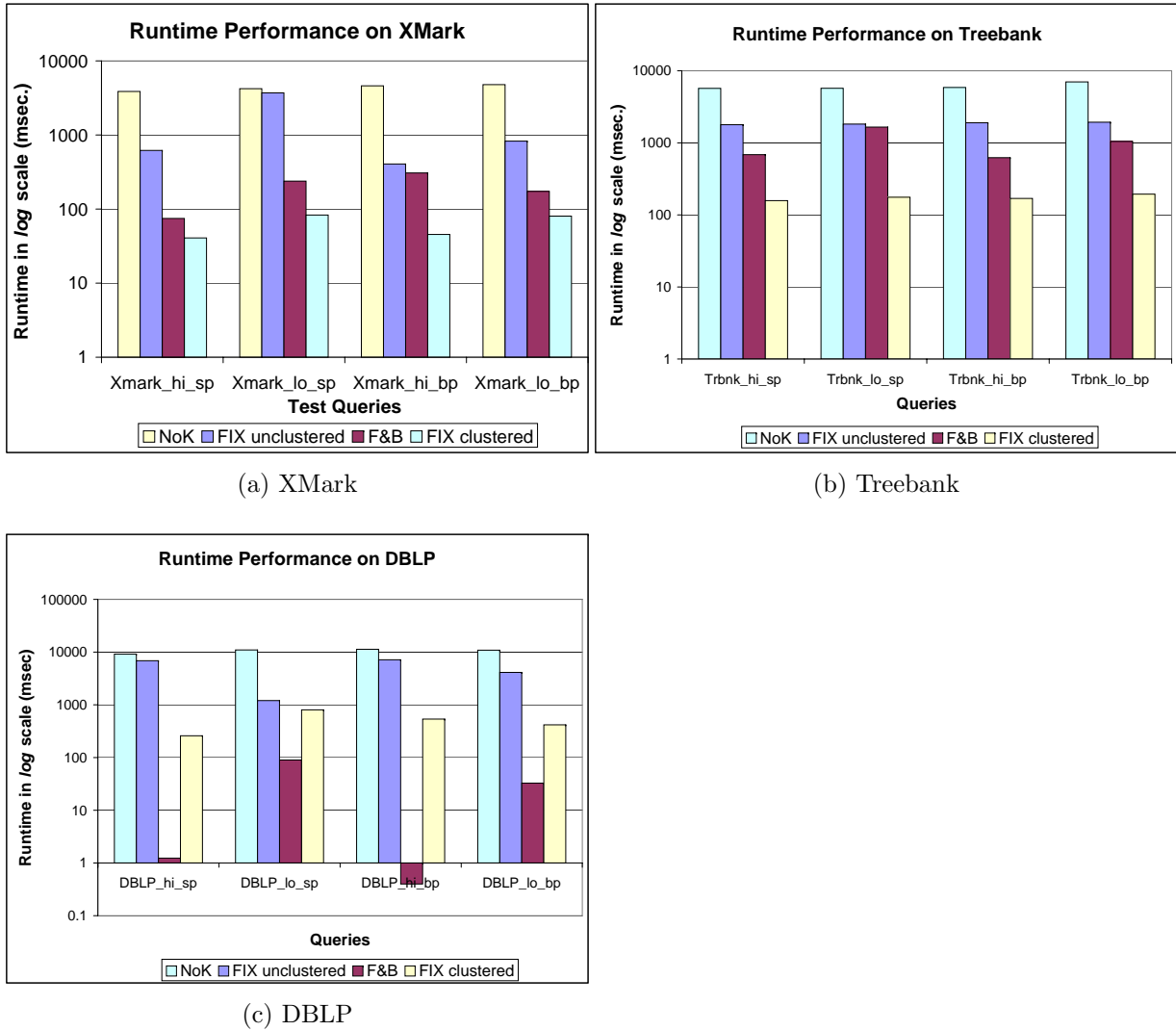


Figure 4.8: Runtime comparisons on XMark, Treebank, and DBLP

```

DBLP_lo : //inproceedings[url]/title
XMark_hi: //category/description[parlist]/parlist/listitem/text
XMark_md: //closed_auction/annotation/description/text
XMark_lo: //open_auction[seller]/annotation/description/text
TrBnk_hi: //EMPTY/S/NP[PP]/NP
    
```

TrBnk_md: //S[VP]/NP/NP/PP/NP

TrBnk_lo: //EMPTY/S[VP]/NP

The selectivity, pruning power, and false-positive ratios for these queries are listed in Table 4.2. For low selectivity queries (e.g., TCMD_lo), FIX does not show strong pruning power. However, since only about 16% of the returned candidates are false positives, the index still performs well in that most of the remaining candidates produce final results. On the other hand, for highly selective queries, such as (almost) all XMark and Treebank queries, FIX prunes very well, very close to the selectivity. This means that the features used in FIX reflect the intrinsic structural characteristics of the patterns. However, the false-positive ratios for queries in this category could also be high (e.g., TrBnk_hi and TrBnk_md). This suggests that there may be other features that are unique in this data set that are missed in the FIX index, which will be considered in the future work. For the queries in the medium category, the effectiveness of FIX varies. The pruning powers of FIX on some queries (e.g., DBLP_md and XMark_md) are very close to their selectivities, and the false-positive ratios are reasonable. On the other hand, some queries have poor pruning power (e.g., TCMD_md) or the false-positive ratio is high (e.g., TrBnk_lo). This case represents the grey area that is hard to estimate the cost.

The average of the three metrics over the random 1000 queries for each data set is shown in Figure 4.7, where Figure 4.7(a) shows the average metrics for 1000 random queries, and Figure 4.7(b) shows the average metrics for the subset contains positive queries only (queries that return non-zero results). These two figures show very similar properties, so only Figure 4.7(a) is explained here. As seen from the figure, the average pruning power is very close to the selectivity for XMark and Treebank, but there are about 32% and 14% differences for TCMD and DBLP, respectively. One of the reasons for this is that, as indicated earlier, unlike XMark and Treebank, XBench TCMD and DBLP are not structure-rich. *Structural* indexes that cluster based on structures are not likely to be effective anyway. The following subsection shows that the integrated structural and value index can improve the pruning power as well as the query processing time.

4.6.3 Run-time Performance Evaluation

Although clustered indexes (such as F&B index and clustered FIX index) are more efficient in query processing, they are less efficient in result subtree construction (due to the loss of document order). Furthermore, clustering criteria may conflict with other sorting criteria, making the unclustered FIX index or the original storage preserving document order (such as the one in [143]) preferable. To conduct fair comparison, the performance are compared in two scenarios: (1) unclustered FIX index vs. the NoK navigational operator without index support, and (2) the clustered FIX index vs. clustered F&B index.

To be able to benchmark different types of queries, both simple path (sp) and branching path (bp) queries are investigated. Together with the selectivity dimension, low (lo) and high (hi) selectivity, there are four test queries for each data sets: $\{\text{hi, lo}\} \times \{\text{sp, bp}\}$. The test queries are listed as follows:

```
XMark_hi_sp: //item/mailbox/mail/text/emph/keyword
XMark_lo_sp: //description/parlist/listitem
XMark_hi_bp: //item[name]/mailbox/mail[to]/text[bold]/emph/bold
XMark_lo_bp: //item[payment][quantity][shipping][mailbox/mail/text]
              /description/parlist
Trbnk_hi_sp: //EMPTY/S/NP/NP/PP
Trbnk_lo_sp: //EMPTY/S/VP
Trbnk_hi_bp: //EMPTY/S/NP[PP]/NP
Trbnk_lo_bp: //EMPTY/S[VP]/NP
DBLP_hi_sp : //inproceedings/title/i
DBLP_lo_sp : //dblp/inproceedings/author
DBLP_hi_bp : //inproceedings[url]/title[sub][i]
DBLP_lo_bp : //article[number]/author
```

Figure 4.8 depicts the speedup of the FIX indexes to the existing techniques in logarithmic scale. As shown in Figures 4.8a and 4.8b, FIX unclustered and clustered indexes performs considerably better than the NoK or F&B indexes, respectively. However, on the more regular and simple data set DBLP (Figure 4.8c), although the FIX unclustered

index still outperforms NoK, the F&B index outperforms the FIX clustered index, particularly in the cases of queries with high selectivity. The reason is that the structure of DBLP data set is very regular and shallow. The whole F&B index for DBLP is only 180 KB, and could easily fit into main memory due to the caching mechanism of F&B index implementation. However, it is more likely that queries on simple data sets usually involve value constraints. For such a general path expression, the majority of processing time is spent on the value-predicate evaluation.

4.6.4 Performance of Value Indexes

To balance the query performance and the space overhead, β is set to 3 when building the value index. Since DBLP is the only real data set (the PCDATA in other data sets are all randomly generated), and since queries with value-predicates are all branching paths, only branching paths are tested with high selectivity and low selectivity on the DBLP data set. The test queries are listed as follows:

```
DBLP_v1_hi: //proceedings[publisher="Springer"][title]
DBLP_v1_hi: //inproceedings[year="1998"][title]/author
```

Figure 4.9a shows the implementation-independent metrics. For low selective queries, the FIX index with values performs comparably to the FIX index with no values as far as the implementation-independent metrics are concerned. For high selective queries, however, FIX index with values demonstrates a significant improvement over the pure structural index, with the selectivity and pruning power almost identical, and false-positive ratio (*fpr*) around 1.7%. Figure 4.9b shows the runtime speedups compared to F&B index. The FIX index with values outperforms the F&B index on both queries by more than a factor of 2. However, the FIX index with values does not come for free, the construction time and memory requirement are much higher than the pure structural index (around a factor of 30 and 10 with $\beta = 10$, respectively). With careful tuning of the β value, one can achieve the balance between the cost associated with the index construction and the savings for the query processing.

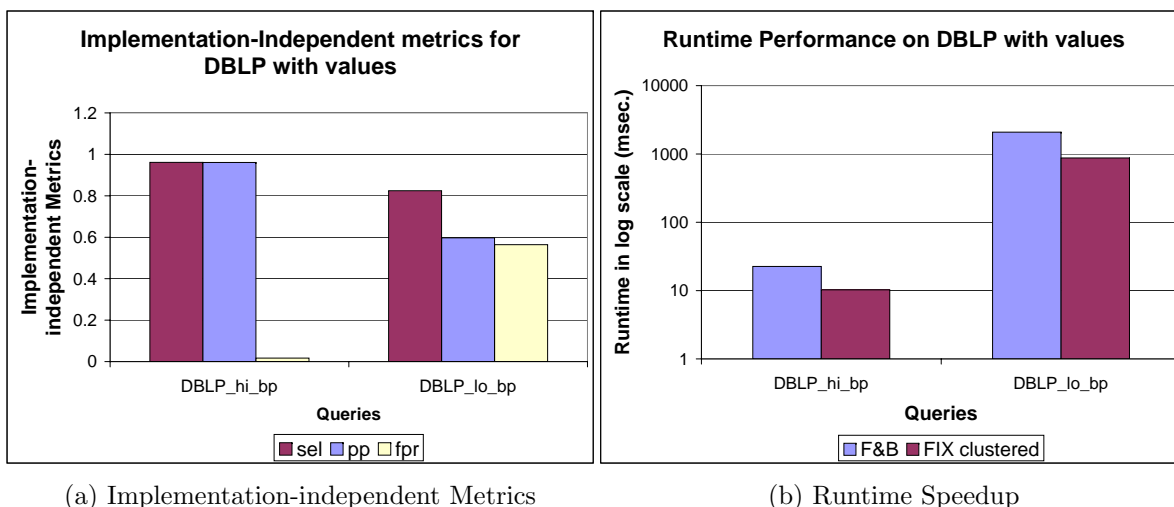


Figure 4.9: DBLP with values

4.7 Comparisons to Related Work

As discussed in Chapter 2, a wide variety of join-based, navigational, and hybrid XPath processing techniques are now available [139, 15, 28, 74, 143]. Much research has focused on indexing techniques to improve these existing query processors. For example, XB-tree [28], XR-tree [82], iTwigStack [34], and ToXin [20] are proposed to prune the input lists for the holistic twig join operator. FIX is also a pruning index, but it is not designed to work for a particular operator, but as a generic index that can be coupled with any path processing operator that can perform query refinement.

Chapter 2 also introduces recent research on clustering indexes [99, 88, 87, 131]. The common theme of these clustering techniques is that they are all based on some variant of *simulation/bisimulation* graph of the XML data tree. FIX does not use the bisimulation graph itself as an index, but uses the structural information extracted from the bisimulation graph. By separating a large bisimulation graph into smaller ones, one can quickly find a substructure as the candidate of a pattern without traversing the whole graph.

Eigenvalues and spectral graph theory have many applications in other areas of computer science. The initial idea of FIX was inspired by the work in computer vision, where

spectra of shock graphs are used to index visual objects [124]. The shock graph is an unlabeled directed graph to represent the vision of an object. They use the full set of eigenvalues as features to approximate query processing, but did not make use or prove the $[\lambda_{min}, \lambda_{max}]$ property for substructure queries. There is also related work in the area of data mining, in which a large collection of graphs are indexed by identifying “features” — frequent substructures [134, 135]. Their features are combinatorial in that features are compared by subgraph isomorphism.

4.8 Conclusion

As more and more documents are stored in XML databases, an index is needed to quickly retrieve a subset of candidates to do further refinement. Depending on the characteristics of the data sets, a value-based index or a structural index or both are appropriate for certain queries. This chapter proposes the feature-based index FIX for indexing substructures as well as values in a document or collection of documents. The FIX approach is the first XML indexing technique to take the substructures and values as a whole object and compute its distinctive features. Unlike many other indexing techniques, FIX can be combined with an XPath query processor with little or no change in its implementation. FIX has been successfully applied as a pruning index for an existing highly optimized navigational operator, resulting in orders of magnitude speedup in running time.

