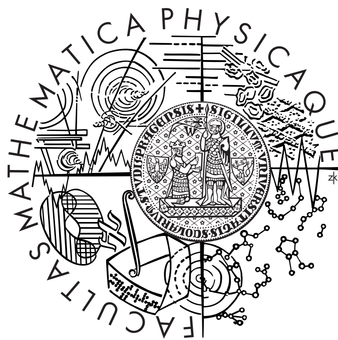
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Tomáš Bartoš

Native Indexing of Large XML Databases

Department of Software Engineering

Advisor: Doc. RNDr. Tomáš Skopal, Ph.D.
Study programme: Software Systems

2010

I would like to thank my advisor, Doc. RNDr. Tomáš Skopal, Ph.D., for his precious advice and time he spent consulting my problems and ideas, my family that supported me at any time, and all others that helped me in any way while working on this thesis.

I hereby proclaim that I worked on this thesis on my own using the referred resources only. I agree with public availability and with publishing of this thesis.

In Prague on April 14th, 2010 Bc. Tomáš Bartoš

Contents

1 Introduction **6**
- 1.1 Contributions . . . 7

2 XML technologies **8**
- 2.1 XPath . . . 9
- 2.2 Graph-structured Data . . . 10
- 2.3 Indexing XML Data . . . 11
  - 2.3.1 Numbering Schemes . . . 12
  - 2.3.2 Structural Joins . . . 13
  - 2.3.3 Mapping to Relational Databases . . . 15
  - 2.3.4 Multi-dimensional Approaches . . . 17

3 iXUPT: Indexing **21**
- 3.1 Overview . . . 21
- 3.2 Indexing Paths . . . 22
- 3.3 Structures . . . 23
  - 3.3.1 NodeTags Table . . . 24
  - 3.3.2 Paths Table . . . 24
  - 3.3.3 Prefix Tree . . . 25
- 3.4 Build Index . . . 26
  - 3.4.1 Start Element . . . 26
  - 3.4.2 End Element . . . 28
- 3.5 Variations . . . 28
  - 3.5.1 Numbering Scheme . . . 29
  - 3.5.2 Rho-Index Structure . . . 29
- 3.6 Implementation Details . . . 32

4 iXUPT: Evaluating XPath Queries **34**
- 4.1 Evaluating Queries . . . 34
- 4.2 Parsing XPath Expressions . . . 35
  - 4.2.1 XPathTree . . . 35

4.2.2 XPathNodes . . . 36
4.3 XPathTree Pre-processing . . . 37
4.4 XPathTree Evaluating . . . 37
4.4.1 Get and Save Candidates . . . 38
4.4.2 Evaluate Predicates and Voting . . . 40
4.4.3 Filter Candidates . . . 41
4.4.4 Merge Candidates . . . 41

5 Experimental Results **42**
5.1 Data Sets . . . 42
5.2 Indexing . . . 43
5.2.1 Index Size . . . 43
5.2.2 Creation Time . . . 44
5.2.3 Paths Count . . . 46
5.3 Querying . . . 47
5.3.1 Sample Queries . . . 47
5.3.2 Query Results . . . 48
5.3.3 Competitors . . . 51

6 Conclusion **53**
6.1 Future Work . . . 54

Bibliography **56**

A CD-ROM Content **60**
A.1 Source Code . . . 60
A.2 Data Sets . . . 60
A.3 Test Results . . . 60
A.4 Documentation . . . 61
A.5 Supplemental Programs . . . 61

Title: *Native indexing of large XML databases*
Author: Bc. Tomáš Bartoš
Department: Department of Software Engineering
Supervisor: Doc. RNDr. Tomáš Skopal, Ph.D.
Supervisor's e-mail address: Tomas.Skopal@mff.cuni.cz

Abstract: In the present work we study the indexing methods for large XML databases and their time efficiency when evaluating path queries.

There are several ways of indexing XML data but we focus on indexing *root-to-leaf* paths and grouping them according to the common criteria, *path labels*. We study the existing methods and combine them in order to create the *iXUPT*, a novel native indexing concept using path templates, which leverages advantages of current approaches. We provide two variations of our solution depending on the way of handling ancestor-descendant relationships. The first one uses the proposed numbering scheme, while the second one relies on the Rho-Index structure. Furthermore, we prove the feasibility of our concept by the implemented prototype and by evaluating sample regular path expressions represented by *XPath* queries. We compare the variations between each other and also with other solutions.

