ToXin: An Indexing Scheme for XML Data

by

Flavio Rizzolo

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Semistructured databases, unlike relational and object-oriented databases, do not have a fixed schema known in advance and stored separately from the data. Broadly speaking, semistructured data is self-describing and can model heterogeneity more naturally than either relational or object-oriented data. Examples of such self-describing data are tagged documents like XML.

Indexing schemes for semistructured data have been developed in recent years to optimize path query processing by summarizing path information. However, most of these indexing schemes can only be applied to some query processing stages whereas others only support a limited class of queries. To overcome these limitations we developed *ToXin*, an indexing scheme for XML data that fully exploits the overall path structure of the database in all query processing stages. ToXin synthesizes ideas from object-oriented path indexes and extends them to the semistructured realm of XML data.

In this thesis we study recent proposals for indexing XML data, present the ToXin architecture, describe its current implementation, and discuss comparative performance results.

Acknowledgments

First and foremost, I would like to thank my advisor Alberto Mendelzon for his invaluable support and advice. Alberto's patience and insight to point out my mistakes forced me to become more rigorous in my reasoning. His guidance was always invaluable, especially at the late stages of my thesis work.

I would also like to thank Renee Miller for being the second reader of my thesis, and the administrative staff of the Department of Computer Science for their permanent willingness to help.

I am also grateful to the Natural Sciences and Engineering Research Council of Canada, and the Department of Computer Science for their generous financial support that made my research possible.

Special thanks go to Jose Maria Turull Torres, my friend and mentor, who first introduced me to the fascinating world of the scientific research and encouraged me to pursue graduate studies. I could never thank Jose Maria enough for his guidance in the first steps of my research career and for being an invaluable source of inspiration. I consider myself lucky for having the chance to work with him.

I also wish to express my gratitude to my officemates Alfredo, Sebastian, and Carlos, and to all my friends in Toronto, especially to Lorena, Hernan, Patricia, Gustavo, Lily, and Diego for helping me and my wife to make this beautiful city our home.

I am also in debt with my parents, Ofelia and Juan Carlos, for helping me to become who I am. I owe them much, and regret that I missed my best opportunities to repay. I dedicate this thesis to them.

Above all, I have to thank a thousand times to my wife, Mariana, the person without whom this thesis could have never been written. Her support and unconditional love are beyond words. It is hard to know who I would be without her; I intend never to have the occasion to find out.

Contents

Chapter 1

Introduction

Traditional database systems have rigid schemas and force all data to conform to them. In contrast, many new applications that appeared in recent years contain data that evolve rapidly or contain irregularities, making the use of a-priori rigid schemas unfeasible. Such data is no longer structured in the traditional sense and is often called semistructured data [Abi97].

Semistructured databases, unlike traditional databases, do not have a fixed schema known in advance and stored separately from the data. Broadly speaking, semistructured data is self-describing and can model heterogeneity more naturally than either relational or object-oriented data. Examples of such self-describing data are tagged documents such as XML [W3C00]. The data model proposed for this type of data consists of an edge-labeled graph in which nodes corresponds to objects or values and edges to elements or attributes. *Figure 1.1* shows a semistructured database modeled as an edge-labeled graph. This data model carries both data and schema information, being naturally suitable to represent semistructured data. In Sections 1.1 and 1.2 we present semistructured data and XML in more detail.

Yet, semistructured data models pose new challenges in many areas. Let us consider query evaluation for instance. Navigation over a semistructured graph is a fundamental part

of query evaluation. Due to the lack of information about the schema, a naïve evaluation that scans the whole database in the search of those paths that satisfy a given query is prohibitively expensive. In addition, since navigating the graph is essentially pointer traversal and the objects may be scattered across the disk or even stored at different locations, some queries may require many disk accesses and cause significant performance degradation.

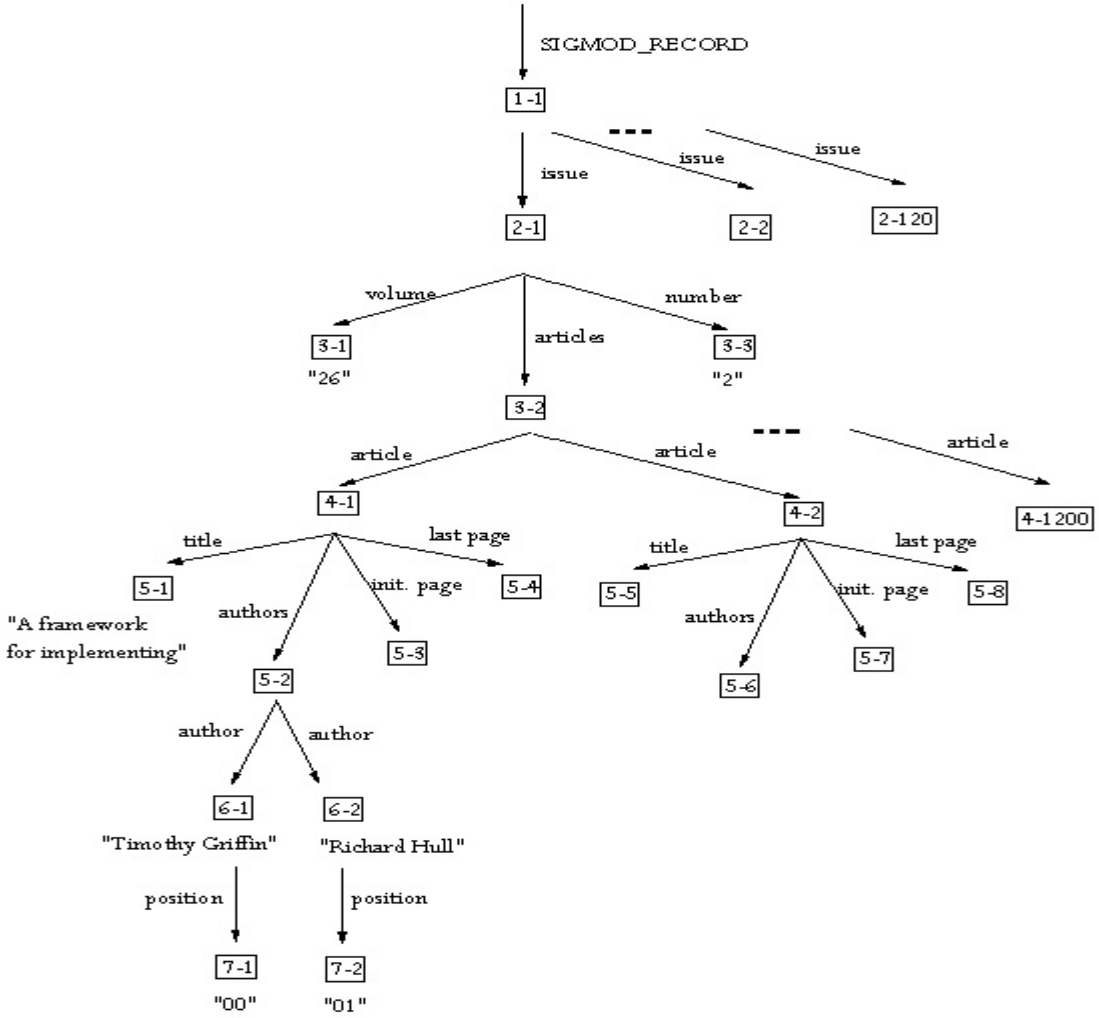


Figure 1.1: The SIGMOD Record database.

Index structures for semistructured data have been developed in recent years in order to address these problems by reducing the portion of the database to be scanned during query processing. Examples of such index structures are dataguides [GW97], 1-indexes and 2-indexes [MS99], T-indexes [MS99], and reversed dataguides [LS00]. They keep record of existing paths in the database summarizing path information. We give a brief overview of these methods here, and discuss some of them in detail in Chapter 2.

In object-oriented databases, several indexing schemes have been proposed for answering path queries efficiently, namely path indexes [BK89, SB96] and Access Support Relations [KM90, KM92], which materialize frequently traversed paths in the database. Since these approaches are based on the paths found in the schema, it is not possible to use them for the typical schema-less XML documents.

Dataguides, on the one hand, are general path indexes that summarize all paths in the database that start from the root. *Figure 1.2* shows a dataguide that corresponds to the *Figure 1.1* database. T-indexes, on the other hand, are specialized path indexes that only summarize a limited class of paths specified by a given path template.

Dataguides and 1-indexes reduce the portion of the database to be scanned for path queries and are useful for navigating the semistructured graph from the root. However, since they do not provide any information about the parent-child relationship between source nodes, they cannot be used for navigation from any arbitrary node. For example, from *Figure 1.2* we know all the objects reached by article.authors.author, but we cannot tell exactly the author objects that correspond to each article object. As a result, in the case of general path queries that require at some point backward navigation, e.g., “find all articles written by a given author”, we need to use additional index structures to optimize the query evaluation.

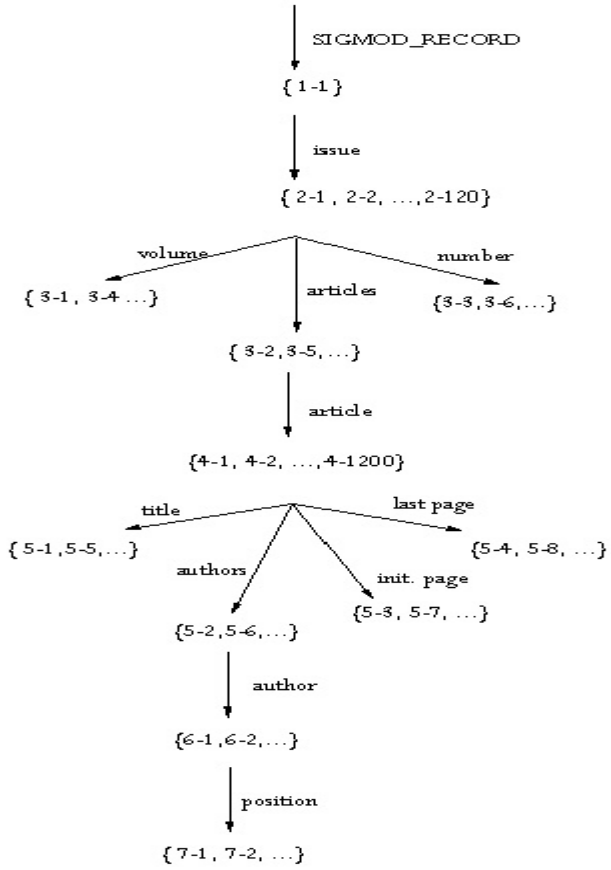


Figure 1.2: A dataguide for the SIGMOD Record database.

The Lore system [MAG+97] attempts to address this problem by using two additional indexes for backward navigation. These indexes, however, do not exploit the database structure as dataguides do: the nodes, edges and values are stored regardless of the overall path schema of the database. Although using such indexes reduces to some degree the number of pointer traversal operations for backward navigation, the forward navigation from any node remains unsupported. Reversed dataguides are also helpful for backward navigation from the leaves of the tree, but they do not support either forward or backward navigation from a generic node.

The goal of this thesis is to synthesize some ideas from dataguides and Access Support Relations and extend them so that we can optimize both forward and backward navigation for any general path query over semistructured data such as XML. To that end we have developed *ToXin*[1], an indexing scheme that fully exploit the overall path structure of the database in all query processing stages. ToXin summarizes all paths in the database and can be used for both forward and backward navigation starting from any node. We provide as well experimental results based on our implementation for tree data sources.

1.1 Semistructured data

In recent years there has been an increasing number of applications that needs to handle fast evolving, irregular data which does not conform to traditional, rigid data models. We use the term semistructured data to refer to data that presents such characteristics [ABS99].

Semistructured data conforms to no fixed schema known in advance and stored separately, being better suited for modeling heterogeneity than either relational or object-oriented data models. Examples of such self-describing data are tagged documents such as XML. In addition to these data-embedded schemas, semistructured data may present some of the following characteristics:

- The schema is large w.r.t. the size of the data. This is a direct consequence of the data heterogeneity, contrasting with relational or object-oriented data in which the schema is often many orders of magnitude smaller than the data.
- The schema is not static. Since the schema may be implicit in the data, updating the schema is as easy as updating the data itself. For instance, in a tagged document containing bibliographical information, each new addition not only has information about

[1] ToXin was developed within the ToX (Toronto XML Engine) project at University of Toronto.

the publication but also the corresponding tags. This means that adding publications to the database may cause implicit schema updates if the new publication does not conform to the previous structure.

- The schema is descriptive rather than prescriptive. Therefore, it has to be recomputed or incrementally updated every time there is a change in the data.

Semistructured data arises from a wide range of applications such as integration of heterogeneous sources, modeling biological databases, digital libraries, and the World Wide Web. Several aspects of managing semistructured data have been extensively studied over the past several years. This research has addressed, among others, data models [BDFS97, CACS94], query languages [AM98, AQM+97, BDHS96, FFLS97, BFS00], and query processing and optimization [CCM96, FS98, MW97].

Most semistructured data models organize data into directed graphs, where each vertex represents an object and each labeled edge represents a relationship between objects. The data in the graph is self-describing. Some objects are atomic and others are complex. The value of an atomic object is one of the base types (e.g., integer, real, string, etc) whereas the value of a complex object is a set of object identifiers. The object identifiers (OIDs) are used to reference nodes in the data graph. They are pointers to either memory or disk locations where nodes are stored. In the web context, they may even be URLs or any other means of locating nodes distributed across several sites. In the XML context, however, we restrict the types of atomic objects to strings.

1.2 XML

Extensible Markup Language (XML) is fast emerging as the standard for representing and interchanging data over the Web. It is a hierarchical data format derived from SGML that

models a document as an augmented tree structure. In contrast to HTML, which is also derived from SGML, XML has been designed as a general data structuring language capable of modeling not only standard documents but also semistructured data in general. Another feature that makes XML more powerful for general data management applications is the separation between content and presentation. XML deals only with content and therefore allows changes in the formatting aspects of a document without modifying its underlying structure.

We will briefly discuss next the concept of document *markup*. Broadly speaking, markup is information about the document schema embedded in the document content in a way that can be identified during the interpretation of the document. In that sense, we need to define some kind of special character or string so that we can distinguish the schema from the data. Such special characters or strings are called markup.

Using markup, structural information is defined in terms of *elements*, the basic components of an XML document. In order to identify such elements as schema rather than data, the element names must appear between *markup delimiters*, which mark the start and end of the markup for an element. Each element name together with its markup delimiters is called a *tag*. The markup delimiters in XML are the characters "<" and ">". *Figure 1.3* shows an XML document for the SIGMOD record database depicted in *Figure 1.1*. The strings <issue> and <article> are tags and therefore part of the document schema. A pair of tags containing the element name delimits each element. The first one, which has the format <element_name> is called *start tag*, whereas the second has the format </element_name> and is called *end tag*. The string between a start tag and an end tag is called an *element content* or *value*. For instance, title in *Figure 1.3* is an element delimited by the start tag <title> and the end tag </title>, and its value is the string "A Framework for Implementing Hypothetical Queries".

In order to append additional element information without including it in the element content, XML defines an additional component: *attributes*. Attributes allow us to include any additional information within an element start tag. In *Figure 1.3*, position is an attribute of the author element and contains additional information regarding the order in which the authors appear in a given publication.

```
<SigmodRecord>
<issue>
    <volume>26</volume>
    <number>2</number>
    <articles>
        <article>
            <title>A Framework for Implementing Hypothetical Queries</title>
            <initPage>231</initPage>
            <endPage>242</endPage>
            <authors>
                <author position="00">Timothy Griffin</author>
                <author position="01">Richard Hull</author>
            </authors>
        </article>
        <article>
            <title>A Toolkit for Negotiation Support Interfaces to Multi-Dimensional Data.</title>
            <initPage>348</initPage>
            <endPage>356</endPage>
            <authors>
                <author position="00">Michael Gebhardt</author>
                <author position="01">Matthias Jarke</author>
                <author position="02">Stephan Jacobs</author>
            </authors>
        </article>
        …
    </articles>
</issue>
...
</SigmodRecord>
```

Figure 1.3: XML document of the SIGMOD Record database.

We must point out that element names are not necessarily unique, e.g., the authors element in our example has two nested author elements. Attribute names, in contrast, are unique within a given element, e.g., one author element cannot have more than one position attribute.

But not all attributes are created equal. XML defines two particular attributes, usually called *ID* and *IDREF*, associated to unique identifiers so that they may be used to link elements beyond the relationship given by the tree structure of the document. This mechanism allows us to define documents that have a graph structure rather than a tree.

Well-formed and Valid XML documents

XML documents must satisfy syntactic constraints. An XML document is called *well-formed* if it is syntactically correct, in other words, if all its elements are neatly nested and each attributes is unique within each element. For instance, we need to end an articles element before ending an issue element. A *valid* document, on the other hand, is a well-formed XML document that has also been validated against a *DTD*. A DTD [W3C98] is a context-free grammar for the document and may be used as an external schema.

XML and Semistructured data models

The XML data model is similar to that for semistructured data in the sense that both represent data as a directed graph that can be interpreted either as *node-labeled* or *edge-labeled*. An important difference, however, is that the XML data model has order while semistructured data is usually unordered. In addition, XML has a component usually not present in the semistructured data model: the attributes. As we discussed before, XML

defines attributes as being part of elements rather than having a separate identity by themselves. Yet, an XML document can be modeled as a semistructured data edge-labeled graph by simply considering XML attributes as edges pointing to atomic objects. Therefore, both XML elements and attributes are described as edges. Since both models are equivalent to represent XML data, we choose the edge-labeled model because it is the prevalent one for semistructured data.

Many important issues about XML have been studied from the database point of view, in particular query languages [BC00] and optimization techniques [Lie99, NW99].

1.3 Path queries and XSLT

An *XSLT stylesheet* [W3C99b] is a set of transformation rules that allows the transformation of one XML document into another. Each rule consists of a *pattern* and a *template*. The *XSLT processor* identifies the nodes to which the template will be applied by using the pattern. During stylesheet processing, the pattern is matched against the nodes of the XML source and the template is instantiated to produce the XML result. It works as follows. The XSLT processor starts from the root and tries to apply the pattern to that node. A pattern specifies the conditions that a given node must satisfy in order to be processed. A node that satisfies the condition matches the pattern; similarly, a node that does not satisfy the condition does not match the pattern. When a node matches the pattern, the XSL processor executes the template of that rule. Each template element specifies one transformation rule. The pattern of that rule is specified by the match attribute. The template used to produce a tree fragment is described in the content of the template element, also called a rule body. The process is then repeated over each of the remaining nodes recursively in a breadth-first fashion.