Chapter 5

Cardinality Estimation of Path Expressions

5.1 Introduction

In Chapters 3 and 4, two physical operators were presented to evaluate a path expression. Depending on the query selectivity, one operator may be more efficient than the other. For example, if the selectivity of a query is very low, which means that almost all elements will be returned as results, the NoK pattern matching operator may be more efficient than an unclustered (or even clustered) FIX index since the former entails sequential I/O and the latter random I/O. On the other hand, if the selectivity is high, a FIX index operator is likely to be more efficient. The query optimizer needs to calculate the cost of each query operator and select the optimal one. Usually the cost of an operator for a given path query is heavily dependent on the number of final results returned by the query in question, and the number of temporary results that are buffered for its sub-queries (see e.g., [142]). Therefore, accurate cardinality estimation is crucial for a cost-based optimizer.

The problem of cardinality estimation for a path query in XML distinguishes itself from the problem of cardinality estimation in relational database systems. One of the major differences is that a path query specifies *structural constraints* (a.k.a. tree patterns) in addition to value-based constraints. These structural constraints suggest a combined combinatorial and statistical solution. That is, one needs to consider not only the statistical

distribution of the values associated with each element, but also the structural relationships between different elements. Estimating cardinalities of queries involving value-based constraints has been extensively studied within the context of relational database systems, where histograms are used to compactly represent the distribution of values. Similar approaches have also been proposed for XML queries [111]. This chapter focuses on the structural part of this problem and proposes a novel synopsis structure, called XSEED¹ [146], to estimate the cardinality for path queries that only contain structural constraints. Although XSEED can be incorporated with the techniques developed for value-based constraints, the general problem is left for future work.

The XSEED synopsis is inspired by the previous work for estimating cardinalities of structural constraints [64, 37, 112]. These approaches, usually, first summarize an XML document into a compact graph structure called a *synopsis*. Vertices in the synopsis correspond to a set of nodes in the XML tree, and edges correspond to parent-child relationships. Together with statistical annotations on the vertices and/or edges, the synopsis is used as a guide to estimate the cardinality using a graph-based estimation algorithm. XSEED follows this general idea but develops a solution that meets multiple criteria: the accuracy of the estimations, the types of queries and data sets that this synopsis can cover, the adaptivity of the synopsis to different memory budgets, the cost of the synopsis to be created and updated, and the estimation time comparing to the actual querying time. These are all important factors for a synopsis to be useful in practice.

None of the existing approaches considers all of these criteria. For example, TreeSketch [112] focuses on the accuracy of the cardinality estimation. It starts off by building a count-stable graph² to capture the complete structural information in the tree (i.e., cardinality estimation can be 100% accurate for all types of queries). The count-stable graph could be very large (e.g., the count-stable graph for the 100MB XMark [119] data set has 59,015 vertices and 315,011 edges). Then it relies on an optimization algorithm to reduce the count-stable graph to fit into the memory budget and still retain as much information as possible. Since the optimization problem is NP-hard, the solutions are usually sub-optimal, and the construction time could be prohibitive for large and complex data sets (e.g., it takes

¹XSEED stands for XML Synopsis based on Edge Encoded Digraph.

²See Section 2.4.2 for the detailed explanation of count-stable graphs.

more than 13 hours to construct the synopsis for the 100MB XMark [119] data set on a dedicated machine). Therefore, this synopsis is hardly affordable for a complex data set.

In contrast, XSEED takes the opposite approach: an XSEED structure is constructed by first building a very small *kernel* (usually a couple of kilobytes for most tested data sets), and then incrementally adds/deletes information to/from the synopsis. The kernel captures the coarse structural information in the data, and can be constructed easily. The purpose of the small kernel is not to make it optimal in terms of accuracy; it has to work for all types of queries and data sets, while, at the same time, having several desirable features such as the ease of construction and update, a small footprint, and the efficiency of the estimation algorithm. A unique feature of the XSEED kernel is that it recognizes and captures recursions in the XML documents. Recursive documents usually represent the most difficult cases for path query processing and cardinality estimation. None of the existing approaches address recursive documents and the effects of recursion over the accuracy of cardinality estimation.

Even with the small kernel, XSEED provides good accuracy in many test cases (see Section 5.5 for details). In some cases, XSEED's accuracy is an order of magnitude better than other synopses (e.g., TreeSketch) that use a larger memory budget. The ability to capture recursion in the kernel is one major contribution to better accuracy. However, the high compression ratio of the kernel introduces information loss, resulting in greater estimation errors in some cases. To remedy the accuracy deficiency for these cases, another layer of information, called a *hyper-edge table (HET)*, is introduced on top of the kernel. The HET captures the special cases that deviate from the assumptions that the kernel relies on. The experiments show that even a small amount of this extra information can greatly improve the accuracy for many cases. The HET can be pre-computed in a similar or shorter time than other synopses, or it can be dynamically fed by a self-tuning optimizer using query feedback. This information can be easily maintained, i.e., it can be added to or deleted from the synopsis whenever the memory budget changes. When the underlying XML data change, the optimizer can choose to update the information eagerly or lazily. In this way, XSEED enjoys better accuracy as well as adaptivity.

Figure 5.1 depicts the process of constructing and maintaining the XSEED kernel and HET and utilizing them to predict the cardinality. In the construction phase, the XML doc-

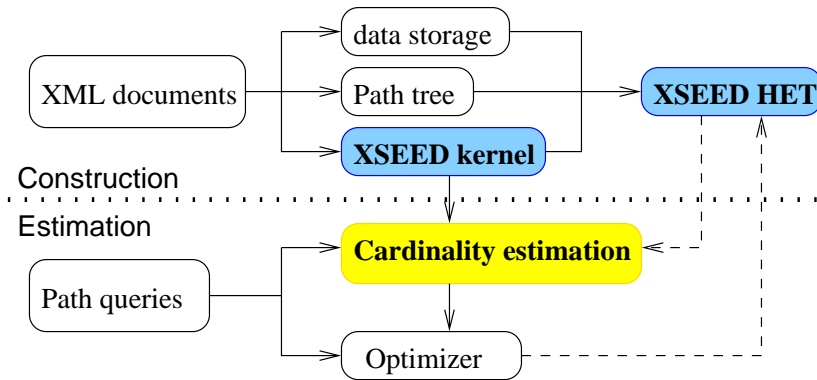


Figure 5.1: Cardinality estimation process using XSEED

ument is first parsed to generate the NoK XML storage structure [143], the path tree [13], and the XSEED kernel. The HET is constructed based on these three data structures if it is pre-computed. In the estimation phase, the optimizer calls the cardinality estimation module to predict the cardinality for an input query, with the knowledge acquired from the XSEED kernel and optionally from the HET. After the execution, the optimizer may feedback the actual cardinality or selectivity of the query to the HET, which might results in an update of the data structure.

The rest of the chapter is organized as follows: Section 5.2 introduces the XSEED kernel. Section 5.3 presents the cardinality estimation algorithm using the XSEED kernel. Section 5.4 introduces optimization techniques to improve XSEED accuracy. Section 5.5 reports the experimental evaluation. Section 5.6 compares XSEED with related work.

5.2 Basic Synopsis Structures—XSeed Kernel

Throughout the chapter, an n -tuple (u_1, u_2, \dots, u_n) is used to denote a path $u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_n$ in an XML tree or a synopsis structure, and $|p|$ is used to denote the cardinality of a path expression p (the number of XML elements returned by p).

Example 5.1 The following DTD describes the structure of an article document.

```
<!ELEMENT article (title, authors, chapter*)>
```

```
<!ELEMENT chapter (title, para*, sect*)>
<!ELEMENT sect    (title?, para*, sect*)>
```

As in the previous chapters, element names in the above DTD are mapped to the alphabet $\{a, t, u, c, p, s\}$:

```
article --> a    title --> t    authors --> u
chapter --> c    para  --> p      sect   --> s
```

An example XML tree instance conforming to this DTD and the above element name mapping is depicted in Figure 5.2a. To avoid possible confusion, a framed character, e.g., \boxed{a} , is used to represent the abbreviated XML tree node label whenever possible. \square

An interesting property of the XML document is that it could be *recursive*, i.e., an element could be directly or indirectly nested in an element with the same name. For example, a `sect` element could contain another `sect` subelement. In the XML tree, recursion represents itself as multiple occurrences of the same label in a rooted path.

Definition 5.1 (Recursion Levels) Given a rooted path in the XML tree, the maximum number of occurrences of any label minus 1 is the *path recursion level* (PRL). The recursion level of a node in the XML tree is defined to be the PRL of the path from root to this node. The *document recursion level* (DRL) is defined to be the maximum PRL over all rooted paths in the XML tree. \square

For example, the recursion level of the path (a, c, s, p) in Figure 5.2a is 0 since each label only occurs once in the path, and the recursion level of path (a, c, s, s, s, p) is 2 since there are three `s` nodes in the path.

Recursion could also exist in a path expression. Recall that a path expression consists of a list of location steps, each of which consists of an axis, a NodeTest, and zero or more predicates. Each predicate could be another path expression. When matching with the nodes in an XML tree, the NodeTests specify the tag name constraints, and the axes specify the structural constraints. We classify path queries into three classes: *simple path* expressions that are linear paths containing $/$ -axes only, *branching path* expressions that include branching predicates but also only have $/$ -axes, and *complex path* expressions that

contain branching predicates and/or $//$ -axes. Note that a general path expression after the rewriting introduced in Section 3.2 may also contain following-sibling axes. At this point, XSEED synopsis does not support this type of axis since it requires keeping the ordering information between siblings, which may not be efficiently compressible.

Definition 5.2 (Recursive Path Expression) A path expression is *recursive* with respect to an XML document if an element in the document could be matched to more than one NodeTest in the expression. \square

For example, a path expression $//s//s$ on the XML tree in Figure 5.2a is recursive since an \boxed{s} node at recursion level greater than zero could be matched to both NodeTests. It is straightforward to see that simple and branching path expressions consisting of only $/$ -axis cannot be recursive. Recursive path queries always contain $//$ -axes, and they usually present themselves on recursive documents. However, it is also possible to have recursive path queries on non-recursive documents, when the queries contain the sub-expression $//*//*$. Similarly, we define the *query recursion level* (QRL) of a path expression as the maximum number of occurrences of the same NodeTests with $//$ -axis along any rooted path in the query tree. In general, recursive documents are the hardest documents to summarize, and recursive queries are the hardest queries to evaluate and to estimate.

As discussed in Chapter 2, a structural summary is a graph that summarizes the nodes and edges in the XML tree. Preferably, the summary graph should preserve all the structural relations and capture the statistical properties in the XML tree. The following definition introduces one structural summary—the label-split graph [110], which is the basis of the XSEED kernel.

Definition 5.3 (Label-split Graph) Given an XML tree $T(V_t, E_t)$, a label-split graph $G(V_s, E_s)$ can be uniquely derived from a mapping $f : V_t \rightarrow V_s$ as follows:

- For every $u \in V_t$, there is a $f(u) \in V_s$.
- A node $u \in V_t$ is mapped to $f(u) \in V_s$ if and only if their labels are the same.
- For every pair of nodes $u, v \in V_t$, if $(u, v) \in E_t$, then there is a directed edge $(f(u), f(v)) \in E_s$.
- No other vertices and edges are present in $G(V_s, E_s)$. \square

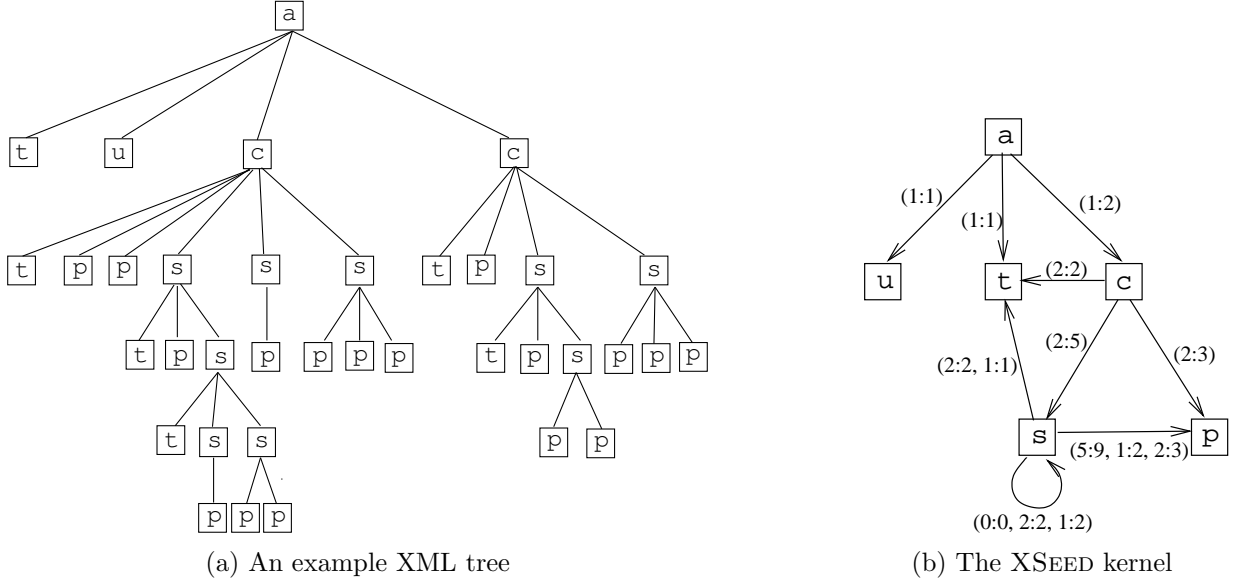


Figure 5.2: An example XML tree and its XSEED kernel

In comparison, bisimulation graph clusters tree nodes by their subtrees, and label split graph clusters tree nodes by node label only. Figure 5.2b, without the edge labels, depicts the label-split graph of the XML document shown in Figure 5.2a. The label-split graph preserves the node label and edge relation in the XML tree, but not the cardinality of the relations.