Keywords: Indexing XML, Rho-Index, path-based indexing, XPath queries

Název práce: *Nativní indexování rozsáhlých XML databází*
Autor: Bc. Tomáš Bartoš
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. RNDr. Tomáš Skopal, Ph.D.
e-mail vedoucího: Tomas.Skopal@mff.cuni.cz

Abstrakt: V predloženej práci študujeme metódy indexovania pre rozsiahle XML databázy a ich časovú zložitosť pri vyhodnocovaní dotazov na vyhľadávanie ciest.

Existuje množstvo spôsobov, ako indexovať XML dáta, ale my sa zameriame na indexovanie ciest *od koreňa k listom* a ich zhlukovanie na základe podobných kritérii. Štúdium a správna kombinácia existujúcich metód nám slúži ako základ pre vytvorenie nového natívneho indexu *iXUPT*, ktorý používa značkovanie ciest. Predstavujeme dve variácie indexovania v závislosti od spôsobu zisťovania vzťahu predok-potomok. Prvá možnosť používa číselnú schému, druhá využíva Rho-Index. Správnosť nášho riešenia dokážeme implementáciou prototypu a vyhodnotením niekoľkých *XPath* dotazov predstavujúcich cesty v grafe. Nakoniec porovnáme jednotlivé varianty a dosiahnuté výsledky s existujúcimi riešeniami.

Klíčová slova: Indexovanie XML, Rho-Index, indexovanie ciest, XPath dotazy

Chapter 1

Introduction

In past few years there has been an expansion of semi-structured data mostly stored as XML [15] files and used for saving and exchanging information over the Internet. The XML files are purely text files with a specific structure containing the information that might be structured, and the simplicity is only one of the factors why the format became so popular.

As more and more files occurred in this format, we wanted to access the stored data and search for the specific information according to prior criteria. For this purpose languages such as XPath [13] or XQuery [14] have been created. They allow searching for elements, attributes, or text values based on either specific values or regular expressions. Moreover, they enable us to create simple or complex queries and specify several search conditions in human readable format. If there are multiple conditions in an XPath query, we can combine them, and we get the *regular path expression* pattern.

The path expression usually matches several elements in the input XML file that finally create the result set. The challenge is to find these elements quickly and efficiently, especially in large files with a high number of elements and with different structures. When the structure of input files is widely known or there exists an XML schema (the description of component parts in DTD [10] or XML Schema [16]), it is always easier to answer queries regarding the content because we can leverage the structure of documents. But when we have to inspect a general XML file, it might be difficult to find the result efficiently because the file might be quite large with lots of distinct elements. Therefore one came with the idea of *indexing* the XML data in order to quickly get the result for any query and possibly for any XML file.

Indexing XML files means that we parse files and extract the valuable information. Afterwards, we store it, probably in other format, into some structures that are ready to answer our future queries. We might combine the existing values

with additional information that will help us to speed up the process of evaluating queries. Which data we save and how the information is stored depend exclusively on the indexing method.

No matter which indexing technique we use, if we had a regular path expression, the most problematic queries would be those with `'//'` (relative paths) or `'*'` (wildcards). These queries match numerous distinct elements and are difficult to handle compared to expressions with absolute paths only. The reason is that it is difficult to find out exactly which elements will be matched because we have to either compare the element names with wildcards or, in case of relative paths, skip some elements on a path from one element to another.

The indexing methods try to, in some way, improve the evaluation in terms of the time or the space complexity (memory used for storing the index for XML files), so they usually focus on specific queries. The best concept would be a *native XML index*, which takes the XML as the fundamental unit and that is effective for any query. Because there is always the trade-off between time and space requirements, we will focus on the effectiveness and on decreasing the time complexity.

1.1 Contributions

In this thesis, our aim is to combine the concepts of existing indexing methods and enhance them in order to improve the evaluation time of XPath queries. We analyze and focus mostly on the techniques using paths' labeling and indexing paths. To achieve our goal, we make contributions to several areas.

- The previous research showed that indexing paths is one of the effective ways of indexing XML documents, so we use this approach and we create *a novel indexing method* based on indexing paths using path templates (iXUPT).

- Additionally, we provide *two variations* of the proposed method. The first one uses a special numbering scheme for revealing ancestor-descendant relationships. For the second option, we replace the numbering method with the existing structure that finds all paths between any two elements in the source file, Rho-Index [3], in order to accelerate the evaluation of XPath regular path expressions. Because the Rho-Index is a general structure, it might be applied on any type of graphs while we designed the numbering scheme for trees only.