XSL patterns are specified by XPath [W3C99a], a simple query language for identifying nodes in an XML document based on their names and values as well as the relationship between nodes. Paths may be specified as absolute or relative. Absolute paths are those that start from the root of the tree, whereas relative paths start from any given node defined as context node. The hierarchical relationship between nodes are expressed by operators "/" and "//". The former specifies a parent-child relationship between two nodes whereas the later specifies an ancestor-descendant relationship between two nodes at any depth. For instance, the query "*find all the titles of articles that appear in the database*" can be expressed in XPath as //article/title. The section between brackets of a XPath expression is called a *filter section*. A filter is a predicate that is applied to the nodes that match the path expression before it (i.e. the pre-filter section). The evaluation of the rest of the pattern (i.e., post-filter section) continues only for those nodes that matches the pre-filter section and satisfies the filter. XPath also includes a wildcard operator "*", that matches any node at its location in the path expression, and a disjunction operator "|". *Figure 1.4* shows an XSLT stylesheet that transforms the XML document from *Figure 1.3* into another XML document that contains only the title of the papers that appeared in the issue Number 2 of Volume 26. The content of the match attribute in the second rule is an XSL pattern. The xsl:apply-templates directive invokes the application of templates (the second rule in *Figure 1.3* example). The xsl:value-of directive constructs the result with information from the XSL pattern.

```
<xsl: template>
    <xsl:apply-templates>
</xsl: template>
<xsl: template match="//issue[/volume=26 and number=2]/articles/article/title" >
    <result>
        <xsl:value-of/>
    </result>
</xsl: template>
```

Figure 1.4 XSLT stylesheet

Let us see how the *Figure 1.4* stylesheet is applied to the *Figure 1.3* database. First, the XSLT processor starts the evaluation from <SIGMOD_Record> and attempts to match the pattern to it. When a rule does not have a match attribute, like the first one, the template matches any node. The next step is evaluating the body of the matching rules, only the first one in this case. The rule body has only one instruction, <xsl: apply-templates**/>**, which implies the recursive processing of the subelements of all matched elements, i.e. volume, number, and articles. Once again, only the first rule matches and we repeat the process on the elements that have further nested structure, i.e. the element articles. After two more steps we reach a node that matches the second rule: title. Applying the rule's body, the XSLT processor generates a result element with the value of the current node: "A Framework for Implementing Hypothetical Queries". The process described above is then repeated over the remaining articles. The result of the query is shown in *Figure 1.5*.

<result> A Framework for Implementing Hypothetical Queries. </result>
<result> A Toolkit for Negotiation Support Interfaces to Multi-Dimensional.</result>
…

Figure 1.5: Result of the Figure 1.3 XSLT stylesheet

1.4 Motivation for path indexing

In order to process path queries like XPath patterns we need to navigate the graph structure of the XML document. As we said before, since navigating the graph is essentially pointer traversal, a query processing strategy that scans the whole database in the search of those paths that satisfies a given query is very expensive.

In order to illustrate the problem let us consider again the SIGMOD database from *Figure 1.3*. Let us suppose that we want to find the volume in which a given article appears. We submit then the following query:

Q1.1= //issue[//title="A Framework for Implementing Hypothetical Queries"]/volume

The query processor will match the query pattern in all possible ways to the data graph. It will traverse it from the root and return all nodes reachable via a path matching the regular expression specified by the pattern. To do that following the naïve approach, it will traverse the entire database in the search of paths that satisfy the regular expression //issue//title from query *Q1.1*. Next, we select those nodes having the value "A Framework for Implementing Hypothetical Queries" out of the nodes reachable via those paths. Then, from each selected node we will first traverse title/article/articles backward and then volume forward to complete the query. If we do not use any auxiliary structure, this back and forth navigation has to be done over the XML source.

On the other hand, if we construct an index precomputing some paths of the graph we may process general path queries much more efficiently. Such index will basically summarize path information of the data graph. For instance, we may want to materialize a view containing all nodes that are reachable via the path SIGMOD_Record/issue/articles/article/title if we know that a query such as Q = //issue//title is posed frequently. Then, we may use that information during query processing rather than navigating the data source. Or we may wish to store every path present in the database along with the set of nodes reachable via the path.

In order to reduce the portion of the database to be scanned, schema-independent index structures have been developed in recent years. We already introduced some of them, which we will describe in the next chapter in more detail.

The reminder of this thesis is organized as follows. Chapter 2 presents the related work. An overview of ToXin and its current implementation is discussed in Chapter 3. Chapter 4 describes the experiment setting and the results. In Chapter 5, we present our conclusions and possible directions for future work.

Chapter 2

Path indexes

In the subsequent chapters, we will need a common formalism to describe ToXin and other related path index schemes. We begin by introducing some definitions.

Definition: let D be an XML document. A XML-graph $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ is a rooted ordered graph induced by D, where N is a set of nodes; $v_o \in N$ is a distinguished node called a root; E is a set of edges containing an edge for each XML element and attribute in D; $A \subset E$ is a distinguished set of edges containing an edge for each attribute in D; ψ is an incidence function mapping E to $N \times N$; Σ is a finite set of XML element and attribute names called an *alphabet*; λ is a labeling function mapping E to Σ, and $<$ is a partial order relation on E such that for every pair of edges e_1 and e_2 emanating from node n, $e_1 < e_2$ iff e_1 occurs before e_2 in n.

Definition: let ε be the empty string, ϕ be the empty set, and Σ be a finite alphabet disjoint from $\{\varepsilon, \phi\}$. We define *regular path expressions* over Σ as follows:

a) The empty string ε, the empty set ϕ, and each $a \in \Sigma$ are regular path expressions.
b) if R_1 and R_2 are regular path expressions, then $R_1 | R_2$, R_1 / R_2 and $(R_1)^*$ are regular path expressions.

Let R_1 and R_2 be regular path expressions over Σ. The expression $(R_1|R_2)$ is the *alternation* of R_1 and R_2, (R_1/ R_2) is the *concatenation* of R_1 and R_2, and $(R_1)^*$ is the kleene closure of R_1. According to the XPath notation, we use (*) to denote the alternation of all elements of Σ, and $(R_1//R_2)$ to denote $(R_1/(*)^*/R_2)$.

Each regular path expression R denotes a set $L(R)$ of strings of symbols from an alphabet Σ. $L(R)$ is defined as follows: $L(\varepsilon) = \{\varepsilon\}$; $L(\phi) = \phi$; $L(a) = a$, for $a \in \Sigma$; $L(R_1|R_2) = L(R_1) \cup L(R_2) = \{\text{string } s \mid s \in L(R_1) \text{ or } s \in L(R_2)\}$; $L(R_1/R_2) = L(R_1)/L(R_2) = \{\text{string } s_1/s_2 \mid s_1 \in L(R_1) \text{ and } s_2 \in L(R_2)\}$; $L(R^*) = \bigcup_{i=0}^{\infty} L(R)^i$, where $L(R)^0 = \{\varepsilon\}$ and $L(R) = L(R)^{i-1} / L(R)$.

Definition: Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be a XML-graph and $p = (v_0, e_1, v_1, \ldots, e_n, v_n)$, where $v_i \in$ N, $0 \le i \le n$, and $e_j \in$ E, $1 \le j \le n$ be a path in X_D. We call the string $\lambda(e_1)/ \lambda(e_2)/ \ldots /\lambda(e_n)$ the *label path* of p, denoted by $\lambda(p)$.

Definition: Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be a XML-graph. A *query path* Q_R on X_D is an expression of the form xRy where x,y are variables and R is a regular path expression over Σ. The *answer set to a query path* $Q_R(X_D)$, denoted $ans(Q_R(X_D))$, is defined as the set of pairs (x,y) s.t. there is a path $p = (x , e_1 , \ldots , e_n , y)$ in X_D and $\lambda(p) \in L(R)$.

2.1 First Approach to Path Indexes

Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be an XML-graph. For each node $v \in N$ let $L_v(X_D)$ be the set of label paths from root v_0 to node v:

$$L_v(X_D) = \{ \lambda(p) \mid p = (v_0 , e_1 , \ldots , e_n , v) \}$$

We define an equivalence relation over nodes in N as follows. Let v,w be nodes in N. We say that $v \equiv w$ iff $L_v(X_D) = L_w(X_D)$. We denote by $[v]$ the equivalence class of v in N.

We call the set of nodes that belongs to the equivalence class $[v]$ the *extent* of $[v]$, denoted $ext([v])$. Then, we construct the index as follows. First, we compute the family of equivalence classes $[v_1]$, $[v_2]$, … , $[v_k]$ that defines a partition of N, and then we attach to each of them the regular expression corresponding to languages $L_{v_1}(X_D)$, … , $L_{v_k}(X_D)$. We also store, for each class $[v]$, its extent $ext([v])$.

Given a query path Q and the index constructed as described above, query evaluation can be performed by simply iterating over all classes $[v_i]$, $1 \le i \le k$, and for each class testing if the corresponding language $L_{v_i}(X_D)$ has a non empty intersection with $L(Q)$. Consequently, $ans(Q)$ will be the union of all extents where that intersection is not empty:

$$ans(\mathrm{Q}) = \{ \bigcup_i ext([v_i]) \mid L_{v_i}(DB) \cap L(Q) \neq \phi \}$$

The index constructed as described above is inefficient because computing the equivalent classes $[v]$ is a PSPACE complete problem [MS99]. In addition, given that the regular languages corresponding to each class $[v_i]$ may overlap, the index size may be much larger than the database.

Two proposals have addressed these problems: dataguides [GW97] and 1-indexes [MS99]. Dataguides consist of a more concise representation of the equivalence classes $[v_i]$ based on a deterministic automaton. 1-indexes, on the other hand, are based in the use of a different equivalence relation to compute the equivalence classes. The resulting index, when viewed as a finite state automaton, is non-deterministic. Next, we will discuss both approaches in more detail. We will use a cyclic data graph corresponding to a generic publications database (*Figure 2.1*).

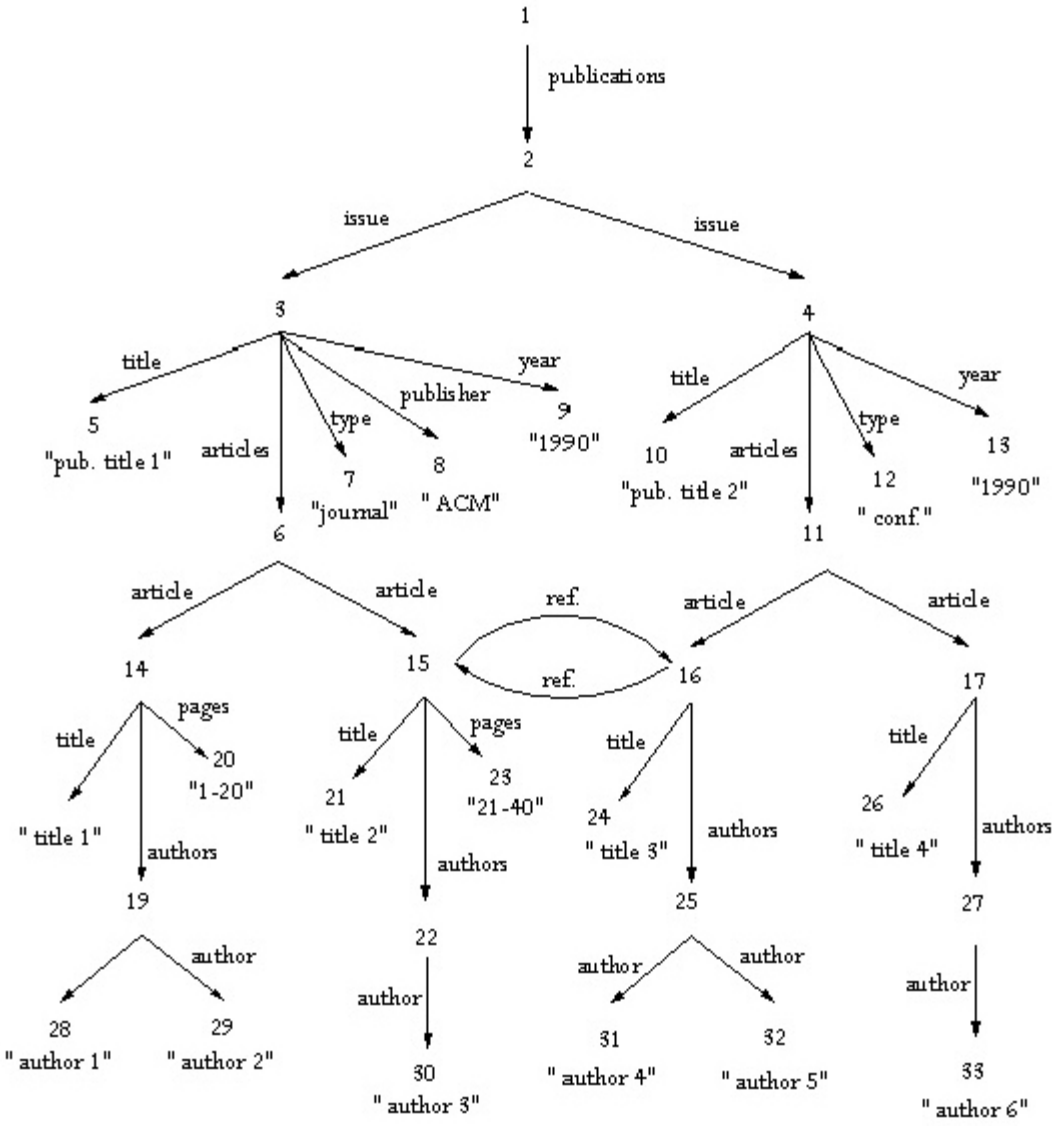


Figure 2.1: A generic publications database.

2.2 Dataguides

Dataguides are a concise and accurate summary of the path structure of a semistructured database. It is accurate because every label path in the data source appears in the data guide and conversely every label path in the data guide appears in the data source. It is concise because the data guide describes every label path of the source exactly once.

Notice that the automaton that describes the language $\bigcup_i L_{v_i}(X_D)$ is a non deterministic finite automaton (NFA). The construction of a dataguide from a data graph is equivalent to the conversion of a NFA into a deterministic finite automaton (DFA) [NUWC97]. This conversion takes linear time when the source is a tree and exponential time (in the worst case) when the source is a graph. It is important to note that a data graph does not necessarily have a unique dataguide in the general case. Since one NFA may have multiple equivalent DFA [HU79], a single data graph may also have several dataguides. In contrast, there is a unique one that is the smallest possible, and it is called *minimal dataguide*. Two possible dataguides for the *Figure 2.1* data graph are shown in *Figure 2.2*.

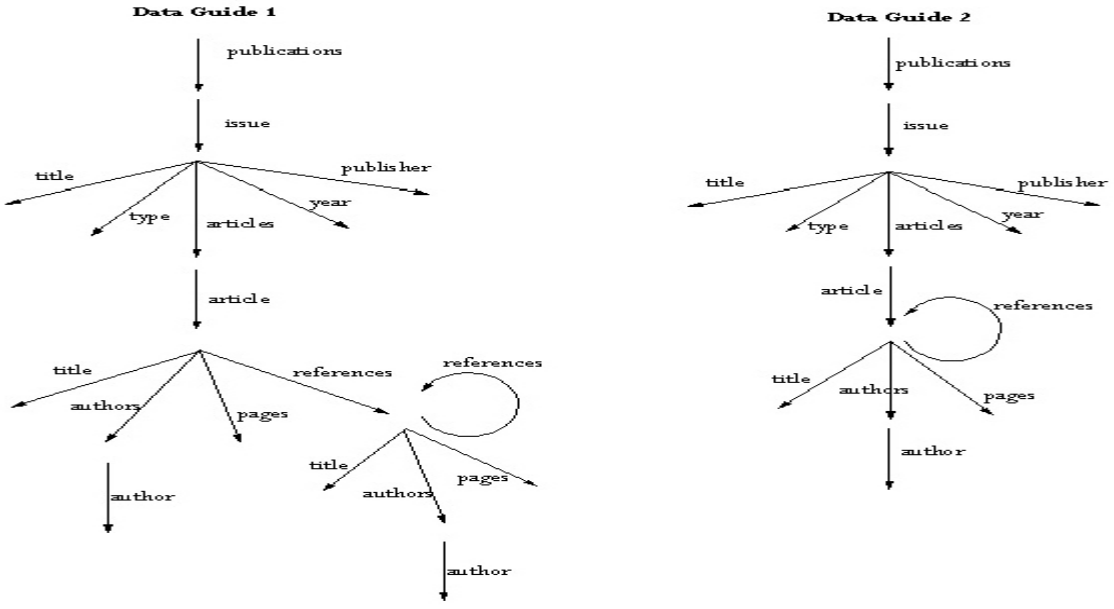


Figure 2.2: Two dataguides corresponding to the publications data graph.

However, not all dataguides are suitable for indexing purposes without additional structures. For instance, the paths publications/issue/articles/article and publications/issue/articles/article/references do not belong to the same equivalence class and yet they reach the same node in the *Dataguide 2* from *Figure 2.2*. This is so because dataguides guarantee that each label path in the data graph reaches one node in the dataguide but do not prevent multiple label paths from reaching the same dataguide node. There is only one type of dataguide which guarantees that all label paths that reach the same node in the dataguide belong to the same equivalence class [*v*], and it is called s*trong dataguide* (*Dataguide 1* in *Figure 2.2*). Since each node represents a different equivalence class, we can add the extents to the nodes to use the strong dataguide as a path index, as shown in *Figure 2.3*. Strong dataguides are the only type of dataguides that can be used for indexing purposes without needing additional structures.