Definition 5.4 (XSeed Kernel) The XSEED kernel for an XML tree is an edge-labeled label-split graph. Each edge $e = (u, v)$ in the graph is labeled with a vector of integer pairs $(p_0:c_0, p_1:c_1, \dots, p_n:c_n)$. The i -th integer pair $(p_i:c_i)$, referred as $e[i]$, indicates that, at recursion level i , there are a total of p_i elements mapped to the synopsis vertex u and c_i elements mapped to the synopsis vertex v . The p_i and c_i are called *parent-count* (referred as $e[i][P_CNT]$) and *child-count* (referred as $e[i][C_CNT]$), respectively. \square

Example 5.2 The XSEED kernel shown in Figure 5.2b is constructed from the XML tree in Figure 5.2a. In the XML tree, there is one **a** node and it has two **c** children. Thus, the edge (a, c) of XSEED kernel is labeled with integer pair $(1:2)$. Out of these two **c** nodes in the XML tree, there are five **s** child nodes. Therefore, the edge (c, s) in the kernel is

labeled with (2:5). Out of the five $\boxed{\mathbf{s}}$ nodes, two of them have one $\boxed{\mathbf{s}}$ child each (for a totally two $\boxed{\mathbf{s}}$ nodes having two $\boxed{\mathbf{s}}$ children). Since the two $\boxed{\mathbf{s}}$ child nodes are at recursion level 1, the integer pair at position 1 of the label of (s, s) is 2:2. Since the recursion level could not be 0 for any path having an edge (\mathbf{s}, \mathbf{s}) , the integer pair at position 0 for this edge is 0:0. Furthermore, one of the two $\boxed{\mathbf{s}}$ nodes at recursion level 1 has two $\boxed{\mathbf{s}}$ children, which makes the integer pair at position 2 of the edge label (s, s) 1:2. \square

The following observations of XSEED kernel are important for the cardinality estimation algorithm given in Section 5.3.

Observation 1: For every path (u_1, u_2, \dots, u_n) in the XML tree, there is a corresponding path (v_i, v_2, \dots, v_n) in the kernel, where the label of v_i is the same as the label of u_i . Furthermore, for each edge (v_i, v_{i+1}) , the number of integer pairs in the label is greater than the recursion level of the path (u_1, \dots, u_{i+1}) . For example, the path $(\mathbf{a}, \mathbf{c}, \mathbf{s}, \mathbf{s}, \mathbf{s}, \mathbf{p})$ in Figure 5.2a has a corresponding path $(\mathbf{a}, \mathbf{c}, \mathbf{s}, \mathbf{s}, \mathbf{s}, \mathbf{p})$ in the XSEED kernel in Figure 5.2b. Moreover, the number of integer pairs in the label vector prevents a path with recursion level larger than 2, e.g., $(\mathbf{a}, \mathbf{c}, \mathbf{s}, \mathbf{s}, \mathbf{s}, \mathbf{s}, \mathbf{p})$, from being derived from the synopsis.

Observation 2: For every node u in the XML tree, if its children have m distinct labels (not necessarily different from u 's label), then the corresponding vertex v in the kernel has at least m out-edges, where the labels of the destination nodes match the labels of the children of u . This observation directly follows from the first observation. For example, the children of $\boxed{\mathbf{c}}$ nodes in the XML tree in Figure 5.2a have three different labels, thus the $\boxed{\mathbf{c}}$ vertex in the XSEED kernel in Figure 5.2b has three out-edges.

Observation 3: For any edge (u, v) in the kernel, the sum of the child-counts over all recursive levels i and greater is exactly the total number of elements that should be returned by the path expression $q//u//v$, whose recursion level is i and where q is a path expression that exists in the kernel. As an example, the number of results of expression $//\mathbf{s}//\mathbf{s}//\mathbf{p}$ on the XML tree in Figure 5.2a is 5, which is exactly the sum of the child-counts of the label associated with edge (\mathbf{s}, \mathbf{p}) at recursion level 1 and 2.

The first observation guarantees that the synopsis preserves the complete information of the simple paths in the XML tree. However, some simple rooted paths that can be

derived from the synopsis may not exist in the XML tree. That is, the kernel may contain false positives for a simple path query. For example, if a new node $\boxed{\mathbf{s}}$ is added as the fifth child of $\boxed{\mathbf{a}}$ in Figure 5.2a, then there will be a new edge (\mathbf{a}, \mathbf{s}) with label $(1:1)$ in the synopsis in Figure 5.2b. A simple path $(\mathbf{a}, \mathbf{s}, \mathbf{t})$ can be derived from the synopsis but it does not exist in the XML tree.

The second observation guarantees that, for any branching path query, if it has a match in the XML tree, it also has a match in the synopsis. Again, false positives for branching path queries are also possible. This is straightforward to see: after insertion of $\boxed{\mathbf{a}}$ node as described above, the branching path query $/\mathbf{a}/\mathbf{s}[\mathbf{t}][\mathbf{p}]$ has a match in the synopsis, but not in the XML tree.

The third observation connects the recursion levels in the data and in the query. This is useful in answering complex queries containing $//$ -axes.

Kernel Construction The XSEED kernel can be generated while parsing the XML document. The pseudo-code in Algorithm 6 can be implemented using a SAX event-driven XML parser.

The *path_stk* in line 1 is a stack of vertices (and other information) representing the path while traversing the kernel. Each stack entry $(\langle u, out_edges \rangle$ in line 9) is a binary tuple, in which the first item indicates which vertex in the kernel corresponds to the current XML element, and the second item keeps a set of (e, l) pairs, in which e is an outedge of u , and l is the recursion level of the rooted path ended with the edge e . These pairs are used to increment the parent-count in the case of a close tag event (line 20).

The *rl_cnt* in line 2 is a data structure, called “counter stacks”, which efficiently calculates the recursion level of a path in expected $O(1)$. When traversing the XML tree, the vertices in the XSEED kernel are pushed onto and popped from *rl_cnt* as in a stack (line 7, 11, and 22). The key idea of the data structure to guarantee the efficiency is to partition the items into different stacks based on their number of occurrences. A hash table is kept to give the number of occurrences for any item pushed onto the counter stacks. Whenever an item is pushed onto *rl_cnt*, the hash table is checked, the counter is incremented, and the item is pushed onto the corresponding stack maintained in the data structure. When an item is popped from *rl_cnt*, its occurrence is looked up in the hash table, popped from

Algorithm 6 Constructing the XSEED Kernel

```

CONSTRUCT-KERNEL( $S$  : Synopsis,  $X$  : XMLDoc)
1   $path\_stk \leftarrow$  empty stack;
2   $rl\_cnt \leftarrow$  empty counter stacks;
3  while the parser generates more event  $x$  from  $X$  do
4      if  $x$  is an opening tag event then
5           $v \leftarrow$  GET-VERTEX( $S, x$ );
6          if  $path\_stk$  is empty then
7               $rl\_cnt.push(v)$ ;
8               $path\_stk.push(\langle v, \emptyset \rangle)$ ;
9          else  $\langle u, out\_edges \rangle \leftarrow path\_stk.pop()$ ;
10              $e \leftarrow$  GET-EDGE( $S, u, v$ );
11              $l \leftarrow rl\_cnt.push(v)$ ;
12              $e[l][C\_CNT] \leftarrow e[l][C\_CNT] + 1$ ;
13              $out\_edges \leftarrow out\_edges \cup (e, l)$ ;
14              $path\_stk.push(\langle u, out\_edges \rangle)$ ;
15              $path\_stk.push(\langle v, \emptyset \rangle)$ ;
16         end
17     elseif  $x$  is a closing tag event then
18          $\langle v, out\_edges \rangle \leftarrow path\_stk.pop()$ ;
19         for each pair  $(e, l) \in out\_edges$  do
20              $e[l][P\_CNT] \leftarrow e[l][P\_CNT] + 1$ ;
21         end
22          $rl\_cnt.pop(v)$ ;
23     end
24 end

```

the corresponding stack indicated by the occurrence, and the occurrence counter in the hash table is decremented. The recursion level of the whole path is indicated by the number of non-empty stacks minus 1. As an example, after pushing the sequence of (a, b, b, c, c, b) the data structure is shown in Figure 5.3. a and b are pushed onto counter stack 1 since their occurrences are 0 before inserting. When the second b is pushed, the counter of b is already 1, thus the new b is pushed to stack 2. Similarly, the following c, c,

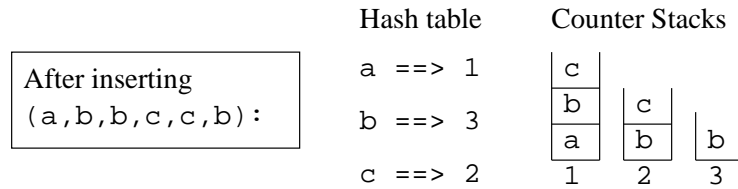


Figure 5.3: Counter stacks for efficient recursion level calculation

and **b** are pushed onto stacks 1, 2 and 3, respectively. This data structure guarantees efficient calculation of recursion levels and is also used in the cardinality estimation algorithm introduced in Section 5.3.

The functions `GET-VERTEX` and `GET-EDGE` (lines 5 and 10) search the kernel and return the vertex or edge indicated by the parameters. If the vertex or edge is not in the graph then it is created.

Synopsis Update When the underlying XML document is updated, i.e., some elements are added or deleted, the kernel can be incrementally updated. The basic idea is to compute, for each subtree that is added or deleted, the kernel structure for the subtree. Then it can be added or subtracted from the original kernel using an efficient graph merging or subtracting algorithm [46].

When deleting a subtree, the algorithm constructs a new kernel for the subtree. The next step is to determine which vertex in the original kernel corresponds to the parent of the root of the new kernel. Suppose that the new kernel and the original kernel are k' and k , respectively, the root of k' is r' and its parent in k is p , then subtraction of k' from k takes two steps:

1. Get the label of edge (p, r') , subtract 1 from the child-count of the integer pair at the recursion level of r' . If the child-count is 0, then set the parent-count to 0 as well, and adjust the size of the vector if necessary.
2. For each edge e' in k' , locate the same edge e in k , subtract the parent-count and child-count in e' from e at each recursion level. The vector size should also be adjusted accordingly, and if the size of a vector is 0, the edge should be deleted. When adding a subtree to the XML tree, the way to incrementally update the kernel is similar.

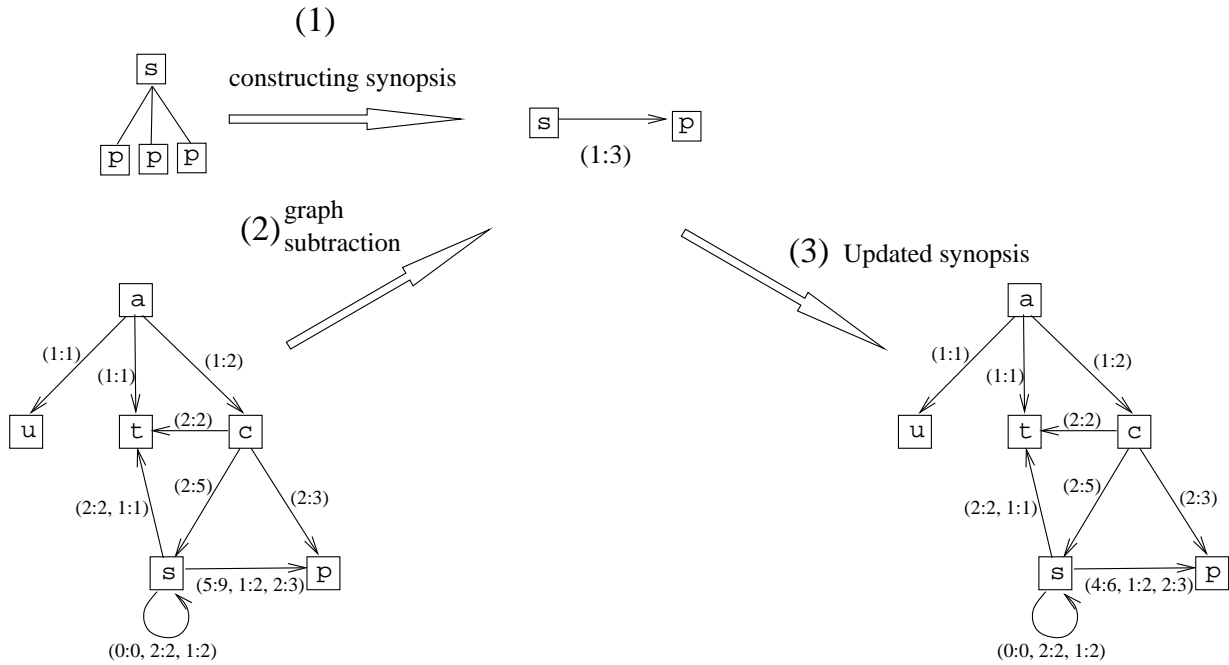


Figure 5.4: Update synopsis after deleting a subtree $sp)p)p))$ from the XML tree in Figure 5.2a

The only difference is to change the minus operation to plus, and to add edges if necessary.

Figure 5.4 shows an example of deleting the whole subtree rooted at s , whose string representation is $sp)p)p))$, as the rightmost child of the second c element. In this example, only the label of the edge (s, p) need to be updated.

The hyper-edge table can also be incrementally updated when a subtree is added (deleted) to (from) the XML tree. One only needs to (re-)compute the errors related to the paths that are updated by the new kernel. The old entries in the table are deleted and the new entries with the new errors are added.

5.3 Cardinality Estimation

The following notions are crucial to understand how cardinalities are estimated using XSEED.

Definition 5.5 (Forward and Backward Selectivity) For any rooted path $p_{n+1} = (v_1, v_2, \dots, v_n, v_{n+1})$ in the XSEED kernel $G(V_s, E_s)$, denote $e_{(i,i+1)}$ as the edge (v_i, v_{i+1}) , p_i as the sub-path (v_1, v_2, \dots, v_i) , and r_i as the recursion level of p_i , then the *forward selectivity* ($fsel$) and *backward selectivity* ($bsel$) of path p_{n+1} are defined as:

$$\begin{aligned} fsel(p_{n+1}) &= \frac{|/v_1/v_2/\dots/v_n/v_{n+1}|}{S_{n+1}}, \\ bsel(p_{n+1}) &= \frac{|/v_1/v_2/\dots/v_n[v_{n+1}]|}{|/v_1/v_2/\dots/v_n|}, \end{aligned}$$

where S_{n+1} is the sum of child-counts at recursion level r_{n+1} over all in-edges of vertex v_{n+1} , i.e.,

$$S_{n+1} = \sum e_{(i,n+1)}[r_{n+1}][\text{C_CNT}], \quad \forall e_{(i,n+1)} \in E_s. \quad \square$$

Intuitively, forward selectivity is the proportion of v_{n+1} nodes that are contributed by the path (v_1, v_2, \dots, v_n) , and backward selectivity captures the proportion of v_n nodes that: (1) are contributed by the path $(v_1, v_2, \dots, v_{n-1})$; and (2) have a child v_{n+1} .

In Definition 5.5, if the probability that v_n has a child v_{n+1} is independent of v_n 's ancestors, then $bsel$ can be approximated as:

$$bsel(p_{n+1}) \approx \frac{e_{(n,n+1)}[r_{n+1}][\text{P_CNT}]}{S_n},$$

where S_n is defined similar to S_{n+1} . This approximated $bsel$ is the proportion of v_n that are contributed by *any* path that have a child v_{n+1} . Combining the definition and approximation, the cardinality of the branching path $p_n[v_{n+1}]$ can be estimated using the cardinality of the simple path p_n as follows:

$$\begin{aligned} |p_n[v_{n+1}]| &= |p_n| \times bsel(p_{n+1}) \\ &\approx |p_n| \times \frac{e_{(n,n+1)}[r_{n+1}][\text{P_CNT}]}{S_n}. \end{aligned}$$

More generally, given a branching path expression $p = /v_1/v_2/\cdots/v_n[v_{n+1}]\cdots[v_{n+m}]$, letting $q = /v_1/v_2/\cdots/v_n$, and assuming that $bsel$ of q/v_{n+i} is independent of $bsel$ of q/v_{n+j} for any $i, j \in [1, m]$, then the cardinality of p is estimated as:

$$\begin{aligned} |q/[v_{n+1}]\cdots[v_{n+m}]| &\approx |q| \times bsel(q/v_{n+1}) \times \cdots \\ &\quad \times bsel(q/v_{n+m}) \\ &= |q| \times absel(p), \end{aligned}$$

where $absel(p)$ denotes the *aggregated bsels* (products) of the rooted paths that end with a predicate query tree node. Since the $bsel$ of any simple path can be approximated using the XSEED kernel, the problem is reduced to how to estimate the cardinality of a simple path query.

For the simple path query $/v_1/v_2/\cdots/v_n/v_{n+1}$ in Definition 5.5, if again the probability of v_i having a child v_{i+1} is independent of v_i 's ancestors, the cardinality of $/v_1/v_2/\cdots/v_n/v_{n+1}$ can be approximated as:

$$|/v_1/v_2/\cdots/v_n/v_{n+1}| \approx e_{(n,n+1)}[r_{n+1}][\mathbf{C_CNT}] \times fsel(p_n).$$

Intuitively, the estimated cardinality of $/v_1/v_2/\cdots/v_n/v_{n+1}$ is the number of v_{n+1} that are contributed by v_n times the proportion of v_n that are contributed by the path $/v_1/v_2/\cdots/v_{n-1}$. Based on this, $fsel$ can be estimated as:

$$fsel(p_{n+1}) \approx \frac{e_{(n,n+1)}[r_{n+1}][\mathbf{C_CNT}] \times fsel(p_n)}{S_{n+1}}.$$

Since $fsel$ is defined recursively, the calculation of $fsel(p_{n+1})$ should be bottom-up, starting with $fsel(p_1)$, and then $fsel(p_2)$ and so on. At the same time, the estimated cardinalities of all sub-expressions are also calculated.

Example 5.3 Suppose the optimizer wants to estimate the cardinality of query $/a/c/s/s/t$ on the kernel shown in Figure 5.2b. The following table shows the vertices in a path while traversing the kernel, the estimated cardinality, forward selectivity, and backward selectivity.

vertex	cardinality	<i>fsel</i>	<i>bsel</i>
$\boxed{\mathbf{a}}$	1	1	1
$\boxed{\mathbf{c}}$	2	1	1
$\boxed{\mathbf{s}}$	5	1	1
$\boxed{\mathbf{s}}$	2	1	0.4
$\boxed{\mathbf{t}}$	1	1	0.5

The first row in the table refers to the path consisting of the single root node $\boxed{\mathbf{a}}$; the second row refers to the path of (a, c) in the kernel, and so on. In particular the cardinality of the last row indicates the estimated cardinality of the path expression $/\mathbf{a}/\mathbf{c}/\mathbf{s}/\mathbf{s}/\mathbf{t}$.

When traversing the first vertex $\boxed{\mathbf{a}}$, the cardinality, *fsel*, and *bsel* are all set at their initial values of 1. When traversing the second vertex $\boxed{\mathbf{c}}$, the cardinality is approximated as $|\mathbf{a}/\mathbf{c}| = e_{(a,c)}[0][\mathbf{C_CNT}] \times fsel(\mathbf{a}) = 2 \times 1 = 2$, since the recursion level of path (\mathbf{a}, \mathbf{c}) is 0. *fsel* (\mathbf{a}, \mathbf{c}) is estimated as $\frac{|\mathbf{a}/\mathbf{c}|}{S_{(a,c)}} = \frac{2}{2} = 1$, where $S_{a,c}$ is the sum of child-counts of all in-edges of \mathbf{c} at recursion level specified by path (\mathbf{a}, \mathbf{c}) . *bsel* (\mathbf{a}, \mathbf{c}) is estimated as $\frac{e_{(a,c)}[0][\mathbf{P_CNT}]}{S_{(a)}} = \frac{1}{1} = 1$. When traversing a new vertex, the same calculations will take the results associated with the old vertices and the edge labels in the XSEED kernel as input, and produce the cardinality, *fsel*, and *bsel* for the new vertex as output. \square

The cardinality of a simple path query can be estimated as above; if the optimizer wants to estimate the cardinality of a branching query or a complex path query consisting of $//$ -axes and wildcards (*), a matching algorithm needs to be developed to match the pattern tree specified by the expression to the kernel. In fact, the XSEED estimation algorithm defines a *traveler* (Algorithm 7) and a *matcher* (Algorithm 8). The matcher calls the traveler, through the function call NEXT-EVENT, to traverse the XSEED kernel in depth-first order. The rooted path is maintained while traveling. Whenever a vertex is visited, the traveler generates an *open event*, which includes the information about the label of the vertex, the DeweyID of this vertex, the estimated cardinality, the forward selectivity, and the backward selectivity of the current path. When finishing the visit of a vertex (due to some criterion introduced later), a *close event* is generated. In the end, an end-of-stream (EOS) event is generated when the whole graph is traversed. The matcher accepts this stream of events and maintains a set of internal states to match the tree pattern specified by the path expression.

Algorithm 7 Synopsis TravelerNEXT-EVENT()

```

1  if pathTrace is empty then
2    if no last event then           ▷ current vertex is the root
3      h ← hash value of curV;
4      fp ← ⟨curV, 1, 1.0, 1.0, 0, h⟩;
5      pathTrace.push(fp);
6      evt ← OPEN-EVENT(v, card, fsel, bsel);
7    else evt ← EOS-EVENT();
8    end
9  else evt ← VISIT-NEXT-CHILD();
10 end

```

VISIT-NEXT-CHILD()

```

1  ⟨u, card, fsel, bsel, chdcnt, hsh⟩ ← pathTrace.top();
2  kids ← children of curV;
3  while size of kids is greater than chdcnt do
4    v ← kids[chdcnt];
5    if ¬END-TRAVELING(v, chdcnt) then
6      curV ← v;
7      ⟨v, card, fsel, bsel, hsh⟩ ← pathTrace.top();
8      evt ← OPEN-EVENT(v, card, fsel, bsel);
9      return evt;
10   end
11   increment chdcnt in pathTrace.top() by 1;
12 end
13 evt ← CLOSE-EVENT(u);
14 return evt;

```

END-TRAVELING(*v* : SynopsisVertex, *chdCnt* : int)

```

1  old_rl ← the recursion level of current path without v;
2  rl ← the recursion level of current path and v;
3  ⟨stop, card, fsel, bsel, n_h⟩ ← EST(v, rl, old_rl);
4  if stop then
5    return true;
6  end
7  fp ← ⟨v, card, fsel, bsel, 0, n_h⟩;
8  pathTrace.push(fp);
9  return false;

```

EST(*v* : SynopsisVertex, *rl* : int, *old_rl* : int)

```

1  ⟨u, card, fsel, bsel, chdcnt, hsh⟩ ← pathTrace.top();
2  e ← GET-EDGE(u, v);
3  if rl < e.label.size() then
4    n_card ← e[rl][C_CNT] * fsel;
5    sum_cCount ← TOTAL-CHILDREN(u, old_rl);
6    n_bsel ← e[rl][P_CNT] / sum_cCount;
7  else n_card ← 0;
8  end
9  sum_cCount ← TOTAL-CHILDREN(v, rl);
10 n_fsel ← n_card / sum_cCount;
11 if n_card ≤ CARD.THRESHOLD then
12   stop ← true;
13 else stop ← false;
14 end
15 return ⟨stop, n_card, n_fsel, n_bsel, n_hsh⟩;

```

Algorithm 7 is a simplified pseudo-code for the traveler algorithm. When traversing the graph, the algorithm maintains a global variable *pathTrace*, which is a stack of *footprint* (line 4). A footprint is a tuple including the current vertex, the estimated cardinality of the current path, the forward selectivity of the path, the backward selectivity of the path, the index of the child to be visited next, and the hash value for the current path. If the next vertex to be visited is the root of the synopsis, an open event with initial values are generated, otherwise the NEXT-EVENT function calls the VISIT-NEXT-CHILD function to move to the next vertex in depth-first order. The latter function calls the END-TRAVELING function to check whether the traversal should terminate (this is necessary for a synopsis containing cycles). Whether to stop the traversal is dependent on the estimated cardinality calculated in the EST function. In the EST function, the cardinality, forward selectivity, and backward selectivity are calculated as described earlier. If the estimated cardinality is less than or equal to some threshold (*CARD_THRESHOLD*), the END-TRAVELING function returns **true**, otherwise it returns **false**. The OPEN-EVENT function accepts the vertex, the estimated cardinality, the forward selectivity, and the backward selectivity as input, and generates an event including the input parameters and the DeweyID as output. The DeweyID of the event is maintained by the OPEN-EVENT and CLOSE-EVENT functions and is not shown in Algorithm 7.

If the sequence of open and close events are treated as open and close tags of XML elements, with the cardinality and selectivities as attributes, the traveler generates the following XML document from the XSEED kernel in Figure 5.2b:

```
<a dID="1." card="1" fsel="1" bsel="1">
  <t dID="1.1." card="1" fsel="0.2" bsel="1"/>
  <u dID="1.2." card="1" fsel="1" bsel="1"/>
  <c dID="1.3." card="2" fsel="1" bsel="1">
    <t dID="1.3.1." card="2" fsel="0.4" bsel="1"/>
    <p dID="1.3.2." card="3" fsel="0.25" bsel="1"/>
    <s dID="1.3.3." card="5" fsel="1" bsel="1">
      <t dID="1.3.3.1." card="2" fsel="0.4" bsel="0.4"/>
      <p dID="1.3.3.2." card="9" fsel="0.75" bsel="1"/>
      <s dID="1.3.3.3." card="2" fsel="1" bsel="0.4">
```

```

<t dID="1.3.3.3.1." card="1" fsel="1" bsel="0.5"/>
  <p dID="1.3.3.3.2." card="2" fsel="1" bsel="0.5"/>
    <s dID="1.3.3.3.3." card="2" fsel="1" bsel="0.5">
      <p dID="1.3.3.3.3.1." card="3" fsel="1" bsel="1"/>
    </s> </s> </s> </c> </a>

```

The tree corresponding to this XML document is dynamically generated and does not need to be stored. Since it captures all the simple paths that can be generated from the kernel, it is called the *expanded path tree* (EPT). In a highly recursive document (e.g., Treebank), the EPT could be even larger than the original XML document. This is because a single path with a high recursion level will result in generating other non-existing paths during the traversal. In this case, a higher *CARD_THRESHOLD* is needed to limit the traversal. As demonstrated by the experiments, this heuristic greatly reduces the size of the EPT without causing much error.