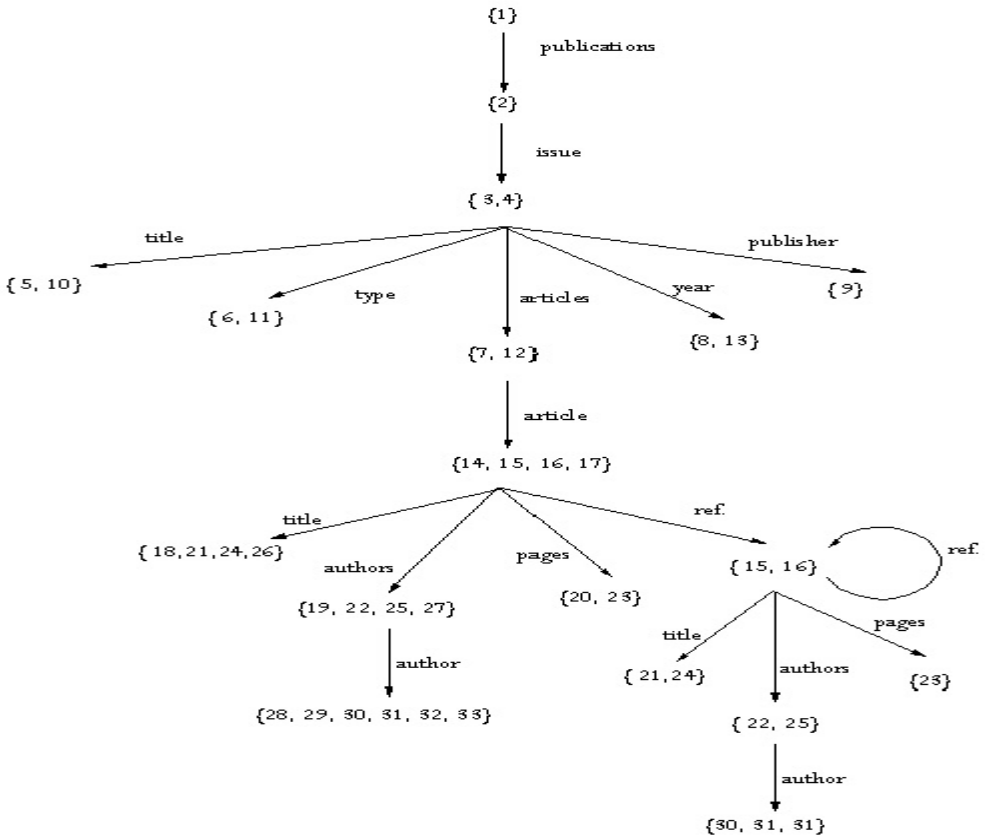


Figure 2.3: Strong dataguide with extents for the publications data graph

We define next strong dataguides more precisely in the context of the XML-graph formalism.

Definition: Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be an XML-graph. A strong dataguide $D_X = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, ext)$ is a rooted graph where N_I is a set of nodes; $x_o \in N_I$ is a distinguished node called root; E_I is a set of edges; ψ_I is an incidence function mapping E_I to $N_I \times N_I$; Σ is the alphabet of X_D; λ_I is a labeling function mapping E_I to Σ; *ext* is a function mapping N_I to 2^N defined as $ext(x)=\{ v \mid \forall p_I = (x_0, \ldots, x), \exists p = (v_0, \ldots, v)$ s.t. $\lambda_I (p_I)=\lambda(p) \}$; and the following two conditions are satisfied:

1. For every pair of paths $p = (v_0, \ldots, v)$, $p' = (v_0, \ldots, v')$ s.t. $\lambda(p) = \lambda(p')$, there is exactly one path $p_I = (x_0, \ldots, x)$ s.t. $\lambda_I (p_I) = \lambda(p) = \lambda(p')$.
2. For every $p_I = (x_0, \ldots, x)$, there is a path $p = (v_0, \ldots, v)$ s.t. $\lambda_I (p_I) = \lambda(p)$.

One of the problems that we face when we construct a dataguide of a deeply nested, cyclic graph is that, in the worst case, we may end up creating a node for every subset of nodes in the data source. As a result, the size of the data guide will grow exponentially in terms of the number of objects in the source. When the source is a tree, however, the size of the dataguide is equal to that of the source in the worst case. A comprehensive study of the theoretical foundations behind dataguides can be found in [NUWC97].

Algorithm 2.1 below constructs a strong dataguide from an XML-graph. It is based in the subset construction algorithm from [ASU86].

Algorithm 2.1: construction of a *strong dataguide* from a XML-*graph*

INPUT:

XML-graph $X_D = (N, v_0, E, A, \psi, \Sigma, \lambda, <)$

OUTPUT:

Strong dataguide $D_X = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, ext)$

METHOD:

$E_I \leftarrow \phi$
$N_I \leftarrow$ new node x_0
$ext(x_0) \leftarrow \{ v_0 \}$
Extents $\leftarrow$ unmarked set $ext(x_0)$
while there is a set $S \in$ *Extents* do
    mark S
    for each symbol $a \in \Sigma$ do
        $T \leftarrow \{ y \mid (x,y) \in ans(Q_a(X_D)),\ x \in S \}$
        if $T \notin$ *Extents* then
            Extents $\leftarrow$ *Extents* + unmarked set T
            $N_I \leftarrow N_I \cup \{$ new node $y \}$
            $ext(y) \leftarrow T$
        else
            $y \leftarrow$ node $y \in N$ s.t. $ext(y) = T$
        fi
        $E_I \leftarrow E_I \cup \{$ new edge $e \}$
        $x \leftarrow$ node $x \in N$ s.t. $ext(x) = S$
        $\psi_I (e) \leftarrow (x,y)$
        $\lambda_I (e) \leftarrow a$
    od
od

2.3 1-index

Like dataguides, 1-indexes are intended to be used by queries that search the database from the root for nodes matching some arbitrary path expression *R*. A 1-index therefore, represents the same set of paths that dataguides do, although using a different approach. The basic idea behind the index construction is the generation of a non-deterministic automaton (*NFA*) to get a more compact structure than dataguides (*DFA*).

To construct the 1-index of a data graph we compute for each node the equivalence class using a bisimulation as equivalence relation. We define bisimulation next.

Definition: Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be a XML-graph. A binary relation ~ on nodes in N is called a *bisimulation* if it satisfies the following. If $v \sim v'$, then for any edge e, $\psi(e) = (u, v)$, $\lambda(e) = a$, there exists an edge e', $\psi(e') = (u', v')$, $\lambda(e') = a$ s.t. $u \sim u'$. Conversely, If $v \sim v'$, then for any edge e', $\psi(e') = (u', v')$, $\lambda(e') = a$, there exists an edge e, $\psi(e) = (u, v)$, $\lambda(e) = a$ s.t. $u \sim u'$.

We say that two nodes v, u are bisimilar (noted $v \approx u$) iff there exists a bisimulation ~ s.t. $v \sim u$.

Using bisimulation we can tackle the index size and the construction cost problems that dataguides yield, obtaining an index structure in which extents are disjoint. As we mentioned before, the type of dataguides used for indexing purposes are the strong dataguides. Since they are constructed over the power set of nodes in the database, the size of the dataguide graph may be as large as exponential in that of the database, while for 1-index it is at most linear. In addition, given that dataguide target sets may overlap (due to the power set construction), the total size of the target sets may be as large as exponential on the size of the database in the worst case, while for 1-index is again at most linear.

We construct the index as follows. Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be an XML-graph. The nodes of the index will be the family of equivalence classes $[v_i]$ computed by bisimulation, which defines a partition of N. The index I has an edge e, $\psi(e) = ([v_i], [v_j])$, $\lambda(e) = a$ iff data graph D contains an edge e', $\psi(e') = (v, v')$, $\lambda(e) = a$ for some $v \in [v_i]$ and $v' \in [v_j]$. We can think of 1-indexes, when seen as finite state automata, as a non-deterministic version of dataguides (*Figure 2.4*).

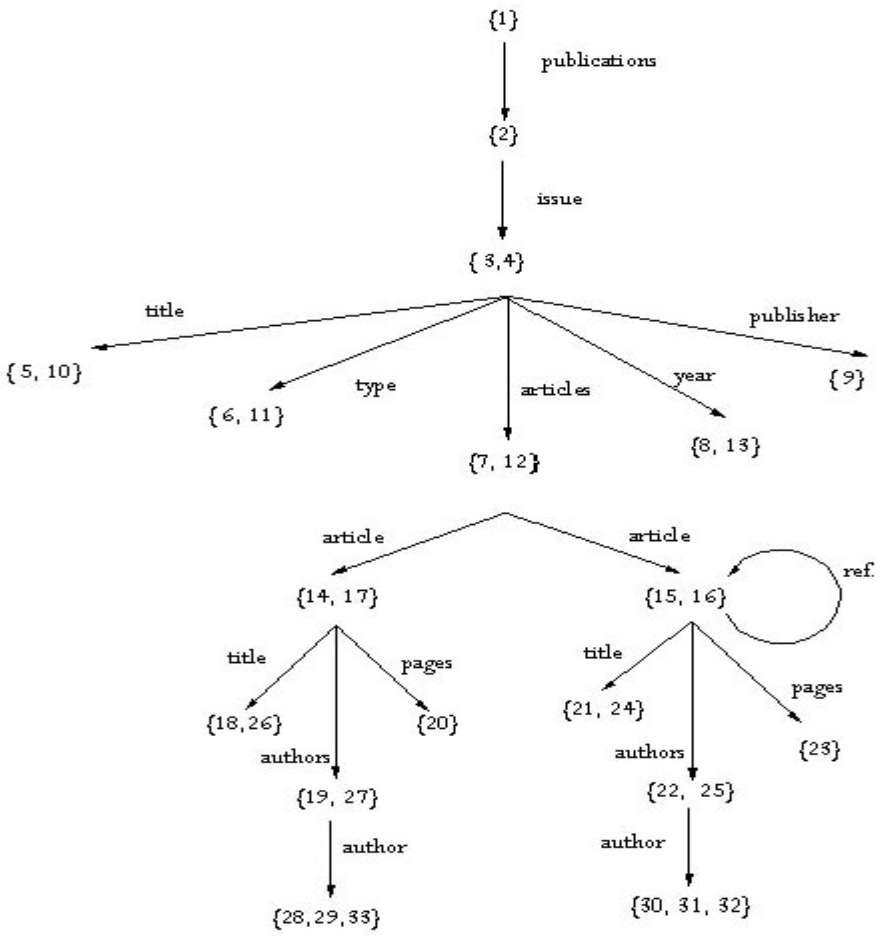


Figure 2.4: A 1-index for the publications data graph.

2.4 2-index

2-indexes are intended to be used by queries that search the database for pairs of nodes matching some arbitrary path expression *R*. Similarly to 1-indexes, we have to define a language equivalence to construct equivalence classes of nodes. We define next a language between pairs of nodes in *N*.

Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be a XML-graph. For each pair of nodes v,w in N let $L_{v,w}(X_D)$ be the set of label paths from root v to node w:

$$L_{v,w}(X_D) = \{ \lambda(p) \mid p = (v , e_1 , \ldots , e_n , w) \}$$

Then, we define an equivalence relation over nodes in N, as follows. Let v,w,v',w' be nodes in N. We say that pairs $(v,w) \equiv (v',w')$ iff $L_{v,w}(DB) = L_{v',w'}(DB)$. As in language equivalence for single nodes, we denote by $[(v,w)]$ the equivalence class of (v,w) in N.

As before, pairs (v,w), (v',w') in N^2 may be distinguished by a query path, iff $(v,w) \notin [(v',w')]$. Using the previous language definition, we construct the index as follows. We compute the family of equivalence classes $[(v_1,w_1)]$, $[(v_2,w_2)]$, … , $[(v_k,w_k)]$ that defines a partition of N^2, and then we attach to each of them the regular expression corresponding to languages $L_{v_1w_1}(X_D)$, … , $L_{v_kw_k}(X_D)$. In addition, we also store the extent for each class, consisting of all pairs (v,w) in the class $[(v,w)]$.

Since computing equivalence pairs of nodes is expensive, we may consider using bisimulation, as we did for 1-indexes.

2.5 Access Support Relations

Access Support Relations [KM90] are general indexing structures for object-oriented databases. They are designed to support functional join along arbitrary reference chains leading from one object instance to another. They also support collection-valued attributes within the attribute chain by materializing frequently traversed reference chains of arbitrary length. Since access support relations are based on the paths found in the schema, it is not possible to use them in a straight-forward manner for the typical schema-less XML

documents. However, we present them here because some of the techniques behind them can be combined with other path index schemes for indexing XML data.

Access Support Relations are a generalization of the binary join indices originally proposed for the relational model [Val87]. One fundamental difference, however, is that rather than relating only two relations (or object types), Access Support Relations materialize access paths of arbitrary length. For comprehensive coverage of the object-oriented data model, we refer the reader to [KKS92].

Definition: A path expression on object o has the form $p = o.\ A_1.\ A_2 \ldots A_n$, where o is an object that contains the attribute A_1, $o.A_1$ refers to an object or set of objects that have an attribute A_2, and so on. Since Access Support Relations are typed, we give next a formal definition of path expressions based on types. For simplicity, we assume that types are not being defined as a subtype of any other type.

Definition: [KM92] let $t_0 \ldots t_n$ be (not necessarily distinct) types. A path expression p on t_0 is an expression $p = t_0.\ A_1.\ A_2 \ldots A_n$ if for each $1 \leq i \leq n$ one of the following two conditions holds:

1. The type t_{i-1} is defined as type t_{i-1} is [..., A_i : t_i, ...], i.e., t_{i-1} is a tuple with an attribute A_i of type t_i.
2. The type t_{i-1} is defined as type t_{i-1} is [..., A_i : t_i', ...] and the type t_i' is defined as type t_i' is { t_i }. i.e., t_i' is a set type containing elements that are instances of t_i. In this case, we say that there is a set occurrence at A_i in the path $p = t_0.\ A_1.\ A_2 \ldots A_n$.

According to this definition, we can represent the SIGMOD Record database as shown in *Figure 2.5*. Each level from top to bottom corresponds to a different type and the rectangular boxes represent the objects.

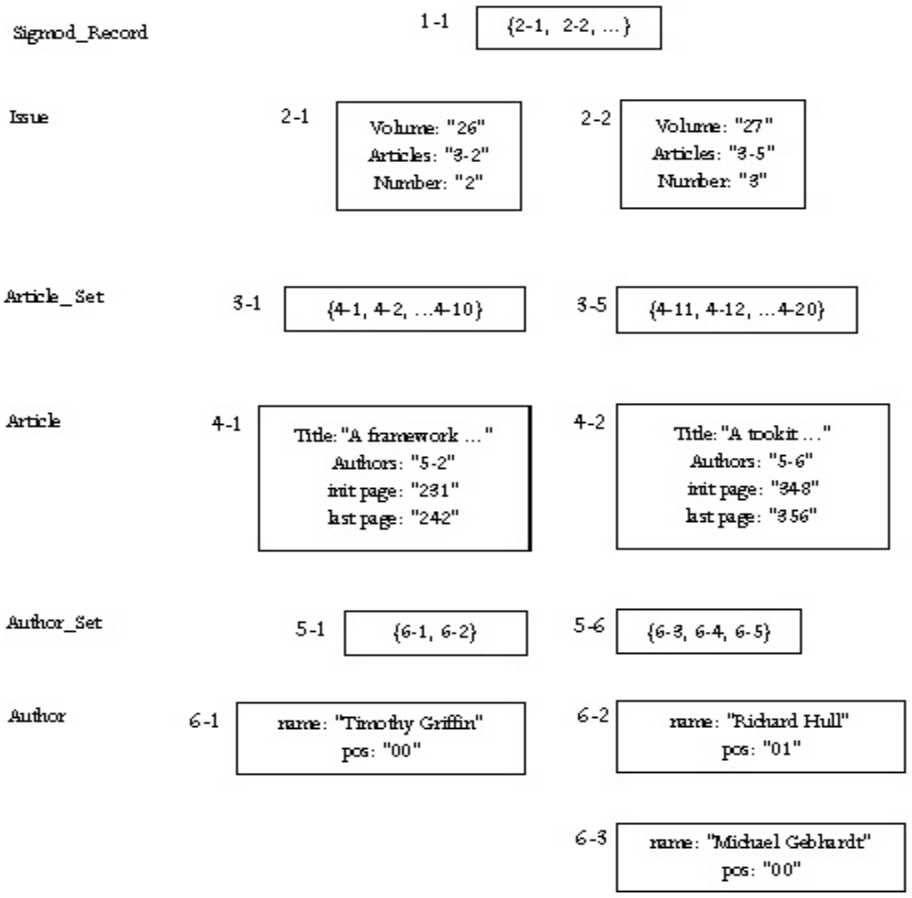


Figure 2.5: object-oriented representation of the SIGMOD Record database

Having defined the concept of path in this object-oriented data model, we may define Access Support Relations as following.

Definition: [KM92]**:** let $t_0 \ldots t_n$ be (not necessarily distinct) types and $p = t_0.A_1.A_2 \ldots A_n$ be a path expression on t_0. Then the *Access Support Relation* $[t_0.A_1.A_2 \ldots A_n]$ is of arity n+1 and its tuples have the form $[S_0, \ldots, S_n]$. The domain of the attributes S_i is the set of OIDs of objects of type t_i, $1 \leq i \leq n$. If t_n is an atomic type then the domain of S_n is t_n, i.e. values are stored directly in the ASR.

Access Support Relations may be maintained in four different extensions, which define the amount of information kept in the index structure. The four possible extensions are the following:

1. The canonical extension, denoted $[t_0.\ A_1.\ A_2 \ldots A_n]_{can}$, which contains only complete paths, i.e. paths from t_0 ending in t_n.
2. The left-complete extension, denoted $[t_0.\ A_1.\ A_2 \ldots A_n]_{left}$, which contains all paths originating in type t_0 but not necessarily ending in type t_n.
3. The right-complete extension, denoted $[t_0.\ A_1.\ A_2 \ldots A_n]_{right}$, which contains paths ending in t_n but possibly originating in some object of type t_j that is not referenced by any object of type t_{j-1} via attribute A_j.
4. The full extension, denoted $[t_0.\ A_1.\ A_2 \ldots A_n]_{full}$, which contains all partial paths.

Figure 2.6 shows a full extension of an Access Support Relation for the path SIGMOD_Record/issue/articles/article/title on the *Figure 2.5* database. It contains all paths corresponding to the indexed path expression.

[Sigmod record / Issue / Articles / Article / Title]

Sigmod record	Issue	Articles	Article	Title
1-1	2-1	3-2	4-1	"A framework"
1-1	2-1	3-2	4-2	"A toolkit ..."
1-1	2-1	3-2	4-3	"... "
...				

Figure 2.6: Access Support Relation (full extension).

For storage, Access Support Relations use an approach similar to binary join indexes [Val87]. Each relation is redundantly stored in two B+-trees: the first keyed on the left-most attribute and the second keyed on the right-most attribute. For the relations of *Figure 2.6*, there is a B+-tree on SIGMOD_Record and another one on title. This storage schema is well suited for traversing paths forward and backward.

2.6 T-indexes

Since 1-indexes and 2-indexes are not tailored to specific paths in the database, when faced with very irregular, cyclic data the index may become too large and inefficient. Restricting the class of queries supported by the index structure can reduce the index complexity and yield better performance. This approach is similar to that of Access Support Relations.