Algorithm 8 shows the pseudo-code for matching a query tree rooted at *qroot* with the EPT generated from the kernel *K*. The algorithm maintains a stack of *frontier sets*, which is a set of query tree nodes (QTN) for the current path in the traversal. The QTNs in the frontier set are the candidates that can be matched with the incoming event. Initially the stack contains a frontier set consisting of the *qroot* itself. Whenever a QTN in the frontier set is matched with an open event, the children of the QTN are inserted into a new frontier set (line 11). Meanwhile, the matched event is buffered into the output queue of the QTN as a candidate match (line 12). In addition to the children of the QTN that match the event, the new frontier set should also include all QTNs whose axis is “//” (line 15). After that, the new frontier set is ready to be pushed onto the stack for matching with the incoming open events if any.

Whenever a close event is seen, the matcher first cleans up the unmatched events in the output queue associated with each QTN (line 20). The call *qroot.rmUnmatched()* checks the output queue of each QTN under *qroot*. If some buffered event does not have all its children QTN matched, these events are removed from the output queue. After the cleanup, if the top of the output queue of *qroot* indicates a total match, the estimated cardinality is calculated (line 22). Otherwise, if *qroot* is not a total match, the partial results should be removed from *qroot*. Finally, the stack for the frontier set is popped

Algorithm 8 Synopsis Matcher

CARD-EST(K : Kernel, $qroot$: QueryTreeNode)

```

1   $frtSet \leftarrow \{qroot\}$ ;
2   $frtStk.push(frtsSet)$ ;
3   $est \leftarrow 0$ ;
4   $evt \leftarrow \text{NEXT-EVENT}()$ ;
5  while  $evt$  is not an end-of-stream (EOS) event do
6    if  $evt$  is an open event then
7       $frtSet \leftarrow frtStk.top()$ ;
8       $new\_fset \leftarrow \emptyset$ ;
9      for each query tree node  $q \in frtSet$  do
10       if  $q.label = evt.label \vee q.label = "*" \text{ then}$ 
11         insert  $q$ 's children into  $new\_fset$ ;
12         insert  $evt$  into  $q$ 's output queue;
13       end
14       if  $q.axis = "/" \text{ then}$ 
15         insert  $q$  into  $new\_fset$ ;
16       end
17     end
18      $frtStk.push(new\_fset)$ ;
19     else if  $evt$  is a close event then
20        $qroot.rmUnmatched()$ ;
21       if  $qroot.isTotalMatch() \text{ then}$ 
22          $est \leftarrow est + \text{OUTPUT}(evt.dID, qroot)$ ;
23       else if  $evt$  is matched to  $qroot \text{ then}$ 
24          $qroot.rmDescOfSelf(evt.dID)$ ;
25       end
26     end
27      $frtStk.pop()$ ;
28   end
29 end
30    $evt \leftarrow \text{NEXT-EVENT}()$ ;
31 end
32 return  $est$ ;

```

OUTPUT(dID : DeweyID, $qroot$: QueryTreeNode)

```

1   $Q \leftarrow rstQTN.outQ$ ;
2   $est \leftarrow 0$ ;
3   $absel \leftarrow \text{AGGREGATED-BSEL}(qroot)$ ;
4  for each  $evt \in Q$  do
5      $est \leftarrow est + evt.card * absel$ ;
6  end
7   $Q.clear()$ ;
8   $rstQTN.rmDescOfSelfSubTree(dID)$ ;
9  return  $est$ ;

```

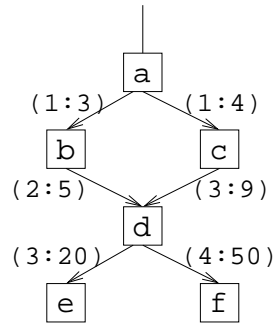


Figure 5.5: Example of ancestor independence assumption breaks

indicating that the current frontier set is finished matching.

In the `OUTPUT` function, the algorithm needs to sum the cardinalities of all the events cached in the resulting QTN. If there are predicates, the function `AGGREGATED-BSEL` calculates the product of backward selectivities of all events matched with predicate QTNs. After the summation, the output queue of the resulting QTN and all its descendant QTNs should be cleaned up.

5.4 Optimization for Accuracy—HET

The whole idea of the cardinality estimation using the `XSEED` kernel is to first *compress* the XML tree, in the construction phase, into a graph structure that contains a small amount of statistical annotations, and then, in the estimation phase, *decompress* the graph into a tree (EPT) based on the independence assumption (explained in detail later). The accuracy of cardinality estimation, therefore, depends upon how well the independence assumption holds on a particular XML document. Intuitively, the independence assumption refers to whether u having a child v is independent of whether u has a particular parent/ancestor or other children. To capture the cases that violate the independence assumption, additional information is collected and kept.

There are two cases where the estimation algorithm relies on the independence assumption. The first case happens when there are multiple in-edges and out-edges to a vertex v . The probability of v having a child, say w , is independent of which node is the parent

of v . This assumption ignores the possible correlations between ancestor and descendants. This case is best illustrated by the following example.

Example 5.4 Given the XSEED kernel depicted in Figure 5.5, the optimizer wants to estimate the cardinality of $\mathbf{b/d/e}$. Since vertex \mathbf{d} in the graph has two in-edges incident to \mathbf{b} and \mathbf{c} , the estimation algorithm assumes that the total number of \mathbf{e} nodes (20) from \mathbf{d} nodes is independent of whether \mathbf{d} 's parents are \mathbf{b} nodes or \mathbf{c} nodes. Under this assumption, the cardinality of $\mathbf{b/d/e}$ is the cardinality of $\mathbf{d/e}$ times the proportion of \mathbf{d} nodes that are contributed by \mathbf{b} nodes, namely the forward selectivity of path $p = \mathbf{b/d/e}$:

$$\begin{aligned} |p| &= |\mathbf{d/e}| \times fsel(\mathbf{b/d/e}) \\ &= e_{(d,e)}[0][\mathbf{C_CNT}] \times \frac{e_{(b,d)}[0][\mathbf{C_CNT}]}{e_{(b,d)}[0][\mathbf{C_CNT}] + e_{(c,d)}[0][\mathbf{C_CNT}]} \\ &= 20 \times \frac{5}{14} \approx 7.14. \end{aligned}$$

The estimate of $|\mathbf{b/d/e}|$ is not accurate, due to the ancestor independence assumption. \square

The second type of independence assumption is in the case of branching path queries. If a vertex u in the kernel has two children v and w , estimation algorithm assumes that the number of u nodes that have a child node v is independent of whether or not u also has a child w , ignoring the possible correlations between two siblings.

Example 5.5 Consider the XSEED kernel in Figure 5.5, and the path expression $\mathbf{b/d[f]/e}$. Based on the independence assumption, the cardinality of the path expression $\mathbf{b/d[f]/e}$ is the cardinality of $\mathbf{b/d/e}$ times the proportion of \mathbf{d} elements that have a \mathbf{f} child, namely the backward selectivity of \mathbf{f} in the path $p = \mathbf{b/d/f}$:

$$\begin{aligned} |p| &= |\mathbf{b/d/e}| \times bsel(\mathbf{b/d/f}) \\ &= |\mathbf{b/d/e}| \times \frac{e_{(d,f)}[0][\mathbf{P_CNT}]}{e_{(b,d)}[0][\mathbf{C_CNT}] + e_{(c,d)}[0][\mathbf{C_CNT}]} \\ &= 20 \times \frac{5}{14} \times \frac{4}{14} \approx 2.04. \end{aligned}$$

Again, this estimate is not accurate. \square

A simple solution to this problem is to keep, in what is called the *hyper-edge table* (HET), the actual cardinalities of the simple paths (e.g., **b/d/e**) and the “correlated backward selectivity” of the branching paths (e.g., the backward selectivity of **f** correlated with its sibling **e** that is contributed by the path **b/d** in Example 5.5) when they induce large errors, so that it does not need to be estimated. In principle, the HET serves the same role as a histogram in relational database systems.

HET Construction HET can be pre-computed or populated by the optimizer through query feedback. While constructing the HET through query feedback is relatively straightforward, there are two issues related to pre-computation: (1) although the optimizer can estimate the cardinality using the XSEED kernel, it needs an efficient way to evaluate the actual cardinalities to calculate the errors; and (2) the number of simple paths is usually small, but the number of branching paths is exponential in the number of simple paths. Thus, a heuristic is needed to select a subset of the branching paths to evaluate.

To solve the first issue, the construction algorithm generates the path tree [13] while parsing the XML document (see Figure 5.1). The path tree captures the set of all possible simple paths in the XML tree. While constructing the path tree, the algorithm associates each node with the cardinality and backward selectivity of the rooted simple path leading to this node. Therefore, the actual cardinality of a simple path can be computed efficiently by traversing the path tree. To evaluate the actual cardinality of a branching path, the system uses the Next-of-Kin (NoK) operator [143], which performs tree pattern matching while scanning the data storage (see Figure 5.1) once, and returns the actual cardinality of a branching path.

In order to solve the second issue, two thresholds are introduced to effectively control the number of candidate branching paths. The first threshold is the maximum number of branching predicates MBP in the candidate path expressions. This threshold is the most effective one, since if the only cases considered are those whose branching are restricted at the leaf level, the number of candidate branching paths could be $\sum_{i=1}^n \sum_{j=1}^{\min(f_i, MBP+1)} \binom{f_i}{j}$, where n is the number of nodes in the path tree, and f_i is the fan-out of node v_i in the path tree. MBP should be set to a very small number, say 2 (in which case it is called a *2BP HET*), in order to obtain a reasonable number of candidate

paths. In order to further reduce the candidates, another threshold ($BSEL_THRESHOLD$) is introduced for the backward selectivity of the path tree node to be examined, i.e., if $b_{sel}(v) < BSEL_THRESHOLD$, the algorithm evaluates the actual backward selectivity of the branching paths that have v as a predicate; otherwise v is omitted.

Accordingly, the construction of the hyper-edge table is straightforward: for every node v in the path tree, the estimated cardinality and actual cardinality are calculated. The path is put into a priority queue keyed by the absolute estimation error. If $b_{sel}(v) < BSEL_THRESHOLD$, all branching paths (only at the leaf level) with this node as one of the predicates are enumerated, and the paths are put into the priority queue. To limit the memory consumption of the hyper-edge table, a hashed integer (32 bits) is used to represent the string of path expression. When the hash function is reasonably good, the number of collisions is negligible. The hashed integer serves as a key to the actual cardinality and the correlated backward selectivity of the path expression. Table 5.1 is an example HET for the XSEED kernel in Figure 5.5, where actual hyper-edges rather than hashed values are shown.

hyper-edges	cardinality	correlated b_{sel}
/a/b/d/e	14	0.1
/a/c/d/e	6	0.14
/a/b/d/f	21	0.25
/a/c/d/f	29	0.52
d[e]/f	4	0.35

Table 5.1: Hyper-Edge Table

The HET is managed simply: all the hyper-edges are sorted in descending order of their errors on secondary storage and only the top k entries which have the largest errors are kept in main memory to fill the memory budget. In the experiments, a 1BP hyper-edge table does not take a lot of disk space (less than 500,000 entries in the most complex Treebank data set and less than 1,000 entries for all the other tested data sets), but 2BP and 3BP could be very large for complex data sets.

Cardinality estimation. If the HET is available, the traveler and matcher algorithms need to be modified to exploit the extra information. The following changes apply to the 1BP HET. In the traveler algorithm, the lines 2 to 7 are modified in function EST as follows:

```

1  if HET is available then
2  then  $n\_hsh \leftarrow incHash(hsh, v)$ ;
3      if  $n\_hsh$  is in HET then
4      end
5  else
6       $e \leftarrow GET-EDGE(u, v)$ ;
7      if  $rl < e.label.size()$  then
8          then  $n\_card \leftarrow e[rl][C\_CNT] * fsel$ ;
9               $sum\_cCount \leftarrow TOTAL-CHILDREN(u, old\_rl)$ ;
10              $n\_bsel \leftarrow e[rl][P\_CNT] / sum\_cCount$ ;
11 end

```

This snippet of code guarantees that the actual cardinalities of simple paths are retrieved from the HET. The *incHash* function incrementally computes the hash value of a path: given an old hash value for the path up to the new vertex and the new vertex to be added, the function returns the hash value for the path including the new vertex.

The matcher also needs to be modified to retrieve the correlated backward selectivity from the HET. The following should be inserted after line 11 in function CARD-EST:

```

1  if HET is available and  $q$  is a predicate QTN then
2  then  $p \leftarrow q$ 's parent QTN;
3       $r \leftarrow p$ 's non-predicate child QTN;
4       $hsh \leftarrow incHash("p[q]/r")$ ;
5      if  $hsh$  is in HET then
6          then  $\langle card, bsel \rangle \leftarrow HET.lookup(hsh)$ ;
7  end

```

In this code, the correlated backward selectivity of q and its non-predicate sibling QTN is checked. The parameter to the *incHash* function is the string representation of the branching path $p[q]/r$.

5.5 Experimental Results

The performance of the XSEED synopsis is evaluated in terms of the following: (1) compression ratio of the synopsis on different types of data sets, and (2) accuracy of cardinality estimation for different types of queries.

To evaluate the combined effects of the above two properties, accuracies are compared with different space budgets against a state-of-the-art synopsis structure, TreeSketch [112]. TreeSketch is considered the best synopsis in terms of accuracy for branching path queries, and it subsumes XSketch [110] for structure-only summarization.