Definition: Let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be an XML-graph. A *generalized query path* Q_R' on X_D is an expression of the form $x_0R_1x_1R_2... R_nx_n$, where x_i, $0 \le i \le n$, are variables and R_j, $1 \le j \le n$ are regular path expression over Σ. The *answer set to a generalized query path* $Q_R'(X_D)$, denoted $ans(Q_R'(X_D))$, is defined as the set of tuples $(x_0, ..., x_n)$ s.t. for all pairs (x_{j-1}, x_j), there is a path $p_j = (x_{j-1} , e_1 , \dots , e_k , x_j)$ in X_D and $\lambda(p_j) \in L(R_j)$.

In order to define T-indexes, the concept of path template was introduced [MS99]. A path template *t* has the form $v_0T_1x_1... T_nx_n$ where x_i, $1 \le i \le n$ are variables and each T_i, $1 \le i \le n$, is either a regular path expression or a generic place holder **P** that represents any path. By instantiating each of the place holders **P** with a concrete path expression we get a concrete generalized query path. For instance, the path template v_0 publications x_1 **P** x_2 title x_3 has several possible instantiations, two of them are:

$i_1 = v_0$ publications x_1 // x_2 title x_3

$i_2 = v_0$ publications x_1 //article x_2 title x_3

Once defined a path template *t*, a T-index is constructed to support path queries that are instances of *t*. As in Access Support Relations, templates are used to guide the indexing mechanism on the more frequently queried part of the database. Special cases of T-indexes are 1-indexes and 2-indexes. The template that corresponds to a 1-index is $t_1 = v_0\mathbf{P}x_1$ and the one that corresponds to a 2-index is $t_2 = v_0$ // x_1 **P** x_2.

For the construction of the index, we proceed in a manner similar to 1 and 2-indexes. We first define the language equivalence to be the equivalence relation on nodes, this time a tuple (x_1,... ,x_n). Then, we compute the equivalence classes defined by the language equivalence. For a comprehensive discussion of T-indexes, we refer the reader to [MS99].

Chapter 3
ToXin

3.1 Overview

In order to speed up query processing for regular path queries we have developed an index scheme we call *ToXin*. Our original goal when designing ToXin was to have an index that supports navigation of the XML graph both backward and forward to answer any regular path query. At the same time, we wanted to keep the size of the index schema linear (in the worst case) w.r.t. the size of the XML graph. In addition, we needed data structures to help locate nodes that not only satisfy regular path expressions but also predicates over values. Previous index schemes developed for OODB and semistructured data satisfy only some of these requirements. We have described some of them in previous chapters. *Strong dataguides* store only paths from the root and do not help in backward navigation. They also have the additional problem of exponential growth when indexing deeply nested, cyclic data graphs. *Access Support Relations* and *T-indexes*, on the other hand, store only a predefined subset of paths and, therefore, support only a limited class of path queries. ToXin borrows some ideas from these techniques and extends them in several ways.

ToXin consists of two different types of index structures: the *value index* and the *path index*. The path index has two components: the *index tree*, which is a minimal dataguide, and a set of *instance functions*, one for each edge in the index tree. Each instance function keeps track of the parent-child relationship between the pair of nodes that defines each XML element. Since the instance functions play the role of the extents, we can relax the condition requiring that all label paths that reach the same node in the dataguide belong to the same equivalence class. That is the reason why we are able to use a minimal dataguide instead of a strong dataguide.

The storage structure of the instance functions is similar to that of Access Support Relations (see Section 2.5). Each instance function is stored in two redundant hash tables: a *forward instance table* for forward navigation and a *backward instance table* for backward navigation. The value index, on the other hand, consists of a set of value relations that store the XML nodes and values corresponding to an index edge. For each edge in the index schema that corresponds to a set of XML nodes containing values, there is a value relation. Each relation is implemented as a B+-trees keyed on the values, which are always strings. Value and path indexes combined can be used to answer regular path queries with predicates over values such as those expressed using *XPath*.

We present below a definition of the path index. This definition does not interpret IDRef attributes as links, hence the resulting index schema has a tree structure.

Definition: let $X_D = (N, v_o, E, A, \psi, \Sigma, \lambda, <)$ be an XML-graph. A *path index tree* $I_X = (N_I , x_0 , E_I , \psi_I , \Sigma, \lambda_I , X_D, \sigma)$ is a tree where N_I is a set of nodes; $x_o \in N_I$ is a distinguished node called root; E_I is a set of edges; ψ_I is an incidence function mapping E_I to $N_I \times N_I$; Σ is the alphabet of X_D; λ_I is a labeling function mapping E_I to Σ; σ is an instance function mapping $E_I \times N$ to 2^N defined as $\sigma(e_I , v) = \{ w \mid \exists p = (v_0 , \ldots , v, e, w)\ \exists p_I = (x_0 , \ldots , x, e_I , y)$, and $\lambda_I (p_I)=\lambda(p) \}$; and the following two conditions are satisfied:

1. For every pair of paths $p = (v_0, \ldots, v)$, $p' = (v_0, \ldots, v')$ s.t. $\lambda(p) = \lambda(p')$, there is exactly one path $p_I = (x_0, \ldots, x)$ s.t. $\lambda_I(p_I) = \lambda(p) = \lambda(p')$.
2. For every $p_I = (x_0, \ldots, x)$, there is a path $p = (v_0, \ldots, v)$ s.t. $\lambda_I(p_I) = \lambda(p)$.

We will present next an algorithm for computing the ToXin tree from an XML-graph. It performs a depth-first traversal of the tree defined by the natural element nesting in the XML-graph. For each element visited it first checks whether the corresponding edge has already been added to the index and adds it if it was not. Then, it defines the instance function σ for the edge and the current element. Since the algorithm traverses the underlying tree defined by the element nesting, it terminates after traversing all edges exactly once. Note that it does not follow the edges defined by the attributes.

Algorithm 3.1: construction of a *ToXin tree* from a XML-*graph*

INPUT:

XML-graph $X_D = (N, v_0, E, A, \psi, \Sigma, \lambda, <)$

OUTPUT:

ToXin Tree $I_X = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, \sigma)$

METHOD:

$E_I \leftarrow \phi$
$N_I \leftarrow$ new node x_0
Index(x_0, v_0)

procedure *Index*(x, v) // $x \in N_I$, $v \in N$
  for each edge $e \in (E\text{-}A)$ s.t. $\psi(e) = (v, w)$ do
    if there is no edge $e_I \in E_I$ s.t. $\psi_I(e_I) = (x, y)$ and $\lambda_I(e_I) = \lambda(e)$ then
      $E_I \leftarrow E_I \cup \{$ new edge $e_I \}$
      $N_I \leftarrow N_I \cup \{$ new node $y \}$
      $\lambda_I(e_I) \leftarrow \lambda(e)$
      $\psi_I(e_I) \leftarrow (x, y)$
    fi
    $\sigma(e_I, v) \leftarrow w$
    Index(y,w)
  od
end

Let us see with an example how ToXin works. *Figures 3.1, 3.2* and *3.2* show the ToXin tree and the tables for the XML document from *Figure 2.1*. The VT boxes in *Figure 3.1* represent the value tables and the IT boxes the instance tables. In contrast to dataguides and 1-indexes, which index only paths which start from the root, all paths in the database are represented in ToXin. For instance, not only do we find the paths that match x/publications/issue/y (represented by the reference chains in tables *publications* and *publications/issue*) but also those that match x/issue/y (represented by the table *publications/issue*). This way we can use ToXin for both forward and backward navigation starting from any node in the index.

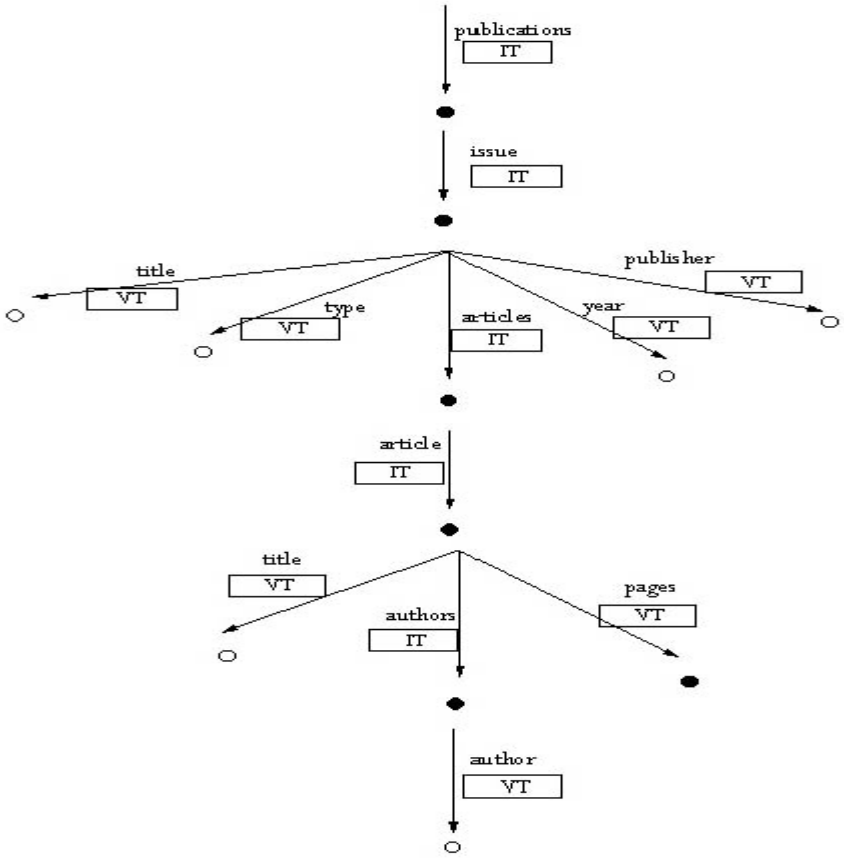
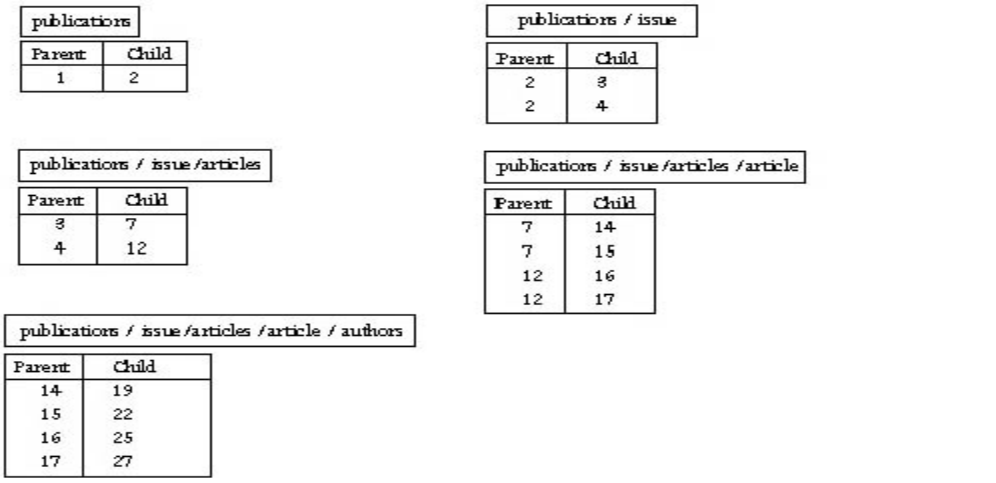


Figure 3.1: ToXin tree for the publication data graph.



publications

Parent	Child
1	2

publications / issue

Parent	Child
2	3
2	4

publications / issue /articles

Parent	Child
3	7
4	12

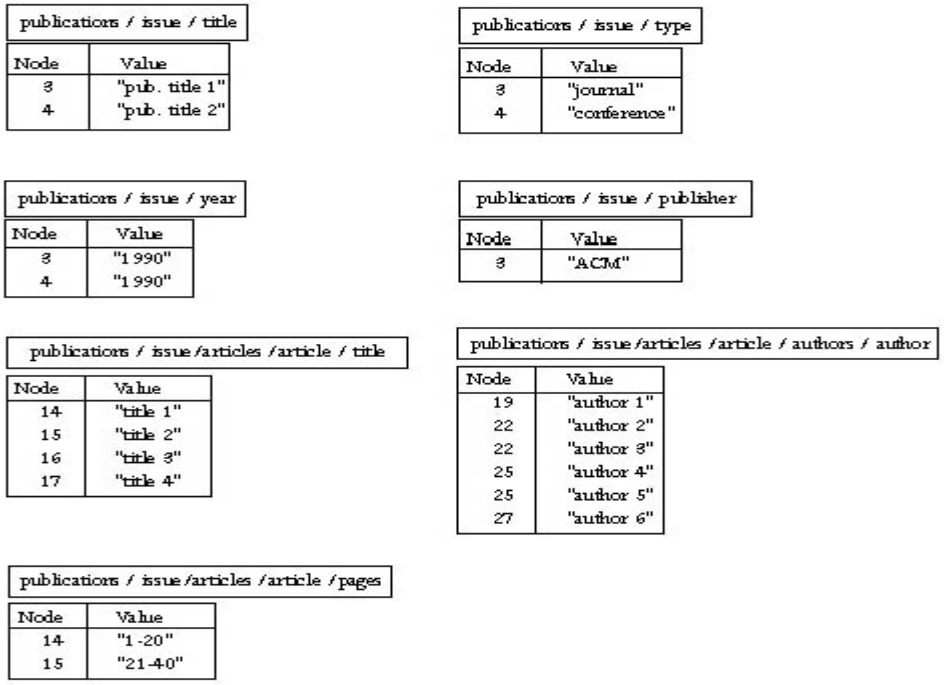
publications / issue /articles /article

Parent	Child
7	14
7	15
12	16
12	17

publications / issue /articles /article / authors

Parent	Child
14	19
15	22
16	25
17	27

Figure 3.2: ToXin instance tables for Figure 3.1 index tree.



publications / issue / title

Node	Value
3	"pub. title 1"
4	"pub. title 2"

publications / issue / type

Node	Value
3	"journal"
4	"conference"

publications / issue / year

Node	Value
3	"1990"
4	"1990"

publications / issue / publisher

Node	Value
3	"ACM"

publications / issue /articles /article / title

Node	Value
14	"title 1"
15	"title 2"
16	"title 3"
17	"title 4"

publications / issue /articles /article / authors / author

Node	Value
19	"author 1"
22	"author 2"
22	"author 3"
25	"author 4"
25	"author 5"
27	"author 6"

publications / issue /articles /article /pages

Node	Value
14	"1-20"
15	"21-40"

Figure 3.3: ToXin value tables for Figure 3.1 index tree.

3.2 Implementation

The Document Object Model

In the previous sections we explained how to construct a path index from an XML-graph. Next, we show how to process an XML document in order to obtain the XML-graph.

To process an XML document we need to use APIs and our choice was the document object model (DOM). DOM is a tree structure-based API from a W3C Recommendation [W3C98]. An XML document is represented in DOM as a tree structure whose nodes are elements. The DOM tree corresponding to the data graph of *Figure 2.1* is depicted in *Figure 3.4*. DOM provides a set of APIs to access and manipulate the nodes in a DOM tree.

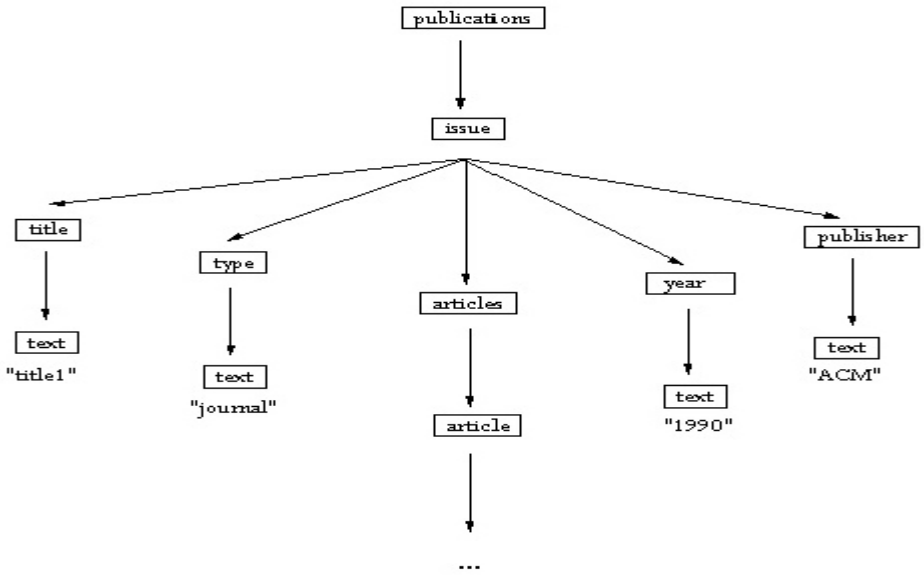


Figure 3.4: DOM tree for the publications data graph

We implemented the prototype of ToXin in Java and we use the IBM parser XML4J to create the DOM tree from the XML document. We based our implementations on the DOM tree generated by the XML4J parser. Since our model requires an edge-labeled graph, first we compute the XML-graph from the DOM tree and then we build the index according to *Algorithm 3.1*.

For each XML element containing values, DOM creates an extra node called text. Therefore, the DOM tree contains more nodes than the number of XML elements in the documents. As a consequence, the XML-graph presents additional text edges as well. If indexed as normal edges, this extra layer of text edges induces a significant overhead for both building and querying the index tree and the XML-graph. Figure 3.5 shows the ToXin tree for XML-graphs presenting text edges. The value tables and instance tables for Figure 3.5 are shown in Appendix 1.