Another aspect of the experiments is to investigate the efficiency of the cost estimation function using the synopsis. The running time of the estimation algorithm is reported for different types of queries. The ratios of the estimation times and the actual query processing times are also reported.

These experiments are performed on a dedicated machine with 2GHz Pentium 4 CPU and 1GB memory. The synopsis construction and cardinality estimation are implemented in C++. The TreeSketch code is obtained from its developers. The reported running times for the estimation algorithms are the averages of five runs.

5.5.1 Data sets and workload

Both synthetic and real data sets are tested: DBLP, XMark10 and XMark100 (XMark with 10MB and 100MB of sizes, respectively), and Treebank.05 (randomly chosen 5% of Treebank) as well as the full Treebank, whose structural characteristics are introduced in previous chapters. The basic statistics about the data sets are listed in Table 5.2.

The workload is divided into three categories: simple path (SP), branching path (BP), and complex path (CP), whose definition can be found in Chapter 1. For each data set, a workload generator generates all possible SP queries, and 1,000 random BP and CP queries. To test the effectiveness of HET with different MBP configurations, the generator also

Data sets	Data characteristics			kernel	construction time (mins)	
	total size	# of nodes	rl_{avg}/rl_{max}		XSEED	TreeSketch
DBLP	169 MB	4022548	0 / 1	2.8KB	0.24 + 27	11
XMark10	11 MB	167865	0.04 / 1	2.7KB	0.01 + 0.27	31
XMark100	116 MB	1666315	0.04 / 1	2.7KB	0.1 + 2.7	815
Treebank.05	3.4 MB	121332	1.3 / 8	24.2KB	0.008 + 52	839
Treebank	86 MB	2437666	1.3 / 10	72.7KB	0.168 + 261	DNF

Table 5.2: Characteristics of experimental data sets, their XSEED kernel size, and construction times. (rl_{avg} and rl_{max} represent average and maximum recursion levels, respectively. The construction time of XS is composed of kernel construction time + HET construction time.)

generates workload that have up to 2 branching predicates (2BP and 2CP) and 3 branching predicates (3BP and 3CP) in each step. The randomly generated queries are non-trivial. A sample CP query looks like `//regions/australia/item[shipping]/location`. The full test workload can be found elsewhere [145].

5.5.2 Construction time

For each data set, the time to construct the kernel and HET are measured separately. As described in Section 5.4, branching paths are estimated only for those path tree nodes whose backward selectivity is less than *BSEL_THRESHOLD*. The threshold is set to 0.1 for all the data sets except Treebank, for which it is set to 0.001. The thresholds are manually chosen to trade off between HET construction time and accuracy improvement.

The construction time for XSEED and TreeSketch are given in Table 5.2. In this table, “DNF” indicates that the construction did not finish in 24 hours. The construction time for XSEED consists of the kernel construction time and the 1BP HET construction time (first and second part, respectively). The total construction time is the sum of these two numbers. As shown in the table, XSEED kernel construction time is negligible for all data sets, and the HET construction time is reasonable; overall they are much smaller than TreeSketch.

5.5.3 Accuracy of the synopsis

To evaluate the accuracy of XSEED synopsis, XSEED is again compared with TreeSketch on different types of queries (SP, BP, and CP). Several error metrics are calculated but two are reported here³: Root-Mean-Squared Error (RMSE) and Normalized RMSE (NRMSE), to evaluate the quality of the estimations. The RMSE is defined as $\sqrt{(\sum_{i=1}^n (e_i - a_i)^2)/n}$, where e_i and a_i are the estimated and actual result sizes, respectively, for the i -th query in the workload. The RMSE measures the average error over all queries in the workload. The NRMSE is adopted from [142] and is defined as RMSE/\bar{a} , where $\bar{a} = (\sum_{i=1}^n a_i)/n$. NRMSE is a measure of the average error per unit of accurate result size.

Since TreeSketch synopses cannot be constructed in the time limit on Treebank, Table 5.3 only lists the error metrics on the DBLP, XMark10, XMark100, and Treebank.05. These data sets represent all three data categories: simple, complex with small degree of recursion, and complex with high degree of recursion. The workload is the combined SP, BP, and CP queries. Both programs are tested using 25KB and 50KB memory budgets. XSEED kernel is also tested without HET. For the DBLP and XMark data sets, XSEED only uses 20KB and 25KB memory respectively for the total of kernel and HET, thus their error metrics on 25KB and 50KB are the same. Even without help from the HET, the XSEED kernel outperforms TreeSketch with 50KB memory budget on the XMark and Treebank.05 data sets. The reason is that the TreeSketch synopsis does not recognize recursions in the document, so even though it uses much more memory, the performance is not as good as the recursion-aware XSEED synopsis. When the document is not recursive, TreeSketch has better performance than the bare XSEED kernel. However, spending a small amount of memory on the HET greatly improves performance. The RMSE for XSEED with 25KB (i.e., a small HET) is almost half of the RMSE for TreeSketch with 50KB memory.

There is only one case—BP queries on DBLP (see Figure 5.6)—where TreeSketch outperforms XSEED even with the help of HET. In this case, XSEED errors are caused by the correlations between siblings that are not captured by the HET. For example, the query `/dblp/article[pages]/publisher` causes a large error on XSEED. The reason is

³The Coefficient of Determination (R-squared) and Order Preserving Degree (OPD) are also calculated, but the values are very close to the perfect score for almost all datasets, so these results are omitted here.

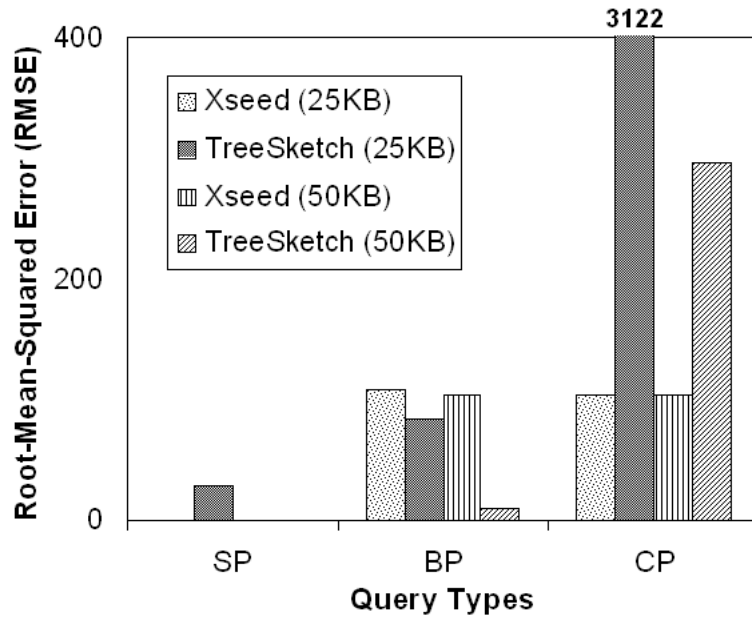


Figure 5.6: Estimation errors for different query types on DBLP

settings		DBLP		XMark10		XMark100		Treebank.05	
		RMSE	NRMSE	RMSE	NRMSE	RMSE	NRMSE	RMSE	NRMSE
XS kernel		1960.5	15.4%	39.6	15.1%	276.15	5.06%	22.7	169%
25KB	XS	103	0.81%	3.737	1.43%	256.3258	4.71%	22.7	169%
	TS	221.5	1.67%	62.738	23.7%	638.1908	11.7%	229.5823	877.14%
50KB	XS	103	0.81%	3.737	1.43%	256.3258	4.71%	12.82	95.61%
	TS	203.09	1.59%	58.3946	22.09%	635.5347	11.65%	227.1157	867.71%

Table 5.3: Error metrics for XSEED (XS) and TreeSketch (TS) in different memory budgets

that the backward selectivity (0.8) of `pages` under `/dblp/article` is above the default `BSEL_THRESHOLD` (0.1), so the hyper-edge `article[pages]/publisher` was omitted in the HET construction step, thus the correlation between `pages` and `publisher` is not captured. It is possible to use better heuristics to address this problem, although it is not investigated further.

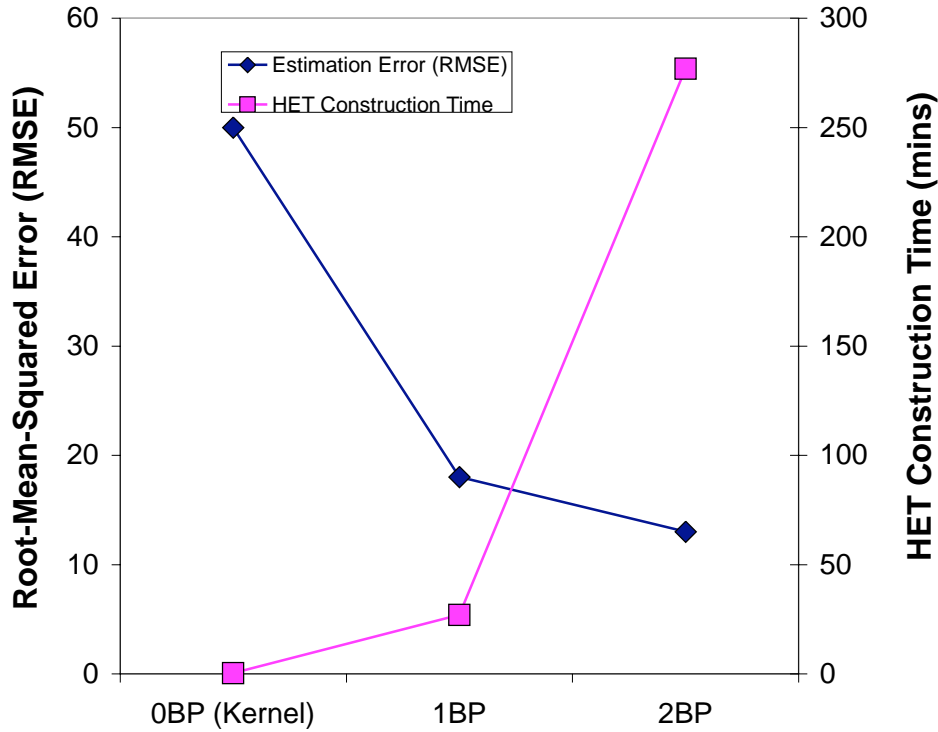


Figure 5.7: Different MBP settings on DBLP

The accuracy of different types of workload (1BP, 2BP and 3BP) are also tested on HETs with different MBP settings ($BSEL_THRESHOLD$ is set to 0.8). The observation is that 1BP HET is usually the best tradeoff between construction time and accuracy. Figure 5.7 shows the HET construction times (on the right y -axis) using different MBP settings for the DBLP dataset, and the estimation errors (on the left y -axis) for each setting on the 2BP workload. The error is reduced significantly (66%) going from no HET to 1BP HET, but the reduction in error from 1BP HET to 2BP HET diminishes to 8%. On the other hand, the construction time of 2BP HET is about 10 times that of 1BP HET.

Performance	DBLP	XMark10	XMark100	Trebank.05	Trebank
EPT to XML tree ratio	0.0035%	0.036%	0.05%	6.9%	5.5%
estimation time to actual running time ratio	0.018%	0.57%	0.0916%	2%	1.5%

Table 5.4: The ratios of EPT to XML tree and estimation time to actual query evaluation time.

5.5.4 Efficiency of cardinality estimation algorithm

To evaluate the efficiency of the cardinality estimation algorithm, the ratio of the time spent on estimating the cardinality and the time spent on actually evaluating the path expression are listed. The path expression evaluator is the NoK operator [143] extended to support `//`-axes.

The efficiency of the cardinality estimation algorithm depends on how many tree nodes there are in the expanded path tree (EPT) that can be generated from traversing the XSEED kernel. For DBLP, XMark10 and XMark100 data sets, the generated EPT is very small—0.0035%, 0.036%, and 0.05% of the original XML tree, respectively. As mentioned previously, the EPT could be large for highly recursive documents such as Trebank.05 and Trebank. To limit the size of EPT, as mentioned earlier, a threshold is established for the estimated cardinality of the next vertex to visit. In these experiments, the threshold is set to 20 (i.e., if the estimated cardinality of the next vertex in depth-first order is less than 20, it will not be visited), and the ratio of EPT size to XML tree size is 6.9% and 5.5%.

The average ratios of the estimation time to the actual query running time on DBLP, XMark10, XMark100, Trebank.05, and Trebank are 0.018%, 0.57%, 0.0916%, 2%, and 1.5%. The detailed EPT to XML tree ratio and the estimation running time vs. actual query evaluation time ratios are listed in Table 5.4. The ratios for XMark10 and XMark100 differ significantly because their XSEED kernels are very similar, but the size of the XML documents differs by a factor of 10.

5.6 Comparisons to Related Work

As discussed in Chapter 2, there are many approaches dealing with cardinality estimation for path queries (e.g., [64, 37, 13, 110, 132, 14, 112, 130]). Some of them [64, 13, 132, 130] focus only on a subset of the possible path expressions, e.g., simple paths (linear chain of steps that are connected by /-axis) or linear paths containing //-axes. Moreover, none of them directly addresses recursive data sets, and only [64] and [130] support incremental maintenance of the synopsis structures.

TreeSketch [112], an extension to XSketch [110], can estimate the cardinality of branching path queries very accurately in many cases. However, it does not perform as well on recursive data sets. Also, due to the complexity of the construction process, TreeSketch is not practical for structure-rich data such as Treebank. XSEED has similarities to TreeSketch, but the major difference is that XSEED preserves structural information in two layers of granularity (kernel and HET); while TreeSketch tries to preserve this information in a complex and unified structure.

The hyper-edge table has been inspired by previous proposals [14, 130]. In [14], the actual statistics of previous queries are recorded into a table and reused later. In [130], a Bloom Filter is used to store cardinality information about simple paths compactly. In this chapter, one hash value is used for that purpose, since practice shows that a good hash function produces very few collisions for thousands of paths.

5.7 Conclusion

This chapter presents a compact synopsis structure to estimate the cardinalities of path queries. XSEED is the first to support accurate estimation for all types of queries and data, incremental update of the synopsis when the underlying XML document is changed, dynamic reconfiguration of the synopsis according to the memory budget, and the ability to exploit query feedback. The simplicity and flexibility of XSEED make it well suited for implementation in a real DBMS optimizer.

Chapter 6

Conclusion and Future Work

6.1 Summary and Contributions

This thesis studies three important issues in managing large volumes of XML data. Novel techniques are proposed for the storage system, query evaluation engine, and cost-based optimizer.

The first issue that is studied is the storage system for large volumes of arbitrarily complex XML documents. The storage system should be able to handle documents with or without a schema and preserve the orders in the documents. These are the basic requirements for a general purpose XML storage system. Furthermore, the storage system should support efficient evaluation and update. I/O performance is crucial to query evaluation and update. Therefore, the storage system should be designed and optimized towards more efficient query evaluation and update. Another desirable feature for the storage system is the space efficiency (succinctness). This is particularly important when the amount of data is significantly large. Previous approaches focus some of the above requirements. The proposed storage scheme addresses all of them [143]. The novelties of the storage system are the following:

- The proposed storage scheme can store any XML document tree, regardless of whether it has a schema or not. In this storage scheme, the tree structural and value information are separately stored and indexes are built to maintain their connections. This

design decision is followed by the principle of separation of concerns and results in efficient evaluation and update.

- The structural information is converted into a string which is guaranteed to be reconstructible to the XML tree. This conversion makes the tree data model easily fit into paged I/O model. Simple statistics kept in the page header can significantly reduce I/O for tree traversing during query evaluation. Furthermore, updating the tree can also be translated into updating the string, for which efficient algorithms are well studied.
- Values of all elements are stored sequentially and ordered by their structural ID (Dewey ID). The Dewey IDs can be easily derived from the structural storage. Therefore, Dewey IDs serve as the keys to connect a value to a certain XML tree node.

Building on the storage system, the thesis proposes a heuristic evaluation strategy for a general path expression. The heuristics is based on the observation that different types of axes can be evaluated by different approaches: local axes by the navigational approach and global axes by the join-based approach. A hybrid evaluation strategy is proposed to first convert a path expression consisting of backward axes to an expression having only forward axes. Then the path expression is decomposed into NoK pattern trees that consists of only local axes. Finally, each NoK expression is evaluated and their intermediate results are structurally joined. The novelties of this strategy and the NoK pattern matching algorithms are as follows:

- The thesis proves that any backward axis can be translated into a pattern tree that consists of only forward axes. This translation makes a one-pass algorithm possible on the input XML data.
- The definition of NoK pattern tree and the heuristics of decomposing a query tree into NoK pattern trees are novel. With this decomposition-and-join approach, it is possible to take the best of both worlds. Experiments justify that the heuristics results in efficient evaluation plans in most cases.

- The NoK pattern matching algorithm is a novel data-driven navigational query processing operator. With the support of the storage system, NoK pattern matching can be evaluated efficiently. Comparing to previous navigational approach, NoK pattern matching algorithm results in simpler and more efficient implementation due to the constraints on the path expressions.

Indexing techniques are proposed to prune the input before applying the NoK pattern matching. As discussed in Chapter 2, path expressions contain both structural and value constraints. Previous research concentrates on either structural indexes or value indexes. In this work, a feature-based index, FIX, is proposed to combine both structural and value information. The contributions in this regard are as follows:

- A list of features consisting of real numbers are identified. The features can be extracted automatically from sub-patterns in database and query trees.
- The thesis proves a sufficient condition for the pattern tree begin an inducted sub-graph of sub-patterns in the index. This condition is the basis for the pruning power of the index, and they do not introduce false-positives.
- The FIX index provides a natural and effective way to index values in the XML documents. Integrating values into the structural index eliminates the need for two index look-up operations and intersection of the temporary results.

The FIX index proposes a general technique to prune the input subtree. Although it is tested by combining with the NoK processor as the refinement operator, it is possible to be combined with other path expression evaluation operators.

The last part of the thesis addresses cardinality estimation of path expressions. Cardinality estimation is crucial in cost-based optimization. Previous approaches concentrate on one or two criteria such as accuracy, or adaptiveness. In addition, a synopsis should address multiple criteria, including efficiency in construction and estimation, updatability, and robustness. This thesis proposes XSEED to address all these criteria:

- XSEED follows a two-layer design that enables it to be adaptable to memory budget. A small kernel can be constructed very efficiently and it captures the basic structural

information. On top of that layer, hyper-edge table (HET), provides additional information about the tree structure. It enhances the accuracy of the synopsis and makes it adaptive to different memory budgets.

- The simplicity of the kernel makes the synopsis robust, space efficient, and easy to construct and update.
- XSEED presents a novel and efficient algorithm for traversing the synopsis structure to calculate the estimates. The algorithm is highly efficient and is well suited to be embedded in a cost-based optimizer.

Due to its robustness, efficiency, adaptivity, and accuracy, XSEED is a practical synopsis well-suited to a database systems.

6.2 Integrating the Techniques into a System

As depicted in Figure 1.5, all the individual modules introduced in previous chapters need to be integrated into a system. This section introduces, from bottom-up, the integration of the existing modules and possible solutions to the missing pieces.

The preprocessing phase of the XDB consists of parsing the XML documents into native storage, constructing indexes, and summarizing the documents into synopses. The detailed construction steps are already introduced in Chapters 3–5. This section introduces how all these techniques are tied together in a system. The native storage and synopses are constructed by parsing the XML documents once, since both construction algorithms are based on the SAX-event API. Indexes are optional and they are constructed based on the native storage. As presented in Chapter 3, the storage is separated into a structural part and a value part. Constructing a pure structural FIX index needs to scan the structural part only once, while a unified structural and value index needs the input from both parts. Scans on two parts are synchronized so that they both read the input only once.

In the querying phase, while the navigational operator only scans the native storage, the index operator may read input from both the native storage and the index. As discussed earlier, the non-leaf nodes in FIX index contain the feature lists (eigenvalues and root labels) while the leaf nodes are pointers to the native storage for the unclustered indexes, or pointers to the clustered subtrees for the clustered indexes. The pruning phase only

involves the index, and once a set of candidate pointers are obtained from the index, the execution is handed over to the navigational operator. Then it is up to the navigational operator to decide which part(s) of the native storage—structural only or structural and value parts—need to be referenced.

On top of the execution engine, the optimizer needs to decide, from among the multiple operators, which operator is the optimal one. While cardinalities of the expression or its subexpressions are important parameters, some intelligence is needed for the optimizer to make the right decision.

One type of intelligence is that the optimizer needs to know how to map the input parameters to the cost (i.e., the cost model), since the cost is the fundamental metric for the optimizer to compare different execution plans. More precisely, a cost model is a function mapping from the *a priori* parameters to a number—the estimated execution time. The *a priori* parameters are usually the number of certain expensive operations. For relational operators, the most expensive operations are usually I/O operations, e.g., page reads and writes. Therefore, developing cost models for the relational operators usually involves analyzing the source code and determining the number of I/O calls based on the knowledge of statistics maintained in the system catalogue. Unfortunately, developing cost models of XML query processing is much harder than developing cost models of relational query processing. The reason is that the data access patterns for these relational operators can often be predicted and modeled in a fairly straightforward way. The data access patterns of complex XML query operators, on the other hand, tend to be markedly non-sequential and therefore quite difficult to model. For these reasons, a statistical learning approach, such as COMET [142], can be used to model the cost of complex XML operators (XNav in [142]). The basic idea is to identify a set of query and data “features” (input parameters to the cost model) that determine the operator cost. Using training data, COMET then automatically learns the functional relationship between the feature values and the cost—the resulting cost function is then applied at optimization time to estimate the cost of XML operators for incoming production queries.

Although COMET is not implemented in the XDB project, the basic idea can be applied to develop the cost models of the physical operators in XDB. There are two key issues in developing cost models using the COMET methodology: (1) proposing the right features

for the cost model, and (2) proposing the right statistics for estimating the feature values. Both issues require deep understanding of the operators' algorithms: features are usually discovered and approximated by analyzing the algorithms. Although the algorithm of the NoK pattern matching operator is different from that of the XNav operator, they are based on similar ideas: both are input-driven automata whose states are constructed dynamically. Therefore, the features and feature value estimations for NoK should be very similar to those for XNav.

Developing a cost model for the FIX index operator could be more complex. As discussed in Chapter 4, the FIX evaluation algorithm can be decomposed into two phases: the index pruning phase based on B^+ tree and the refinement phase using the NoK operator. Assuming the cost model for the NoK operator is available, the cost of the FIX index operator is dependent on the cost model of the index searching in B^+ tree and the number of NoK pattern matching operations. The cost model for a B^+ tree is well-studied (see, e.g., [44]). The challenge lies in the refinement phase: i.e., how to estimate the number of NoK pattern matchings and what is the cost of each of them. It seems straightforward if the cost model for NoK is already developed. However, the I/O access pattern of the index operator is different from that of the non-indexed NoK operator. As a result, the statistics proposed for the non-index NoK operator is insufficient to estimate the feature values. The reason is that some subtrees are pruned from the index and there are no sufficient statistics to estimate the cost of NoK pattern matchings on the remaining ones. Fortunately, the number of remaining subtrees can be estimated fairly easily by using existing techniques such as a histogram on the features indexed in the B^+ tree. A rough estimation of the cost of the refinement phase would be using the proportion of remaining subtrees as the indicator to the savings in cost from that of the non-indexed NoK operator. It is still open, however, whether this estimation is accurate enough, and a more precise cost estimation, particularly the right statistics, remains a challenge.

Besides cost models, other missing pieces in the optimizer are the plan generation and exploration. These, however, have little difference from those found in the relational database systems. Mature techniques (see, e.g., [69]) can be applied here.

6.3 Future Work

There is much work that needs to be done for XDB to be a properly working system. Besides the issues mentioned in the previous section, several techniques need further investigation.

- Currently the XDB system only supports a subset of the XML data model and XML query languages. The important missing features are IDs and IDREFs in the XML documents, and value and positional constraints in the path expressions. The system and the techniques should be extended to support these features.
- Any storage system should consider how to employ concurrency control and how it affects the update process. As the update semantics become defined, the storage system should be enhanced to support updates and concurrent access.
- FIX is designed to be working as a general pruning index. In addition to the navigational operator, a join-based operator should also be able to act as a refinement operator.
- Since the key of FIX index is a list of numbers, a high-dimensional index rather than a B⁺ tree could be more beneficial.
- XSEED synopsis currently only supports structural constraints, an extension to support value constraints is highly desirable.
- The XSEED kernel is currently of fixed size once it is constructed. When the memory budget is higher, additional information is stored in the hyper-edge table. Maintaining these two data structures is complex, therefore it may be desirable to change the kernel itself when additional knowledge about the data is added to the synopsis.

The emergence and increasing applications of Web services have generated many interesting problems and has attracted significant interests from both academic and industrial communities. As mentioned in Chapter 1, messages or data exchanged between Web services are XML documents. These documents may have a small size and simple structure, but may have a very high arrival rate at a popular Web service server. Furthermore, the

stream of XML messages sent by the network tend to be infinite, which is completely different from the scenario where XML data are archived before querying. These problems are typically studied in the context of stream data management [18, 62]. Some constraints are posed on the stream processing model: (1) join-based evaluation techniques is not desirable anymore since they require preprocessing the XML document to encode the elements before the joins; (2) indexing techniques that require preprocessing also do not work since the construction time and update cost will diminish the benefit of using indexes; and (3) synopsis structures need to be changed to facilitate the fact that data are updated frequently. Based on the proposed techniques in this thesis, the following possible solutions should be investigated:

- The storage structure needs to be modified so that it can handle high arrival rate of XML documents. The current storage system separates structural and value information for archiving. This may not be desirable if the documents cannot be archived and the arrival rate does not allow a server to do so. A new string-based storage model (in the stream case, it should be main memory based instead of disk based) should be designed for supporting efficient evaluation.
- The navigational operator developed in this thesis is well-suited to the streaming processing environment. However, the FIX index is no longer available in this scenario. Designing an index for more efficient evaluation, particularly when many online queries are submitted to the system.
- Under the assumption that XML documents in Web services have simple structures, the ability to deal with complex structures is no longer the first priority. The major problem is how to support frequent updates. While the XSEED synopsis supports update, it is not designed for frequent update. The performance and tradeoff between update and accuracy is a major concern.

Bibliography

- [1] Anatomy of a Native XML Base Management System. BerkeleyDB XML Whitepaper, available at <http://www.sleepycat.com/products/bdbxml.html>.
- [2] DocBook. Available at <http://www.oasis-open.org/docbook/>.
- [3] Document Type Definition (DTD). Available at <http://www.w3.org/TR/REC-xml>.
- [4] MPEG-7 Overview. Available at <http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm>.
- [5] Simple Object Protocol (SOAP). Available at <http://www.w3.org/TR/soap/>.
- [6] Universal Description, Discovery and Integration (UDDI). Available at <http://www.uddi.org>.
- [7] University of Washington XML Data Repository. Available at <http://www.cs.washington.edu/research/xmldatasets/www/repository.html>.
- [8] Web Services Architecture. Available at <http://www.w3.org/TR/ws-arch/>.
- [9] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Available at <http://www.w3.org/TR/xhtml1/>.
- [10] Extensible Markup Language (XML) 1.0 (Second Edition). Available at <http://www.w3.org/TR/REC-xml>, October 2000.
- [11] XML Schema. Available at <http://www.w3.org/XML/Schema>, May 2001.