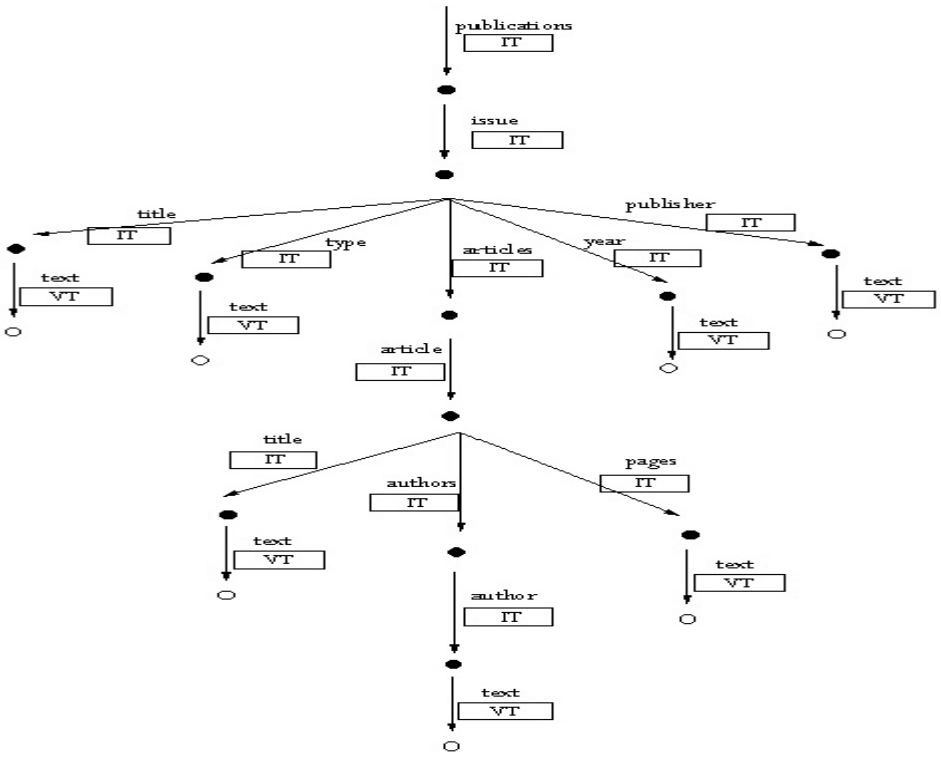


Figure 3.5: Index tree for the publications data graph (with text edges)

Our first implementation of ToXin regards text edges in the XML-graph as normal edges and computes the index tree accordingly. For the second implementation, we eliminate the text edges from the XML-graph and move the nodes at the end of such edges one level up using the following algorithm:

```
for each edge e∈ E, ψ(e) = (v, w), s.t. λ(e) = text  do
        for each edge e'∈ E, s.t. ψ(e') = (u, v) do
                ψ(e') = (u, w)
                remove edge e from E
                remove node v from N
        od
od
```

Applying this procedure, we obtain the more compact and natural structure of *Figure 3.1*. It is easy to see that the XML-graph is always larger in the first implementation than in the second one and, consequently, so is the index graph.

3.2 Navigation Language

For testing ToXin performance, we needed a language that supports path expressions and predicates over values. This functionality is provided by a number of query languages for XML, such as XML-QL [DFF+99], XSLT [W3C99b], XQL [RLS98] and XPath. In addition, most of these languages also provide complex mechanisms for restructuring and updating documents, functionality we did not require for our experiments. Our final choice was XPath because it is simple to implement and yet powerful enough for testing ToXin capabilities in speeding up queries involving tree navigation and value selection. Nevertheless, since the

current ToXin implementation does not support order, we did not implement the XPath functionality for supporting queries involving order.

The navigation language we designed consists of a set of navigation and filtering functions that support tree traversal, both forward and backward, and selection of nodes satisfying a certain value. Functions *navigateDown* and *navigateUp* accept a set of nodes *N* and a regular path expression *R* as input and return the set of nodes that are reachable by paths matching *R*. The only difference between them is the direction in the evaluation of the regular path expression *R*. The function *selectNodes*, on the other hand, accepts a set of nodes *N* and a value *v* as input and returns the set of nodes whose value is exactly *v*. For instance, the following query:

Q3.1 = publications/issue[year = "*1990*"]/articles/article/title

may be expressed in our navigation language as a composition of four subqueries as follows:

Q3.2.1 = navigateDown(v0, "*publications/issue/year*")
Q3.2.2 = selectNodes(Q3.2.1, "1990")
Q3.2.3 = navigateUp(Q3.2.2, "year")
Q3.2.4 = navigateDown(Q3.2.3, "*articles/article/title*")

The ToXin index scheme was designed to speed-up all query processing stages. According to our query decomposition strategy mentioned above we define three stages in the query evaluation. The first stage (called *pre-selection stage*) comprises the first navigation down the tree for the pre-filtering section. The second stage (called *selection stage*) consists of the value selection performed by the filter section. Finally, the third stage (called *post-selection stage*) spans the navigation up after the selection stage and the last navigation down. We classified ToXin along with other path indexes scheme explained in

Chapter 2 with regards to the query evaluation stages in which they can be used. The classification is shown in *Table 3.8*.

	Scheme name	pre-selection stage	selection stage	post-selection stage
General Path Indexes	Dataguides	Yes	-	-
	1-index	Yes	-	-
	2-index	Yes	-	-
	Dataguides + Value Index (Lore)	Yes	Yes	-
	ToXin	Yes	Yes	Yes
Path-specific Indexes	ASR	Yes	Yes	Yes
	T-index	Yes	-	-

Table 3.8: Classification of the indexing schemes described in Chapters 2 and 3.

Query evaluation

Let $I_X = (N_I , x_0 , E_I , \psi_I , \Sigma, \lambda_I , X_D, \sigma)$ be a ToXin tree and let $Q_R=xRy$ be a query path. We have two evaluation strategies depending on whether the variable x is instantiated with the index root x_0 or not. We describe first the evaluation strategy for the general case in which x is not instantiated with the root. Starting from node x, we follow the reference chains over the instance tables that belong to the edges in the paths matching R. The answer $ans(Q_R)$ contains the set of nodes that are at the end of those reference chains. The procedure is the same for both forward and backward navigation; the only difference is the instance tables used for computing the reference chains (forward or backward instance tables respectively).

Let us consider now the special case in which x is instantiated with the index root x_0. In this case, we can avoid following the reference chains over the instance tables. Since there

is no previous filtering involved and we are computing over an index tree, we can simplify the computation as follows. First, we follow the paths over the index schema that match the regular path expression *R*. Then we select the last instance tables of each path and we compute the union of the child column of the selected instance tables. This special case of the evaluation strategy is closely related to query processing over dataguides, where the child column of the instance tables are the equivalent to the extents of dataguides nodes.

In order to present the algorithm that computes the navigateDown and navigateUp functions according to the evaluation strategies described above, we need to define first a formalism called *query graphs*. Query graphs are transition graphs constructed from the NFA corresponding to a regular path expression. By using Thompson's construction [ASU86] we compute first the NFA corresponding to a regular path expression and then we construct the transition graph associated with it.

Definition: let $P = (S, \Sigma, \delta, s_o, F)$ be an NFA corresponding to a regular path expression. The *query graph* associated with *P* is a directed graph $Q = (S, s_0, E_Q, \psi_Q, \Sigma, \lambda_Q, F)$ where *S* is a set of states; $s_0 \in S$ is a distinguished node called initial state; E_Q is a set of edges; ψ_Q is an incidence function that maps E_Q to $S \times S$; Σ is the alphabet of *Q*; λ_Q is a labeling function mapping E_Q to Σ; and $F \in S$ is a set of final states. If state $t \in \delta(s, a)$ for $s, t \in S$ and $a \in \Sigma$, then there is and edge $e \in E_Q$ with $\psi_Q(e) = (s, t)$ and $\lambda_Q(e) = a$.

Algorithm 3.2 implements navigateDown for the general case in which the input node *x* is not the root. The algorithm is related to that in WebOQL for computing the navigation of a tree in a web [Aro97]. *Algorithm 3.3*, on the other hand, implements navigateDown for the case in which the navigation starts from the root x_0.

Algorithm 3.2 (navigateDown v1.0): computation of a forward navigation over a ToXin tree.

INPUT:

ToXin tree $I_X = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, \sigma)$, where $X_D = (N, v_0, E, \psi, \Sigma, \lambda, <)$

query graph $Q = (S, s_0, E_Q, \psi_Q, \Sigma, \lambda_Q, F)$

x_i, v_i

OUTPUT:

ans(Q(X_D))

METHOD:

Result $\leftarrow \phi$
Added $\leftarrow \phi$
Visited $\leftarrow \phi$
Selected $\leftarrow v_i$
SearchDown(x_i , s_i , *Selected*)

procedure *SearchDown*(*x, s, Selected*)
Visited $\leftarrow$ *Visited* $\cup$ {(*v, s*)}
for each edge $e \in E_I$ s.t. $\psi_I(e) = (x, y)$ do
for each edge $e_Q \in E_Q$ s.t. $\psi_Q(e_Q) = (s, t)$ and $\lambda_Q(e_Q) = \lambda_I(e)$ do
if $t \in F$ and $e \notin$ *Added* then
Added $\leftarrow$ *Added* $\cup$ {*e*}
Result $\leftarrow$ *Result* $\cup$ *Follow*(*e*, *Selected*)
fi
if $(y, t) \notin$ *Visited* then
Selected $\leftarrow$ *Follow*(*e*, *Selected*)
SearchDown (*y, t, Selected*)
fi
od
od
end

procedure *Follow*(*e*, *X*)
$Y \leftarrow \phi$
for each $x \in X$ do
$Y \leftarrow Y \cup \sigma(e, x)$
od
Follow $\leftarrow Y$
end

Algorithm 3.3 (navigateDown v2.0): computation of a forward navigation from the root over a ToXin tree.

INPUT:

ToXin tree $I_X = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, \sigma)$, where $X_D = (N, v_0, E, \psi, \Sigma, \lambda, <)$

query-graph $Q = (S, s_0, E_Q, \psi_Q, \lambda_Q, F)$

OUTPUT:

ans(Q(X_D))

METHOD:

Result $\leftarrow \phi$
Added $\leftarrow \phi$
Visited $\leftarrow \phi$
Selected $\leftarrow v_0$
SearchDown(x_0 , s_0)

procedure *SearchDown*(*x, s*)
Visited $\leftarrow$ *Visited* $\cup$ {(*x, s*)}
for each edge $e \in E_I$ s.t. $\psi_I(e) = (x, y)$ do
for each edge $e_Q \in E_Q$ s.t. $\psi_Q(e_Q) = (s, t)$ and $\lambda_Q(e_Q) = \lambda_I(e)$ do
if $t \in F$ and $e \notin$ *Added* then
Added $\leftarrow$ *Added* $\cup$ {*e*}
Result $\leftarrow$ *Result* $\cup$ *Child*(*e*)
fi
if $(y, t) \notin$ *Visited* then
SearchDown (*y, t*)
fi
od
od
end

procedure *Child*(*e*)
$Y \leftarrow \phi$
for each $x \in N$ s.t. $\sigma(e, x)$ is defined do
$Y \leftarrow Y \cup \sigma(e, x)$
od
Child $\leftarrow Y$
end

In order to perform the comparative experiments we have also implemented navigateDown for the XML-graph and Strong dataguides. The algorithms are presented in Appendix 2.

Chapter 4

Experiments

4.1 Experimental Setup

We implemented the ToXin prototype in Java 2 and we used the IBM parser XML4J to create the DOM tree from the XML document. All the experiments presented herein were conducted in a Sun SPARCstation running Solaris 2.5. Both Index and DOM were kept in memory in the experiments.

In order to study the tradeoffs of a set of indexing schemes we carried out a series of comparative performance experiments. We use query processing time as performance metric. The experiments evaluate the effect of several parameters on the performance of the query evaluation using ToXin, dataguides, dataguides + value index, and the XML source itself. The parameters can be classified into *data source-specific* and *query-specific*. The data source-specific parameters are document size, number of XML nodes and values, path complexity (degree of nesting), and average value size (short or long strings). Those query-specific are selectiveness of the path constraints (queries expressed with or without // and * operators), size of the query answer (small or large), and number of elements selected in the filter section (small or large).

Our benchmark consists of four data sources: the conference papers from the DBLP database [Ley00], a sample of movies from the Internet Movies Database [IMDB00], the four religious texts from [Bos98] and twenty Shakespeare plays from [Bos99]. The choice of the document samples was aimed at determining the impact of nesting and average value size on the index performance. DBLP is a classical example of a bibliographical database containing deeply nested data. IMDB presents a flat structure typical of a straight-forward mapping from relational data. The values contained in both the DBLP and IMDB documents are short strings. The religious texts and the Shakespeare works, on the other hand, are exponents of text databases: the former containing in average longer string values than the latter and yet similar degree of nesting. The path structures of the tested documents are shown in Appendix 4.

We use DBLP and IMDB to explore the impact of nesting. To study the effects of different value sizes, we use the religion texts and the Shakespeare works. In addition, to test the index performance with different document sizes and similar path structures, we created two XML documents for each data source, one larger than the other one. *Table 4.1* shows some parameters of the benchmark.

	DBLP		IMDB		Shakespeare		Religion	
	Doc 1	**Doc 2**	**Doc 1**	**Doc 2**	**Doc 1**	**Doc 2**	**Doc 1**	**Doc 2**
File size (Mb)	1.8	8.9	0.8	3.9	1.1	4.4	1.0	7.0
Index generation time (sec)	17.2	80.2	5.5	41.5	4.1	25.7	1.3	9.3
# of XML nodes	90040	405103	57854	293183	45776	181438	16810	95594
# of XML values	44027	190232	27084	137296	20437	81779	8283	46334
# of index nodes (ToXin 1)	47	73	14	14	52	62	25	78
# of index nodes (ToXin 2)	27	40	8	8	31	37	17	49
Avg. Value Size	Short	Short	Short	Short	Long	Long	Long	Long
Degree of Nesting	High	High	Low	Low	High	High	High	High

Table 4.1: Parameters of the benchmark

The queries we used for our experiments test the performance of simple selection of nodes by value and tree navigation. We tested each query in two versions: one with very selective path constraints so that the portion of the index scanned is smaller, and the other with more relaxed constraints with a resulting extensive index navigation. We present next the queries grouped by document sample:

DBLP

q_1'= //[year = “1998”]//title

q_1 = /dblp/conference/issues/issue/inproceedings[year = “1998”]/title

q_2' = //conference[title = “VLDB”]/*//title

q_2 = /dblp/conference[title = “VLDB”]/issues/issue/inproceedings/title

q_3' = //[author = “Serge Abiteboul”]//title

q_3 = /dblp/conference/issues/issue/inproceedings[author = “Serge Abiteboul”]/title

IMDB

q_4’ = //[genre = “Drama”]//title

q_4 = /movies/movie[genre = “Drama”]/title

q_5’ = //[title = “Bolero”]//year

q_5 = /movies/movie[title = “Bolero”]/year

q_6’ = //[year = “1950”]//title

q_6 = /movies/movie[year = “1950”]/title

Shakespeare Works

q_7’ = //[speaker = “Mark Anthony”]//line

q_7 = /shakespeare/play/act/*/speech[speaker = “Mark Anthony”]/line

q_8’ = //[title = “The Tragedy of Anthony and Cleopatra”]/act//line

q_8 = /shakespeare/play[title = “The Tragedy of Anthony and Cleopatra”]/act/*/speech/line

q_9’ = //[title = “The Tragedy of Anthony and Cleopatra”]//persona

q_9 = /shakespeare/play[title = “The Tragedy of Anthony and Cleopatra”]/personae/persona

Religious Texts

q_{10}’ = //[bktshort = “Matthew”]//v

q_{10} = /religion/book/bookcoll/book/[bktshort = “Matthew”]/chapter/v

q_{11}’ = /[title = “The New Testament”]//bktlong

q_{11} = /religion/book[titlepg/title = “The New Testament”]/bktcoll/book/bktlong

To compare query performance across different documents we classify the queries with regards to the selectiveness of its path constraints and the sizes of the query answer and node selection in the filter section. The following table shows the classification according with these criteria:

Query Type	Query	Characteristics
LL	q1, q4, q7	Large query answer Large filter selection
LL*	q1', q4', q7'	Large query answer Large filter selection Relaxed path constraints
LS	q2, q8, q10	Large query answer Small filter selection
LS*	q2', q8', q10'	Large query answer Small filter selection Relaxed path constraints
SS	q3, q5, q6, q9, q11	Small query answer Small filter selection
SS*	q3', q5', q6', q9', q11'	Small query answer Small filter selection Relaxed path constraints

Table 4.2: Query classification

4.2 Experimental Results

Figures 4.3, 4.4, 4.5, and *4.6* show the execution time for each stage of every query type of *Figure 4.2*. The rows represent the performance of each query evaluation stage over the XML-graph (XML), the first ToXin implementation with text edges (ToXin 1) and the second implementation without text edges (ToXin 2). The first three columns of each table correspond to the small documents and the second three columns to the large ones. This way we can appreciate the impact of using a path index in each query processing stage.

DBLP		1			2		
		XML	ToXin 1	ToXin 2	XML	ToXin 1	ToXin 2
LL*	Pre	49.86	0.31	0.18	193.12	0.46	0.26
	Sel	1.18	0.03	0.03	13.28	0.03	0.03
	Post	0.84	0.08	0.05	8.33	0.66	0.35
	Total	51.88	0.41	0.25	214.73	1.15	0.64
LL	Pre	22.34	0.10	0.08	116.64	0.09	0.07
	Sel	1.09	0.03	0.03	12.52	0.03	0.03
	Post	0.77	0.09	0.05	8.36	0.67	0.40
	Total	24.20	0.21	0.16	137.53	0.79	0.50
LS*	Pre	52.21	0.20	0.14	197.99	0.39	0.31
	Sel	0.01	0.02	0.02	0.10	0.04	0.04
	Post	36.71	1.29	0.84	36.40	0.64	0.47
	Total	88.92	1.51	1.00	234.50	1.06	0.82
LS	Pre	0.02	0.04	0.02	0.11	0.03	0.02
	Sel	0.00	0.02	0.02	0.05	0.04	0.04
	Post	20.49	0.64	0.51	20.13	0.34	0.29
	Total	20.51	0.70	0.55	20.30	0.40	0.34
SS*	Pre	39.06	0.24	0.13	193.00	0.55	0.30
	Sel	1.86	0.07	0.07	20.42	0.09	0.09
	Post	0.14	0.03	0.02	0.17	0.01	0.01
	Total	41.06	0.34	0.21	213.59	0.65	0.40
SS	Pre	23.85	0.08	0.07	107.78	0.08	0.07
	Sel	1.99	0.07	0.07	20.19	0.09	0.09
	Post	0.14	0.01	0.01	0.16	0.02	0.01
	Total	25.98	0.16	0.15	128.13	0.19	0.17

Table 4.3: Performance results of DBLP queries

IMDB		1			2		
		XML	ToXin 1	ToXin 2	XML	ToXin 1	ToXin 2
LL*	Pre	21.95	0.09	0.05	114.28	0.06	0.03
	Sel	1.57	0.04	0.04	8.71	0.04	0.04
	Post	2.74	0.52	0.29	15.56	0.84	0.49
	Total	26.26	0.65	0.38	138.56	0.94	0.56
LL	Pre	11.64	0.03	0.02	69.85	0.03	0.02
	Sel	1.58	0.04	0.04	8.66	0.04	0.04
	Post	2.66	0.51	0.29	15.86	0.05	0.05
	Total	15.87	0.58	0.35	94.37	0.12	0.11
LS*	Pre	0.00			0.00		
	Sel	0.00			0.00		
	Post	0.00			0.00		
	Total	0.00	0.00	0.00	0.00	0.00	0.00
LS	Pre	0.00			0.00		
	Sel	0.00		0.00	0.00		
	Post	0.00			0.00		
	Total	0.00	0.00	0.00	0.00	0.00	0.00
SS*	Pre	21.48	0.12	0.07	102.88	0.14	0.08
	Sel	1.05	0.06	0.06	6.17	0.14	0.14
	Post	0.01	0.00	0.00	0.01	0.01	0.00
	Total	22.53	0.18	0.13	109.06	0.28	0.22
SS	Pre	12.42	0.03	0.03	61.10	0.04	0.03
	Sel	1.05	0.06	0.06	6.11	0.14	0.14
	Post	0.01	0.00	0.00	0.01	0.01	0.00
	Total	13.48	0.09	0.09	67.22	0.18	0.18
SS*	Pre	23.63	0.13	0.07	107.33	0.08	0.04
	Sel	1.33	0.04	0.04	6.39	0.06	0.06
	Post	0.15	0.04	0.02	0.58	0.08	0.05
	Total	25.11	0.21	0.13	114.29	0.22	0.15
SS	Pre	13.31	0.03	0.02	63.78	0.04	0.03
	Sel	1.33	0.04	0.04	6.40	0.06	0.06
	Post	0.14	0.04	0.02	0.58	0.08	0.05
	Total	14.78	0.11	0.08	70.75	0.18	0.14

Table 4.4: Performance results of IMDB queries

Shakespeare		1			2		
		XML	ToXin 1	ToXin 2	XML	ToXin 1	ToXin 2
LL*	Pre	23.41	0.42	0.27	97.34	0.43	0.26
	Sel	2.78	0.05	0.05	9.73	0.05	0.05
	Post	0.60	0.14	0.12	0.75	0.12	0.09
	Total	26.79	0.61	0.44	107.82	0.60	0.40
LL	Pre	16.88	0.08	0.08	59.09	0.10	0.09
	Sel	2.42	0.05	0.05	8.45	0.05	0.05
	Post	0.52	0.15	0.09	2.08	0.09	0.07
	Total	19.82	0.28	0.22	69.63	0.24	0.22
LS*	Pre	23.63	0.41	0.24	96.10	0.40	0.27
	Sel	0.12	0.02	0.02	0.40	0.03	0.03
	Post	5.89	0.73	0.55	7.54	0.62	0.43
	Total	29.64	1.16	0.81	104.05	1.05	0.73
LS	Pre	0.02	0.09	0.07	0.04	0.09	0.08
	Sel	0.01	0.02	0.02	0.01	0.03	0.03
	Post	5.11	0.57	0.48	5.11	0.49	0.39
	Total	5.13	0.67	0.56	5.16	0.61	0.50
SS*	Pre	23.13	0.35	0.22	94.95	0.36	0.22
	Sel	0.12	0.02	0.02	0.41	0.03	0.03
	Post	6.16	0.39	0.34	7.87	0.64	0.44
	Total	29.41	0.77	0.58	103.23	1.03	0.69
SS	Pre	0.03	0.05	0.04	0.04	0.09	0.07
	Sel	0.01	0.02	0.02	0.02	0.03	0.03
	Post	0.04	0.07	0.05	0.09	0.01	0.00
	Total	0.08	0.14	0.11	0.14	0.13	0.10

Table 4.5: Performance results of Shakespeare works queries

Religion		1			2		
		XML	ToXin 1	ToXin 2	XML	ToXin 1	ToXin 2
LL*	Pre	0.00			0.00		
	Sel	0.00			0.00		
	Post	0.00			0.00		
	Total	0.00	0.00	0.00	0.00	0.00	0.00
LL	Pre	0.00			0.00		
	Sel	0.00			0.00		
	Post	0.00			0.00		
	Total	0.00	0.00	0.00	0.00	0.00	0.00
LS*	Pre	8.24	0.10	0.06	39.07	0.20	0.12
	Sel	0.01	0.02	0.02	0.08	0.02	0.20
	Post	0.85	0.02	0.02	0.99	0.05	0.03
	Total	9.10	0.14	0.10	40.14	0.27	0.36
LS	Pre	0.15	0.03	0.02	0.72	0.04	0.04
	Sel	0.01	0.02	0.02	0.03	0.02	0.02
	Post	0.63	0.02	0.01	0.97	0.03	0.02
	Total	0.79	0.07	0.06	1.73	0.09	0.08
SS*	Pre	8.57	0.11	0.07	45.37	0.22	0.14
	Sel	0.01	0.02	0.02	0.02	0.02	0.02
	Post	9.00	0.09	0.06	7.58	0.26	0.20
	Total	17.58	0.22	0.15	52.96	0.50	0.35
SS	Pre	0.01	0.03	0.02	0.02	0.07	0.06
	Sel	0.01	0.02	0.02	0.02	0.02	0.02
	Post	0.19	0.04	0.03	0.25	0.27	0.21
	Total	0.20	0.09	0.07	0.29	0.35	0.29

Table 4.6: Performance of religious texts queries

The results suggest that the first stage (pre-selection) generally benefits the most by the use of a path index scheme. All query types, except LS, SS in the Shakespeare works, and religious texts, have shown important performance improvements when using an index in the first stage. In some queries the performance of third stage (post-selection) is also considerably improved by using the index. Those query types with large query answers benefit the most by the use of a path index during the third query processing stage. From the results we can also conclude that, when using an index, the closer to the root the filter section starts, the better the improvement in the performance of the third stage and the worse that of

the first stage. Since dataguides support only the first stage, using only a dataguide without additional structures can do little to improve the performance of the entire query evaluation when the filter section is close to the root.

Regarding the results when using a value index, except for queries with small number of values selected in the filter section, the improvement is also considerable. Nevertheless, it is important to note that in none of the tested queries does the selection stage play an important role in the total performance of the query evaluation.

Figures 4.7, *4.8*, *4.9*, and *4.10* summarize the performance of four different index schemas: both versions of ToXin, dataguides, and dataguides with value indexes. In all of them ToXin outperforms the other two schemes, in some cases by one order of magnitude.

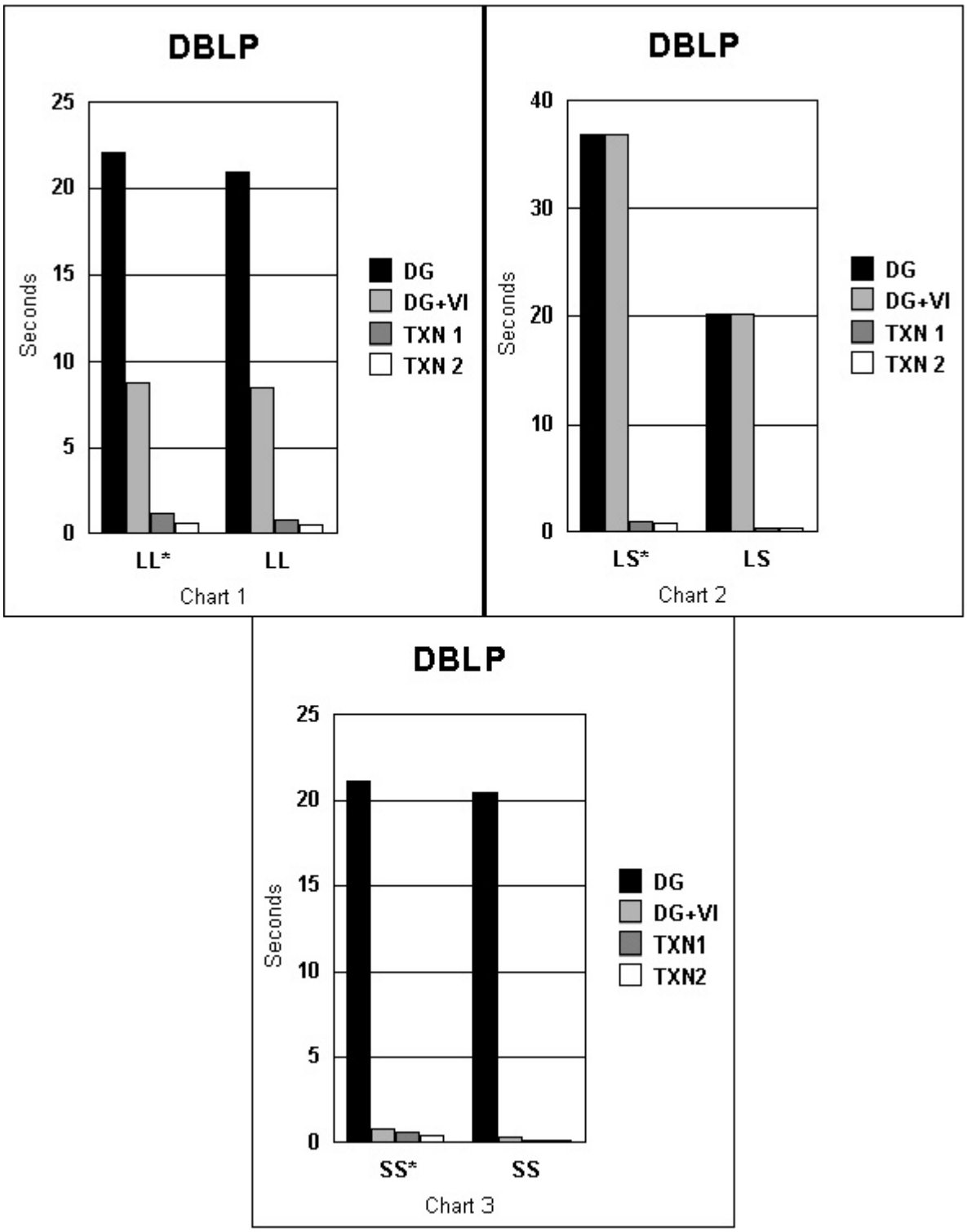


Figure 4.7: Comparative performance of selected index schemes with DBLP.

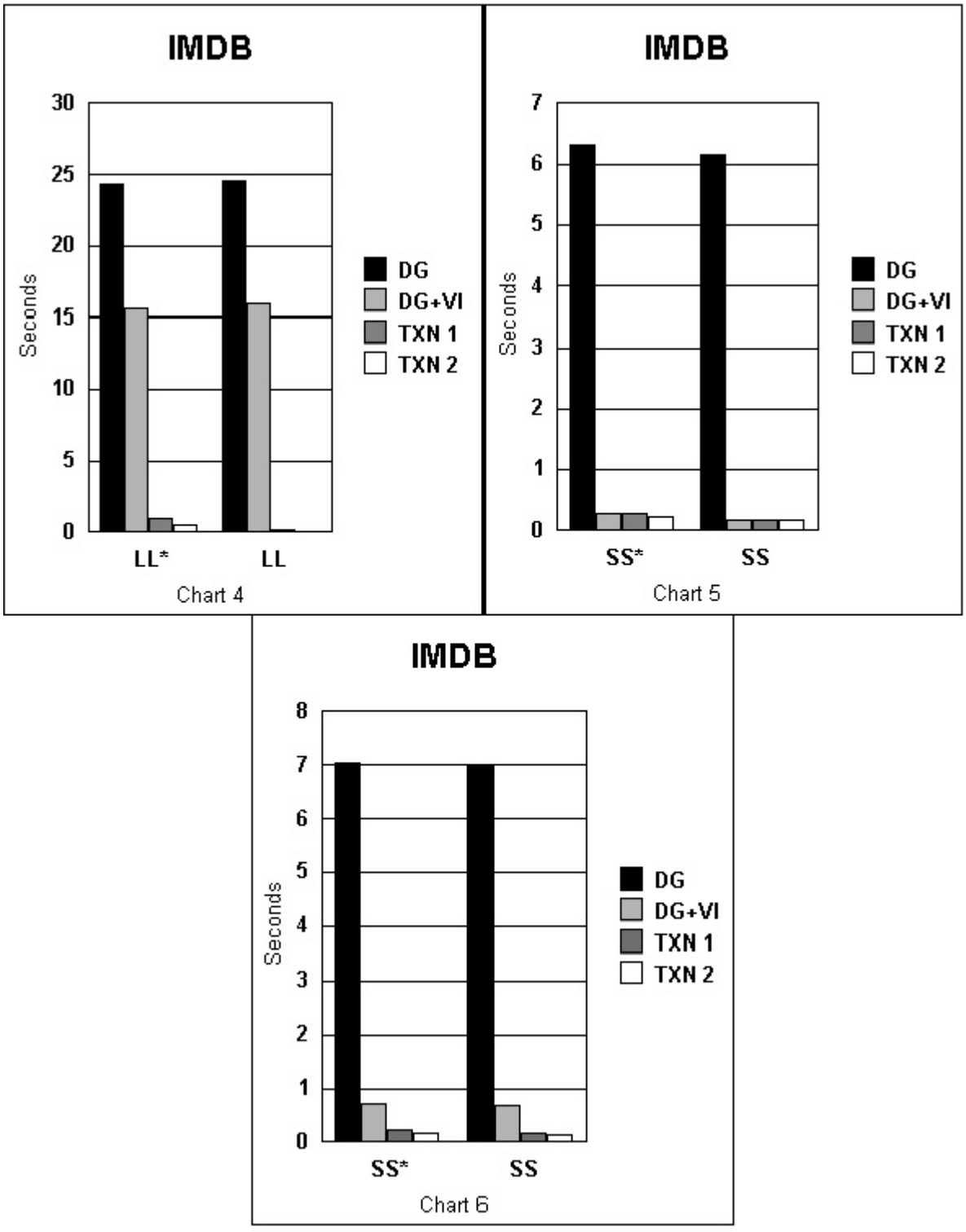


Figure 4.8: Comparative performance of selected index schemes with IMDB.

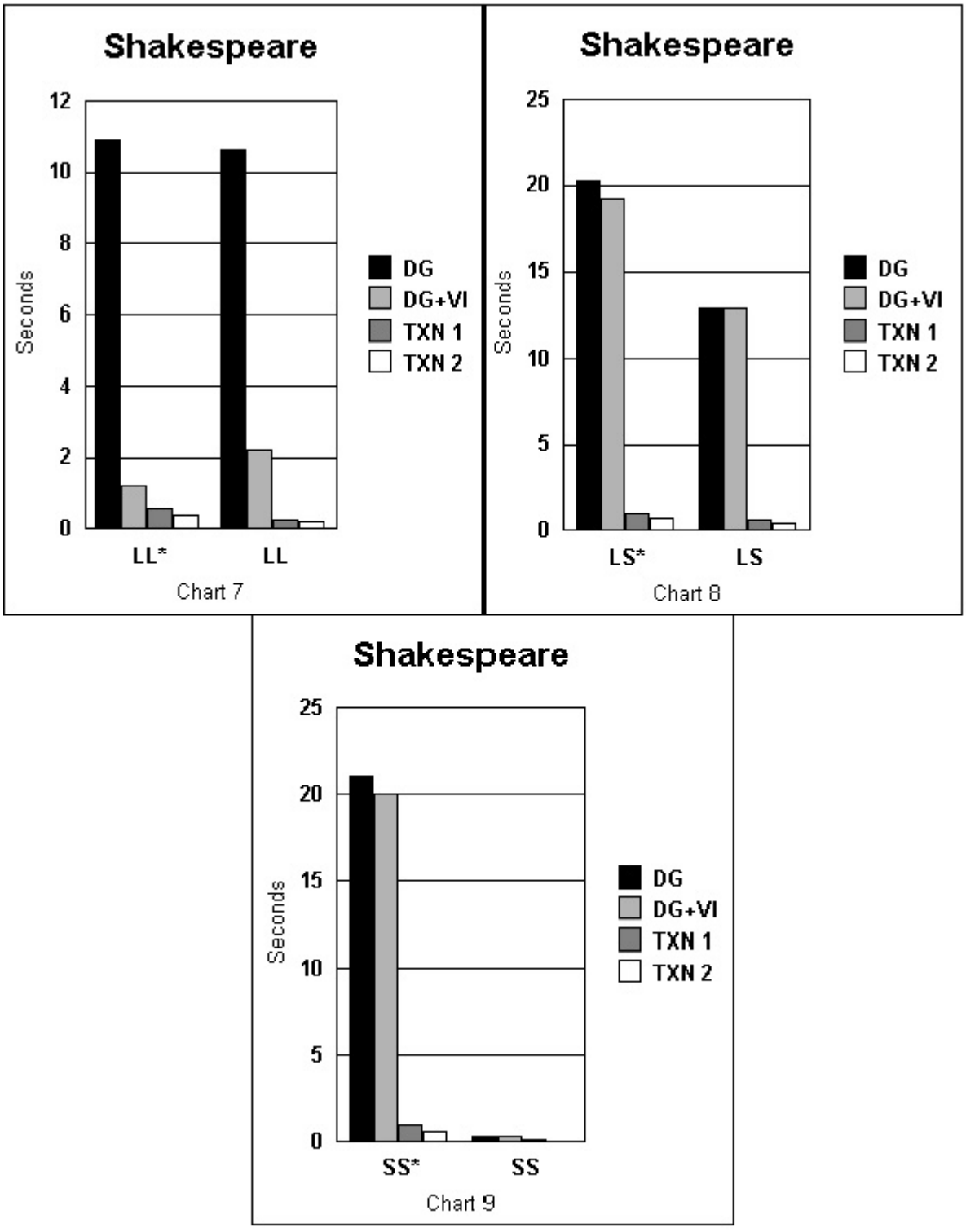


Figure 4.9: Comparative performance of selected index schemes with Shakespeare works.