- [12] S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 122–133, 1997.
- [13] A. Abounaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 591–600, 2001.
- [14] A. Abounaga and J. F. Naughton. Building XML Statistics for the Hidden Web. In *Proc. 12th Int. Conf. on Information and Knowledge Management*, pages 358–365, 2003.
- [15] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. 18th Int. Conf. on Data Engineering*, pages 141–152, 2002.
- [16] M. Arenas and L. Libkin. A Normal Form for XML Documents. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 85–96, 2002.
- [17] M. Arenas and L. Libkin. An information-theoretic approach to normal forms for relational and XML data. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 15–26, 2003.
- [18] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–16, 2002.
- [19] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirement of Evaluating XPath Queries over XML Streams. In *Proc. 23rd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 177–188, 2004.
- [20] A. Barta, M. P. Consens, and A. O. Mendelzon. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 133–144, 2005.

- [21] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *Proc. 19th Int. Conf. on Data Engineering*, pages 455–466, 2003.
- [22] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. Available at <http://www.w3.org/TR/xpath20/>.
- [23] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: One Part Relational, One Part XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 347–358, 2005.
- [24] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Available at <http://www.w3.org/TR/xquery/>.
- [25] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proc. 18th Int. Conf. on Data Engineering*, pages 64–75, 2002.
- [26] B. Bollobás. *Modern Graph Theory*. Springer, 1998.
- [27] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. 21st Int. Conf. on Data Engineering*, pages 705–716, 2005.
- [28] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–322, 2002.
- [29] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on Regular Path Queries. *ACM SIGMOD Record*, 32(4):83–92, 2003.
- [30] M. J. Carey. XML in the middle: XQuery in the WebLogic Platform. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 901–902, 2004.

- [31] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 383–394, 1994.
- [32] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. Available at <http://www.w3.org/TR/xmlquery-use-cases>.
- [33] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An Adaptive Structural Summary for Graph-Structured Data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 134–144, 2003.
- [34] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Index Techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 455–466, 2005.
- [35] Y. Chen, G. Mihaila, R. Bordawekar, and S. Padmanabhan. L-Tree: A Dynamic Labeling Structure for Ordered XML Data. In *EDBT Ph.D. Workshop*, pages 209–218, 2004.
- [36] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar. Labeling Your XML. preliminary version presented at CASCON’02, October 2002.
- [37] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *Proc. 17th Int. Conf. on Data Engineering*, pages 595–604, 2001.
- [38] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 263–274, 2002.
- [39] B. Choi. What are Real DTDs Like. In *Proc. 5th Int. Workshop on the World Wide Web and Databases (WebDB)*, pages 43–48, 2002.
- [40] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Available at <http://www.w3.org/TR/wsdl>.

- [41] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.
- [42] J. Clark. XSL Transformations (XSLT) Version 1.0. Available at <http://www.w3.org/TR/xslt>.
- [43] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 271–281, 2002.
- [44] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [45] M. P. Consens and T. Milo. Algebras for Querying Text Regions: Expressive Power and Optimization. *Journal of Computer and System Sciences*, 57:272–288, 1998.
- [46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [47] D. M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs: Theory and Application*. Academic Press, 1979.
- [48] D. DeHann, D. Toman, M. P. Consens, and M. T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 623–634, 2003.
- [49] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, 1999.
- [50] M. Dubiner, Z. Gallil, and E. Magen. Faster Tree Pattern Matching. *J. ACM*, 41(2):205–213, 1994.
- [51] H. Eves. *Elementary Matrix Theory*. Allyn and Bacon, Inc, 1966.
- [52] W. Fan and J. Siméon. Integrity Constraints for XML. In *Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 23–34, 2000.

- [53] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics. Available at <http://www.w3.org/TR/query-semantics/>.
- [54] M. Fernandez, J. Marsh, and M. Nagy. XQuery 1.0 and XPath 2.0 Data Model. Available at <http://www.w3.org/TR/query-datamodel/>.
- [55] M. F. Fernandez, J. Siméon, and P. Wadler. An Algebra for XML Query. In *Proc. of the 20th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 11–45, December 13–15 2000.
- [56] T. Fiebig, S. Helmer, C. Kanne, J. Mildenerger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. Technical report, University of Mannheim, 2002.
- [57] T. Fiebig, S. Helmer, C.-C. Kanne, J. Mildenerger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *The VLDB Journal*, 11(4):292–314, 2002.
- [58] D. Florescu and D. Kossman. Storing and Query XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
- [59] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–191, 2002.
- [60] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 41(4):616–641, 2002.
- [61] M. N. Garofalakis and A. Kumar. Correlating XML data streams using tree-edit distance embeddings. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 143–154, 2003.
- [62] L. Golab and M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

- [63] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data model and Query Language. In *Proc. ACM SIGMOD Workshop on The Web and Databases (WebDB)*, pages 25–30, 1999.
- [64] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23th Int. Conf. on Very Large Data Bases*, pages 436–445, 1997.
- [65] G. H. Gonnet and F. W. Tompa. Mind Your Grammar: a New Approach to Modelling Text. In *Proc. 13th Int. Conf. on Very Large Data Bases*, pages 339–346, 1987.
- [66] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 95–106, 2002.
- [67] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath query evaluation. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 179–190, 2003.
- [68] G. Gottlob, C. Koch, and K. Schulz. Conjunctive Queries over Trees. In *Proc. 23rd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 189–200, 2004.
- [69] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.
- [70] T. Grust. Accelerating XPath Location Steps. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 109–121, 2002.
- [71] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 524–535, 2003.
- [72] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Sys.*, 29(1):91–131, 2004.

- [73] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 16–27, 2003.
- [74] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed Mode XML Query Processing. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 225–236, 2003.
- [75] H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. In *Proc. 20th Int. Conf. on Data Engineering*, pages 683–694, 2004.
- [76] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulation on Finite and Infinite Graphs. In *Proc. 36th Annual Symp. on Foundations of Computer Science*, pages 453–462, 1995.
- [77] C. M. Hoffmann and M. J. O’Donell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, 1982.
- [78] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [79] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Papparizos, J. Patel, D. Srivastava, and Y. Wu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.
- [80] H. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. 8th Int. Workshop on Database Programming Languages*, pages 149–164, 2001.
- [81] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. 19th Int. Conf. on Data Engineering*, pages 253–263, 2003.
- [82] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 273–284, 2003.

- [83] G. Jacobson. Space Efficient Static Tree and Graphs. In *Proc. 30th Annual Symp. on Foundations of Computer Science*, pages 549–554, 1989.
- [84] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.
- [85] C. Kanne, M. Brantner, and G. Moerkotte. CostSensitive Reordering of Navigational Primitives. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 742–753, 2005.
- [86] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proc. 16th Int. Conf. on Data Engineering*, pages 198–209, 2000.
- [87] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexing for Branching Path Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 133–144, 2002.
- [88] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *Proc. 18th Int. Conf. on Data Engineering*, pages 129–140, 2002.
- [89] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 3rd edition, 1997.
- [90] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata -based Approach. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 249–260, 2003.
- [91] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. In *Proc. 24th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 84–97, 2005.
- [92] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, and V. Arora. Towards an industrial strength SQL/XML Infrastructure. In *Proc. 21st Int. Conf. on Data Engineering*, pages 991–1000, 2005.

- [93] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating Structured and Semistructured Data. In *Proc. 7th Int. Workshop on Database Programming Languages*, pages 297–323, 1999.
- [94] M. Ley. DBLP XML Repository. Available at <http://dblp.uni-trier.de/xml>; accessed April 2004.
- [95] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 361–370, 2001.
- [96] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery Processing in Oracle XMLDB. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 828–833, 2005.
- [97] D. Maier. Database Desiderata for an XML Query Language. In *The Query Language Workshop*, 1998. Available at <http://www.w3.org/TandS/QL/QL98/pp/maier.html>.
- [98] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.
- [99] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. 7th Int. Conf. on Database Theory*, pages 277–295, 1999.
- [100] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 11–22, 2000.
- [101] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *Proc. 38th Annual Symp. on Foundations of Computer Science*, pages 118–126, 1997.
- [102] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram,

- F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Quarterly Bulletin on Data Engineering*, 24(2):27–33, 2001.
- [103] F. Neven. Automata Theory for XML Researchers. *ACM SIGMOD Record*, 31(3):39–46, 2002.
- [104] M. Nicola and B. V. der Linden. Native XML Support in DB2 Universal Database. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 1164–1174, 2005.
- [105] L. Nie. Efficient XQuery Processing Using B+ Tree Indexes. Master’s thesis, University of Waterloo, 2004.
- [106] D. Olteanu, H. Meuss, tim Furche, and F. Bry. XPath: Looking Foward. In *Proc. of the EDBT Workshop on XML Data Management*, pages 109–127, 2002.
- [107] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan. XQuery Implementation in a Relational Database System. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 1175–1186, 2005.
- [108] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML Data Stored in a Relational Database. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 1134–1145, 2004.
- [109] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. 11th Int. Conf. on Data Engineering*, pages 251–260, 1995.
- [110] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph Structured XML Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 358–369, 2002.
- [111] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 466–477, 2002.

- [112] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274, 2004.
- [113] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity Estimation for XML Twigs. In *Proc. 20th Int. Conf. on Data Engineering*, pages 264–275, 2004.
- [114] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.
- [115] H. Prüfer. Neuer Beweis eines satzes über permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.
- [116] P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proc. 20th Int. Conf. on Data Engineering*, pages 288–300, 2004.
- [117] A. Sahuguet. KWEELT, the Making-of: Mistakes Made and Lessons Learned. Technical report, University of Pennsylvania, November 2000.
- [118] M. Schkolnick. A Clustering Algorithm for Hierarchical Structures. *ACM Trans. Database Sys.*, pages 27–44, 1977.
- [119] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.
- [120] H. Schöning and J. Wäsch. Tamino - An Internet Database System. In *Advances in Database Technology — EDBT'00*, pages 383–387, 2000.
- [121] L. Segoufin. Typing and querying XML documents: some complexity bounds. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 167–178, 2003.
- [122] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Query XML Documents: Limitations and Opportunities. In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 302–314, 1999.

- [123] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 39–53, 2002.
- [124] A. Shokoufandeh, D. Macrini, S. Dickinson, K. Siddiqi, and S. W. Zucker. Indexing Hierarchical Structures Using Graph Spectra. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 27(7):1125–1140, 2005.
- [125] J. Siméon and M. Fernandez. The Galax System: The XQuery Implementation for Discriminating Hackers. Available at <http://www-db-out.bell-labs.com/galax/>.
- [126] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 204–215, 2002.
- [127] V. Vianu. A Web Odyssey: from Codd to XML. In *Proc. 20th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–15, 2001.
- [128] H. Wang and X. Meng. On the Sequencing of Tree Structures for XML Indexing. In *Proc. 21st Int. Conf. on Data Engineering*, pages 372–383, 2005.
- [129] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 110–121, 2003.
- [130] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 240–251, 2004.
- [131] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li. Efficient Processing of XML Path Queries Using the Disk-based FB Index. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 145–156, 2005.
- [132] Y. Wu, J. M. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *Advances in Database Technology — EDBT’02*, pages 590–680, 2002.

- [133] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. 19th Int. Conf. on Data Engineering*, pages 443–454, 2003.
- [134] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 335–346, 2004.
- [135] X. Yan, P. S. Yu, and J. Han. Substructure Similarity Search in Graph Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 766–777, 2005.
- [136] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. 20th Int. Conf. on Data Engineering*, pages 621–633, 2004.
- [137] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Trans. Internet Technology*, 1(1):110–141, 2001.
- [138] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree Signatures for XML Querying and Navigation. In *International XML Database Symposium*, pages 149–163, 2003.
- [139] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 425–436, 2001.
- [140] H. Zhang and F. W. Tompa. XQuery Rewriting at the Relational Algebra Level. *Journal of Computer Systems, Science, and Engineering*, 18(5):241–262, 2003.
- [141] N. Zhang, S. Agarawal, and M. T. Özsu. BlossomTree: Evaluating XPath in FLWOR Expressions. In *Proc. 21st Int. Conf. on Data Engineering*, pages 388–389, 2005.
- [142] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 289–300, 2005.

- [143] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th Int. Conf. on Data Engineering*, pages 54 – 65, 2004.
- [144] N. Zhang and M. T. Özsu. Optimizing Correlated Path Expressions in XML Languages. Technical Report CS-2002-36, University of Waterloo, November 2002. Available at <http://db.uwaterloo.ca/~ddbms/publications/xml/TR-CS-2002-36.pdf>.
- [145] N. Zhang, M. T. Özsu, A. Abounaga, and I. F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. Technical report, University of Waterloo, 2005. Available at http://db.uwaterloo.ca/~ddbms/publications/xml/TR_XSEED.pdf.
- [146] N. Zhang, M. T. Özsu, A. Abounaga, and I. F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proc. 22nd Int. Conf. on Data Engineering*, page 61, 2006.
- [147] N. Zhang, M. T. Özsu, I. F. Ilyas, and A. Abounaga. FIX: Feature-based Indexing Technique for XML Documents. Technical Report CS-2006-07, University of Waterloo, 2006. Available at <http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-07.pdf>.