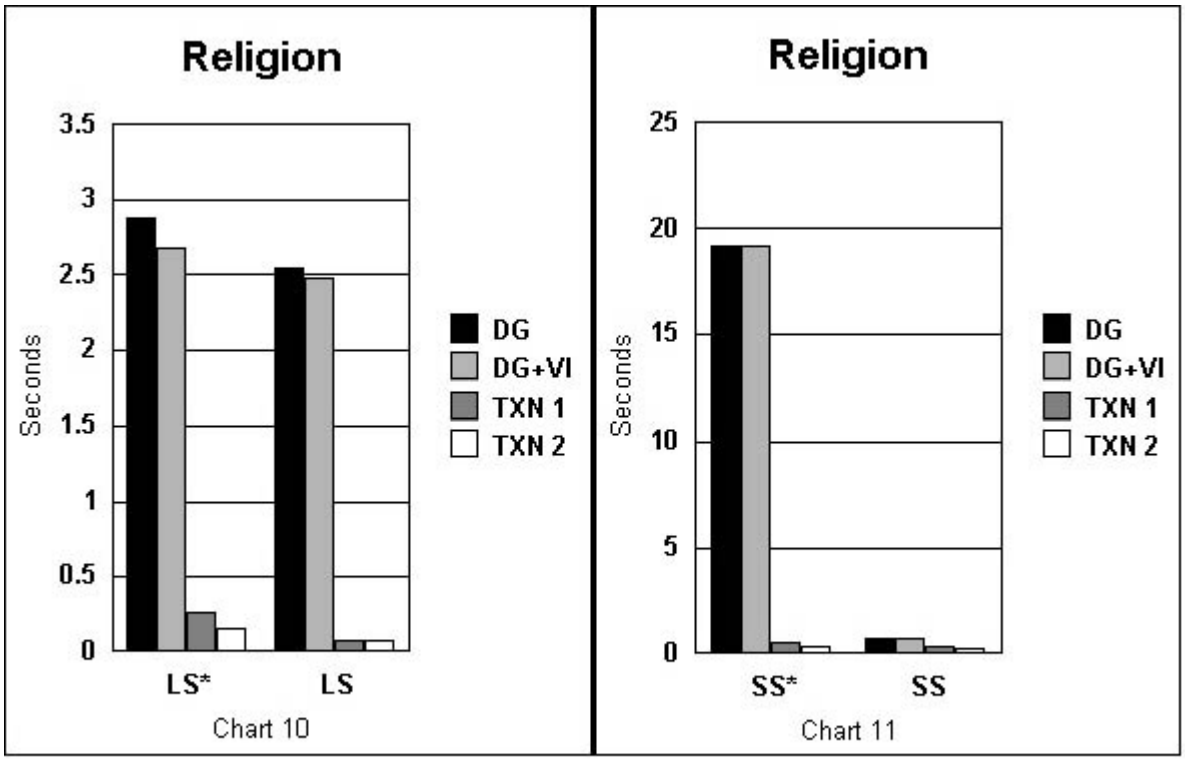


Figure 4.10: Comparative performance of selected index schemes with the religious texts.

Chapter 5

Conclusions and Future Work

5.1 Summary

In this thesis we presented *ToXin*, an indexing scheme that supports path queries over XML data. We discussed its architecture, related work and presented performance results. The motivation for this work was to overcome some limitations of current indexing proposals for semistructured data, such as the lack of support for all query processing stages (dataguides, 1-indexes and 2-indexes), and the need for an explicit specification of the paths to index (T-indexes). To that end, we combined ideas from dataguides and access support relations in a way that allows us to use the index in all the query evaluation stages for any general path query.

5.2 Future Work

The work presented in this thesis can be extended en several ways. The main directions are: adding order to the index structure; implementing the ToXin graph, by extending the ToXin tree with the semantic of the IDRefs; making the index persistent; and investigating ways to extend ToXin so it can be used as an alternative to DOM for storing, querying and updating XML documents. We give next a brief overview of these possible extensions.

Adding order

We are considering two options for this extension. One is to replace the Hash structures in the instance tables with a data structure that supports order, such as B-trees. Other option is to give the index more flexibility by keeping the order information in a separate structure that may be added or not depending on the particular user's needs. This new *order tables* would be attached to the ToXin nodes and would contain information about the order in which the elements of the instance and value tables appear in the original XML document. With this addition, ToXin will support queries involving order of XML elements and attributes.

Extending the graph index with IDRefs

In order to construct the ToXin graph from the XML graph we use *Algorithm 5.1*. It performs a depth-first search of the underlying tree (the element nesting tree) and for each edge checks whether it is an attribute or an element. Since attributes do not have further structure, when the algorithm finds an attribute it simply adds it to the index and does not continue with the recursive search.

Algorithm 5.1: construction of a *ToXin graph* from an XML-*graph*

INPUT:

XML-graph $X_D = (N, v_0, E, A, \psi, \Sigma, \lambda, <)$

OUTPUT:

ToXin Graph $I_X = (N_I, x_0, E_I, A_I, \psi_I, \Sigma, \lambda_I, X_D, \sigma)$

METHOD:

$E_I \leftarrow \phi$

$N_I \leftarrow$ new node x_0

Index(x_0, v_0)

procedure *Index*(x, v) // $x \in N_I$, $v \in N$

for each edge $e \in E$ s.t. $\psi(e) = (v, w)$ do

if there is no edge $e_I \in E_I$ s.t. $\psi_I(e_I) = (x, y)$ and $\lambda_I(e_I) = \lambda(e)$ then

$E_I \leftarrow E_I \cup \{$ new edge $e_I \}$

$N_I \leftarrow N_I \cup \{$ new node $y \}$

$\lambda_I(e_I) \leftarrow \lambda(e)$

$\psi_I(e_I) \leftarrow (x, y)$

fi

$\sigma(e_I, v) \leftarrow w$

If $e \in A$ then

$A_I \leftarrow A_I \cup \{ e_I \}$

else

Index(y,w)

fi

od

end

Making the index persistent

In its current implementation ToXin is kept in main memory. To make the index persistent we are currently considering two approaches: mapping the index to a RDBMS or to an OODBMS. Some hybrid scheme is also possible, since the value and instance tables are more naturally suited to be stored in relational tables whereas the schema graph can be best described using an object-oriented data model. A number of methods for storing XML

documents using relational and object-oriented databases have been proposed over the past few years (see [Bar00] for a survey). Some of these approaches can be adapted for storing ToXin, and we plan to study in particular two of them: Ozone [WLA99] and Edge Tables [FK99].

Extending ToXin with DOM functionality

Since the DOM tree is used only to populate the instance and value tables and all further query processing is done over the index, once we add order to ToXin we can discard the DOM structure for query purposes. However, if we want to have the full DOM functionality, we need to implement the core class interfaces of DOM. *Table 5.1* contains a short description of the minimum set of class interfaces that needs to be implemented.

Class Interface Name	Description
Attr	It represents an attribute within an element object. DOM views attributes as properties of elements rather than objects itself.
CDATASection	It represents a CDATA section, primarily used for XML fragments.
CharacterData	It provides methods for string manipulation.
Comment	It represents the content of an XML comment.
Document	It represents the root of the DOM tree.
Document Fragment	It represents a subtree or a set of subtrees of DOM tree.
DocumentType	It provides access to the list of entities defined in the DTD.
Element	It represents an element in an XML document.
Entity	It represents an entity in an XML document.
NamedNodeMap	It represents an unordered collection of nodes that can be accessed by a name.
Node	It represents a single node in the DOM tree.
NodeList	It represents an ordered collection of nodes.
Notation	It represents a notation defined in the DTD. A notation declares the format of an unparsed entity.
ProcessingInstruction	It represents a processing instruction in an XML document. A processing instruction contains processor-specific information stored in the text of the document.
Text	It represents the textual content of an element object or attribute.

Table 5.1: DOM class interfaces.

Appendix 1

Complete Example

We present here the full extension of value and instance tables corresponding to *Figure 3.5.*

publications / issue / title / text

Node	Value
5	"pub. title 1"
10	"pub. title 2"

publications / issue / type / text

Node	Value
6	"journal"
11	"conference"

publications / issue / year / text

Node	Value
8	"1990"
12	"1990"

publications / issue / publisher / text

Node	Value
9	"ACM"

publications / issue /articles /article / title / text

Node	Value
18	"title 1"
21	"title 2"
24	"title 3"
26	"title 4"

Node	Value
20	"1-20"
23	"21-40"

publications / issue /articles /article / authors / author / text

Node	Value
28	"author 1"
29	"author 2"
30	"author 3"
31	"author 4"
32	"author 5"
33	"author 6"

Figure A1.1: Value tables for the index tree of Figure 3.5

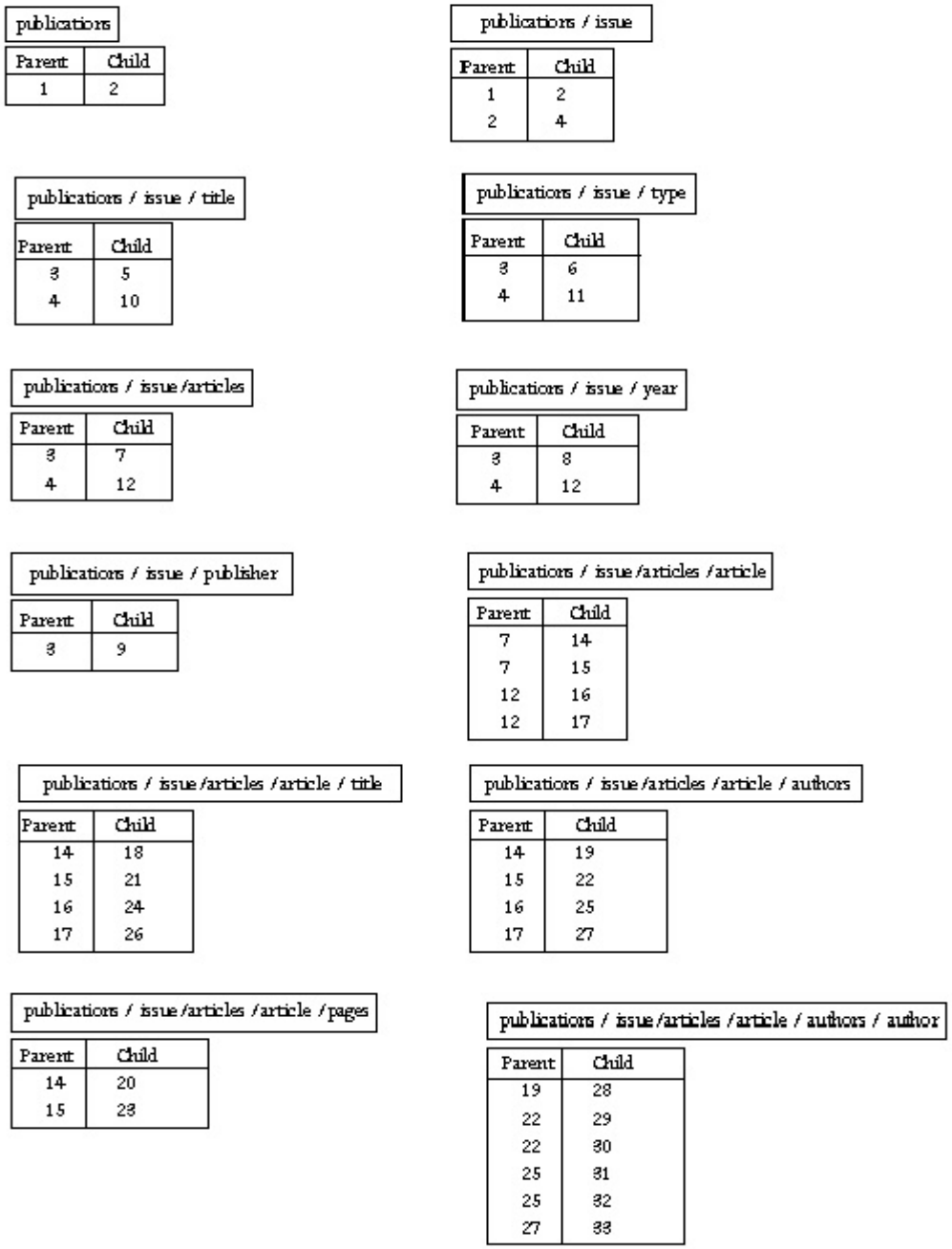


Figure A1.2: Instance tables for the index tree of Figure 3.5

Appendix 2

Additional Algorithms

Algorithm A2.1 (navigateDown): computation of a forward navigation over an XML-graph

INPUT:

XML-graph $X_D = (N, v_0, E, A, \psi, \Sigma, \lambda, <)$

query graph $Q = (S, s_0, E_Q, \psi_Q, \lambda_Q, F)$

OUTPUT:

ans(Q(X_D))

METHOD:

Result $\leftarrow \phi$
Added $\leftarrow \phi$
Visited $\leftarrow \phi$
SearchDown(v_0 , s_0)

procedure *Navigate*(*v, s*)
    Visited $\leftarrow$ *Visited* + (*v, s*)
    for each edge $e \in E$ s.t. $\psi(e) = (v, w)$ do
        for each edge $e_Q \in E_Q$ s.t. $\psi_Q(e_Q) = (s, t)$ and $\lambda_Q(e_Q) = \lambda(e)$ do
            if $t \in F$ and $e \notin$ *Added* then
                Added $\leftarrow$ *Added* $\cup \{e\}$
                Result $\leftarrow$ *Result* $\cup \{w\}$
            fi
            if $(w, t) \notin$ *Visited* then
                SearchDown(*w, t*)
            fi
        od
    od
end

Algorithm A2.2 (navigateDown): computation of a forward navigation over a dataguide

INPUT:

index-graph $D_x = (N_I, x_0, E_I, \psi_I, \Sigma, \lambda_I, X_D, ext)$, where $X_D = (N, v_0, E, \psi, \Sigma, \lambda, <)$

query-graph $Q = (S, s_0, E_Q, \psi_Q, \lambda_Q, F)$

OUTPUT:

ans($Q(X_D)$)

METHOD:

Result $\leftarrow$ *ext*(x_0)
Added $\leftarrow \phi$
Visited $\leftarrow \phi$
SearchDown(x_0 , s_0)

procedure *Search*(*x, s*)
Visited $\leftarrow$ *Visited* $\cup$ {(*x, s*)}
for each edge $e_I \in E_I$ s.t. $\psi_I(e_I) = (x, y)$ do
for each edge $e_Q \in E_Q$ s.t. $\psi_Q(e_Q) = (s, t)$ and $\lambda_Q(e_Q) = \lambda(e_I)$ do
if $t \in F$ and $e_I \notin$ *Added* then
Added $\leftarrow$ *Added* $\cup$ { e_I }
Result $\leftarrow$ *Result* $\cup$ *ext*(*y*)
fi
if (*w, t*) $\notin$ *Visited* then
SearchDown (*w, t*)
fi
od
od
end

Appendix 3

Comparative Performance Results

We present in this Appendix the complete results of the comparative performance experiments summarized in *Figure 4.7, 4.8, 4.9* and *4.10.*

DBLP		DG	DG + VI	ToXin1	ToXin 2
LL*	Pre	0.46	0.46	0.46	0.26
	Sel	13.28	0.03	0.03	0.03
	Post	8.33	8.33	0.66	0.35
	Total	22.07	8.82	1.15	0.64
LL	Pre	0.09	0.09	0.09	0.07
	Sel	12.52	0.03	0.03	0.03
	Post	8.36	8.36	0.67	0.40
	Total	20.97	8.47	0.79	0.50
LS*	Pre	0.39	0.39	0.39	0.31
	Sel	0.10	0.04	0.04	0.04
	Post	36.40	36.40	0.64	0.47
	Total	36.89	36.83	1.06	0.82
LS	Pre	0.03	0.03	0.03	0.02
	Sel	0.05	0.04	0.04	0.04
	Post	20.13	20.13	0.34	0.29
	Total	20.21	20.19	0.40	0.34
SS*	Pre	0.55	0.55	0.55	0.30
	Sel	20.42	0.09	0.09	0.09
	Post	0.17	0.17	0.01	0.01
	Total	21.14	0.81	0.65	0.40
SS	Pre	0.08	0.08	0.08	0.07
	Sel	20.19	0.09	0.09	0.09
	Post	0.16	0.16	0.02	0.01
	Total	20.43	0.33	0.19	0.17

Figure A3.1: Comparative performance results of DBLP queries

IMDB		DG	DG + VI	ToXin1	ToXin 2
LL*	Pre	0.06	0.06	0.06	0.03
	Sel	8.71	0.04	0.04	0.04
	Post	15.56	15.56	0.84	0.49
	Total	24.34	15.66	0.94	0.56
LL	Pre	0.03	0.03	0.03	0.02
	Sel	8.66	0.04	0.04	0.04
	Post	15.86	15.86	0.05	0.05
	Total	24.54	15.93	0.12	0.11
LS*	Pre	0.00	0.00	0.00	0.00
	Sel	0.00	0.00	0.00	0.00
	Post	0.00	0.00	0.00	0.00
	Total	0.00	0.00	0.00	0.00
LS	Pre	0.00	0.00	0.00	0.00
	Sel	0.00	0.00	0.00	0.00
	Post	0.00	0.00	0.00	0.00
	Total	0.00	0.00	0.00	0.00
SS*	Pre	0.14	0.14	0.14	0.08
	Sel	6.17	0.14	0.14	0.14
	Post	0.01	0.01	0.01	0.00
	Total	6.31	0.28	0.28	0.22
SS	Pre	0.04	0.04	0.04	0.03
	Sel	6.11	0.14	0.14	0.14
	Post	0.01	0.01	0.01	0.00
	Total	6.16	0.18	0.18	0.18
SS*	Pre	0.08	0.08	0.08	0.04
	Sel	6.39	0.06	0.06	0.06
	Post	0.58	0.58	0.08	0.05
	Total	7.04	0.72	0.22	0.15
SS	Pre	0.04	0.04	0.04	0.03
	Sel	6.40	0.06	0.06	0.06
	Post	0.58	0.58	0.08	0.05
	Total	7.02	0.68	0.18	0.14

Figure A3.2: Comparative performance results of IMDB queries

Shakespeare		DG	DG + VI	ToXin1	ToXin 2
LL*	Pre	0.43	0.43	0.43	0.26
	Sel	9.73	0.05	0.05	0.05
	Post	0.75	0.75	0.12	0.09
	Total	10.91	1.23	0.60	0.40
LL	Pre	0.10	0.10	0.10	0.09
	Sel	8.45	0.05	0.05	0.05
	Post	2.08	2.08	0.09	0.07
	Total	10.63	2.23	0.24	0.22
LS*	Pre	0.40	0.40	0.40	0.27
	Sel	1.00	0.00	0.00	0.00
	Post	18.86	18.86	0.62	0.43
	Total	20.27	19.26	1.02	0.70
LS	Pre	0.09	0.09	0.09	0.08
	Sel	0.01	0.00	0.00	0.00
	Post	12.76	12.76	0.49	0.39
	Total	12.86	12.85	0.58	0.47
SS*	Pre	0.36	0.36	0.36	0.22
	Sel	1.02	0.00	0.00	0.00
	Post	19.69	19.69	0.64	0.44
	Total	21.06	20.04	1.00	0.66
SS	Pre	0.09	0.09	0.09	0.07
	Sel	0.01	0.00	0.00	0.00
	Post	0.22	0.22	0.01	0.00
	Total	0.31	0.31	0.10	0.07

Figure A3.3: Comparative performance results of Shakespeare works queries

Religion		DG	DG + VI	ToXin1	ToXin 2
LL*	Pre	0.00	0.00	0.00	0.00
	Sel	0.00	0.00	0.00	0.00
	Post	0.00	0.00	0.00	0.00
	Total	0.00	0.00	0.00	0.00
LL	Pre	0.00	0.00	0.00	0.00
	Sel	0.00	0.00	0.00	0.00
	Post	0.00	0.00	0.00	0.00
	Total	0.00	0.00	0.00	0.00
LS*	Pre	0.20	0.20	0.20	0.12
	Sel	0.20	0.01	0.01	0.01
	Post	2.48	2.48	0.05	0.03
	Total	2.88	2.68	0.25	0.16
LS	Pre	0.04	0.04	0.04	0.04
	Sel	0.08	0.01	0.01	0.01
	Post	2.44	2.44	0.03	0.02
	Total	2.55	2.48	0.08	0.07
SS*	Pre	0.22	0.22	0.22	0.14
	Sel	0.01	0.00	0.00	0.00
	Post	18.95	18.95	0.26	0.20
	Total	19.18	19.17	0.48	0.34
SS	Pre	0.07	0.07	0.07	0.06
	Sel	0.00	0.00	0.00	0.00
	Post	0.64	0.64	0.27	0.21
	Total	0.70	0.70	0.33	0.27

Figure A3.4: Comparative performance results of religious texts queries

Appendix 4

Tree Schemas of Document Samples

We include in this appendix the tree schemas of the documents used in the experiments. These tree schemas show the general path structure of the document samples (DBLP, IMDB, Shakespeare works and religious texts)

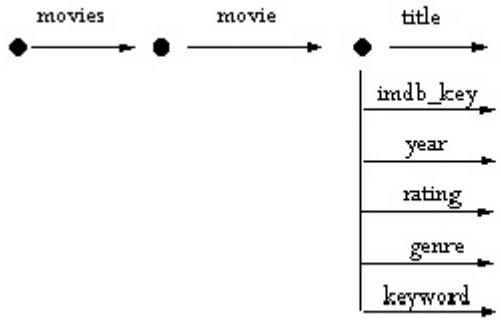


Figure A3.1: Tree schema of the International Movies Database sample

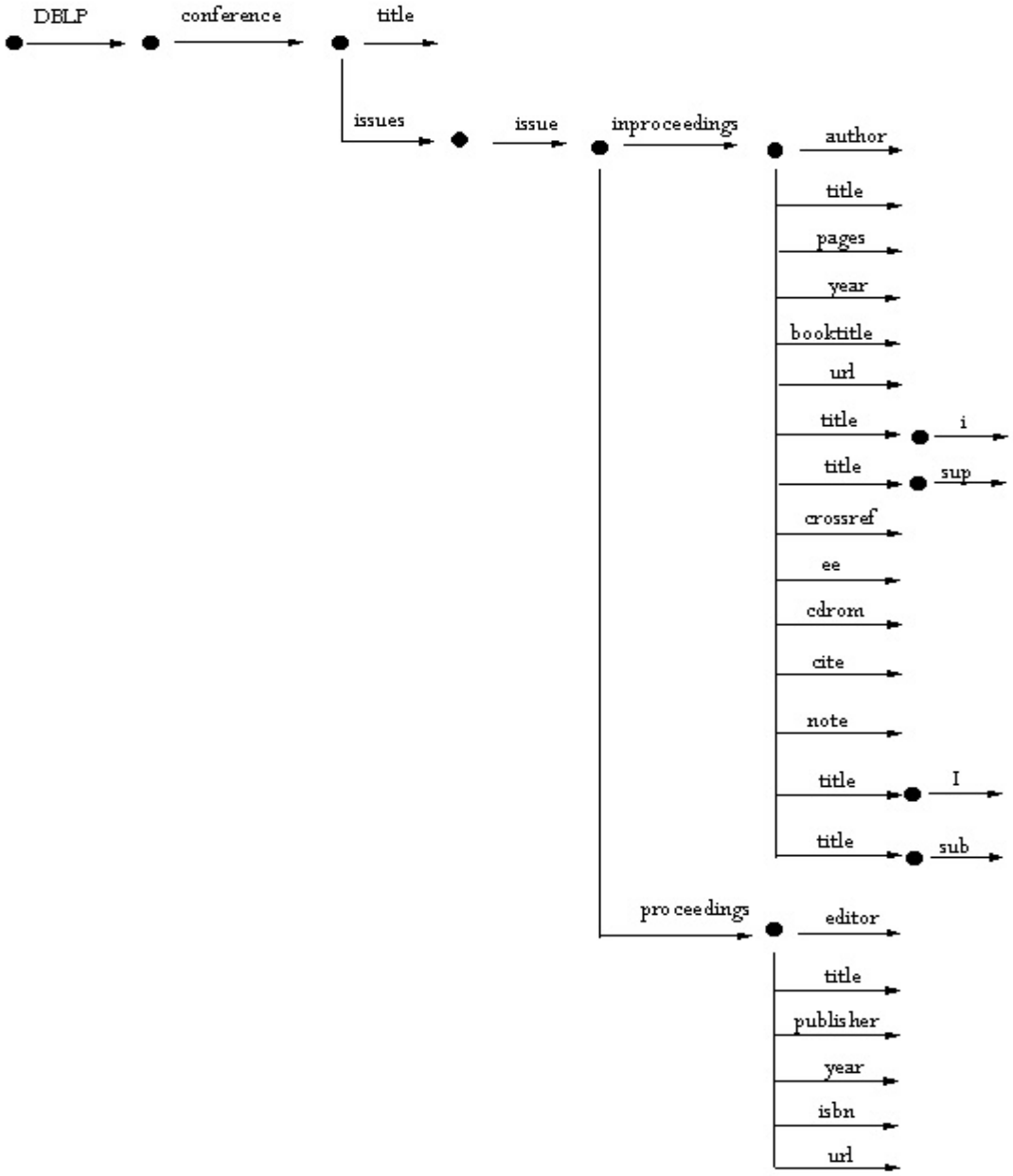


Figure A3.2: Tree schema of the DBLP sample

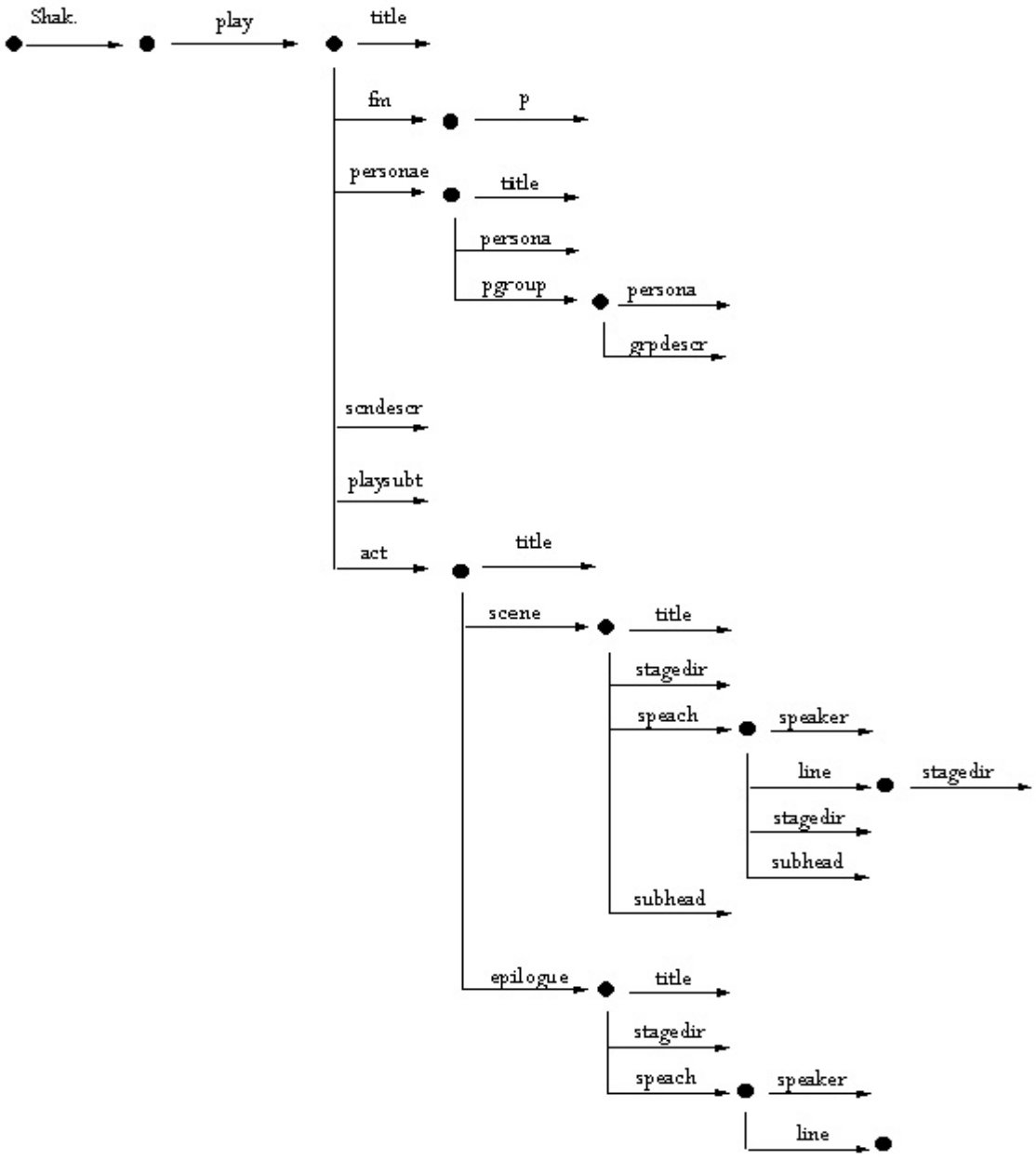


Figure A3.3: Tree schema of the Shakespeare Works sample

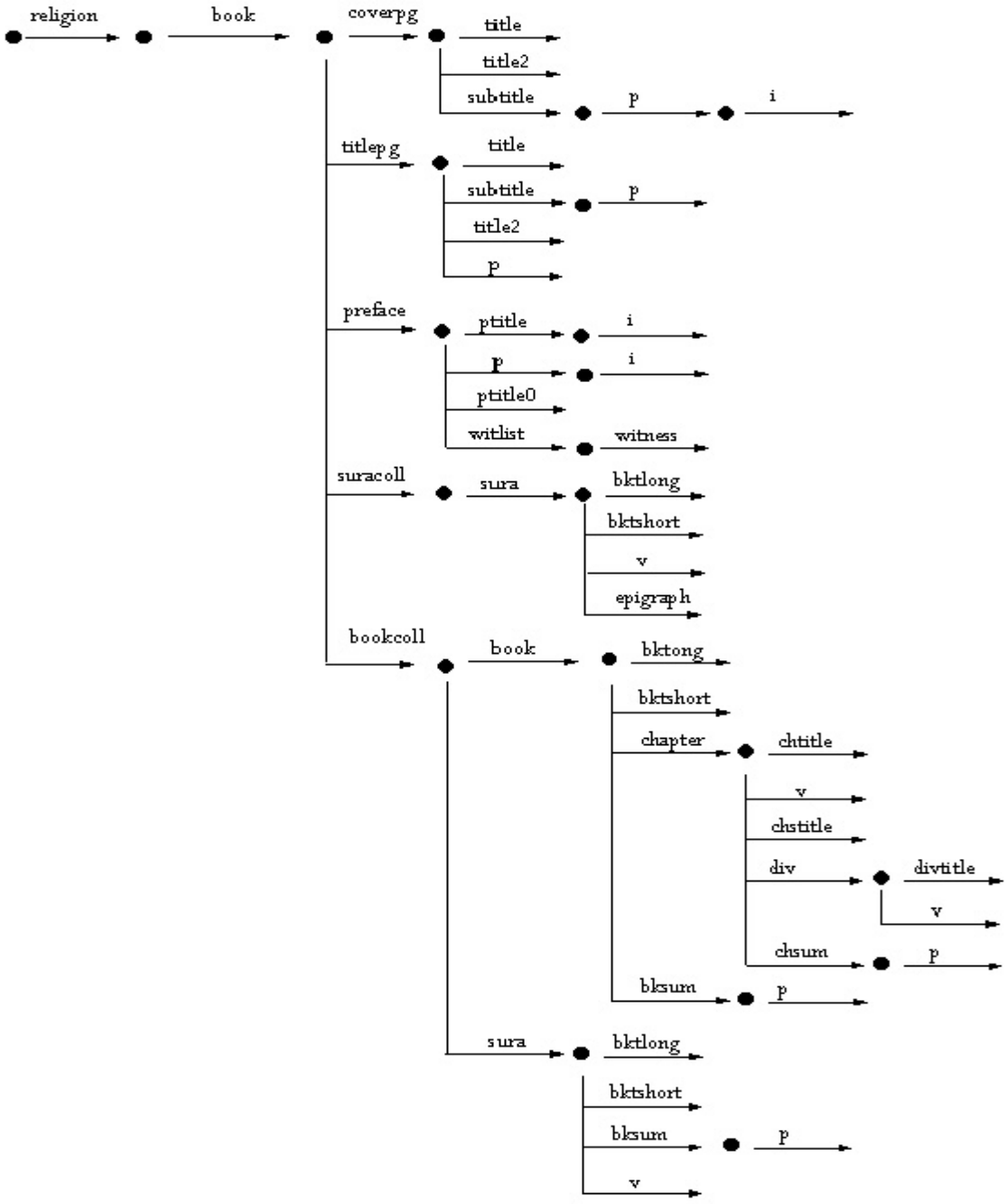


Figure A3.4: Tree schema of the Religious Texts sample

References

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 1999.

[AM98] G. Arocena and A. Mendelzon, WebOQL: Restructuring documents, databases and webs. *Proceedings of the IEEE International Conference on Data Engineering*, pages 24-33, 1998.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1): 68-88, April 1997.

[Aro97] Gustavo Arocena. WebOQL: Exploiting Document Structure in Web Queries. Master's Thesis, Department of Computer Science, University of Toronto, 1997.

[ASU86] Alfred Aho, Ravi Sethi and Jeffrey Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, 1986.

[Bar00] Denilson Barbosa. On Storage Alternatives for Semistructured Data. Oral Qualifying Report, Department of Computer Science, University of Toronto, 2000.

[BC00] Angela Bonifati, Stefano Ceri. Comparative analysis of five XML query languages. *SIGMOD Record* 29(1): 68-79.

[BDFS97] P. Buneman, S. Davidson, G. Hillebrand and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336-350, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505-516, 1996.

[BFS00] Peter Buneman, Mary F. Fernandez and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal* 9(1): 76-110, 2000.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2): 196-214, 1989.

[Bos98] Jon Bosak. Religious Texts in XML. http://www.ibiblio.org/xml/examples/religion. 1998.

[Bos99] Jon Bosak. Complete Plays of Shakespeare in XML. http://www.ibiblio.org/xml/examples/shakespeare. 1999.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 313-324, 1994.

[CCM96] Vassilis Christophides, Sophie Cluet and Guido Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 413-422, 1996.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy adn D. Suciu. A query language for XML. In *Proceedings of the 8th International World Wide Web Conference, WWW8*, 1999.

[FFLS97] M. Fernandez, D. Florescu, A. Levy and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3): 4-11, 1997.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using and RDBMS. *IEEE Data Engineering Bulletin* 22(3): 27-34, 1999.

[FS98] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 14-23, 1998.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Databases*, pages 436-445, 1997.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[IMDB00] The Internet Movie Database. http://www.imdb.com. 2000.

[KKS92] M. Kifer, W. Kim and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393-402, 1992.

[KM90] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings of the International Conference on Very Large Databases*, pages 294-305, 1990.

[KM92] Alfons Kemper and Guido Moerkotte. Access support relations: an indexing method for object bases. *Information Systems*, 17(2): 117-145, 1992.

[Ley00] Michael Ley. DBLP database web site. http://www.informatik.uni-trier.de/~ley/db. 2000.

[Lie99] Hartmut Liefke. Horizontal query optimization on ordered semistructured data. *Informal Proceedings of the International Workshop on the Web and Databases*, pages 61-66, 1999.

[LS00] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153-164, 2000.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass and J. Widom. Lore: a database management system for semistructured data. SIGMOD Record 26(3): 54-66, 1997.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277-295, 1999.

[MW97] Jason McHugh and Jennifer Widom. Query optimization for semistructured data. *Technical Report, Stanford University*, 1997.

[MW99] J. McHugh and J. Widom. Query Optimization for XML. *Proceedings of the International Conference on Very Large Databases*, pages 315-326, 1999.

[NUWC97] Svetlozar Nestorov, Jeffrey Ullman, Janet Wiener and Sudarashan Chawathe. Representative Objects: concise representations of semistructured, hierarchical data. *Proceedings of the IEEE International Conference on Data Engineering*, pages 79-90, 1997.

[RLS98] J. Robie, J. Lapp and D. Schach. XML Query Language (XQL). In *Proceedings of the Query Languages Workshop*, 1998.

[SB96] B. Shidlowsky and E. Bertino. A graph-theoretic approach to indexing in object-oriented databases. *Proceedings of the IEEE International Conference on Data Engineering*, pages 230-237, 1996.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2): 218-246, 1987.

[WLA99] J. Widom, T. Lahiri, S. Abiteboul. Ozone: Integrating structured and semistructured data. In *Proceedings of the International Conference on Database Programming Languages*, 1999.

[W3C98] W3C Recommendation. Document Object Model (DOM) Level 1 Specification. In *http://www.w3.org/TR/REC-DOM-Level-1*. 1998

[W3C99a] W3C Recommendation. XML Path Language (XPath) 1.0. In *http://www.w3.org/TR/xpath*. 1999.

[W3C99b] W3C Recommendation. XSL Transformations (XSLT) 1.0. In http://www.w3.org/TR/xslt. 1999.

[W3C00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, Eve Maler (Eds.). Extensible Markup Language (XML) 1.0 (Second Edition). In *http://www.w3.org/TR/REC-xml*.