- Finally we provide experimental results for both variations of the new concept in terms of the time complexity while building the index and evaluating sample XPath queries, and we show pros and cons for each method. We also compare the new indexing method with other existing solutions.

Chapter 2

XML technologies

The XML (eXtensible Markup Language) format has become the standard for data exchange because it is self-describing and it stores not only information but also the relationships between data. It is easy to use and it is human readable because of strictly distinguishing content and markups that we call *tags*. Simply speaking, *elements*, *attributes*, and *text values* as parts of an XML file define its tree structure and we use them mostly to store semi-structured data (see the structure of a sample XML in Figure 2.1).

This format separates the data layer from the presentation layer which means that we can visualize the same XML file in multiple ways. No matter of the visual style we apply, the data values remain the same. Many technologies benefit and rely on this feature such as service-oriented architecture (SOA [36]) and web services, XForms [17], or RSS feeds [43].

```
<?xml version="1.0" encoding="UTF-8"?>
<faculty name="Faculty of Mathematics and Physics">
  <contact>
    <address>
      <street>Ke Karlovu 3</street>
      <city>Prague</city>
    </address>
    <email>info@mff.cuni.cz</email>
    <phone>22191 1111</phone>
  </contact>
  <department id="1" name="Department of Software Engineering">
  ...
  </department>
</faculty>
```

Figure 2.1: The source of a sample XML file

There are two methods of processing an XML file: *DOM* [12] (*Document Object Model*) and *SAX* [7] (*Simple API for XML*). The first one builds the *DOM tree* that corresponds to the input XML file (leveraging the tree structure of XML files), while the second one processes the XML files as streams and it is upon users to specify which events and elements will be processed and which elements will be skipped.

Furthermore, we can define a schema for XML documents with a specific language for structural description such as DTD [10] or XML Schema [16]. The XML schema determines the sequence of elements, their relationships in terms of subelement containments, required and optional attributes, and we can validate input files against the schema. While the DTD format is older and simpler, using XSD (XML Schema Definition Language) brings additional features such as the type control or the restriction of values.

If there is a collection of XML documents, we might be speaking about an *XML database*. Depending on the target location, we can store these documents as files in the file system, transform and map the segments of an XML file to tables in a relational database, or use the native XML databases (e.g. [32], [23], or [40]).

In our work, we use XML files as the data format for the graph representation. Because we do not expect XML files with a particular structure as the input, we do not use any XML schema as the hint for parsing files.

2.1 XPath

The XPath (XML Path Language [13]) has been created to search for particular elements in an XML file but we can use it also for the navigation in the source file. An XPath expression is a string expression that consists of multiple absolute or relative path expressions. Each path expression addresses some nodes and it is context-specific. Absolute paths start with '/' and they represent the path from the first node, the root. For relative paths (starting with '//' or nothing) we have to define the context. The basic idea of using these types of paths for the navigation is similar to navigating in the directory structures in common file systems.

We can divide both types of path expressions into several steps. Each step produces the result, an unordered set of nodes, that will be used as the context for processing the next step (if any). So the final result is the product of evaluating the very last step in the given expression.

Example 1. *The relative path expression "`//contact/address`" has two steps. The first one selects all* `<contact>` *elements and, as the second step, it searches for* `<address>` *elements that are direct descendants of these elements.*

Table 2.1: XPath axes

XPath axis	**Type**	**Description**
`child`	F	Children (direct descendants) of the context node
`descendant`	F	The transitive closure of the `child` axis of the context node
`attribute`	F	Attributes of the context node
`self`	F	Only the context node itself
`descendant-or-self`	F	The context node itself or its `descendants`
`following`	F	Nodes following the context node in the document order (except for `descendants`)
`following-sibling`	F	Siblings of the context node that are `following`
`namespace`	F	Namespace nodes of the context node
`parent`	R	The parent of the context node (the first node on the path to the root)
`ancestor`	R	The transitive closure of the `parent` axis of the context node
`ancestor-or-self`	R	The context node itself or its `ancestors`
`preceding`	R	Nodes preceding the context node in the document order (except for `ancestors`)
`preceding-sibling`	R	Siblings of the context node that are `preceding`

For the navigation, we might define XPath step expressions with forward and reverse axes. The total number of 13 axes allow users to specify relationships between the nodes. For the forward axes (F), it might be the *child* or the *descendant*, for reverse axes (R), we can use the *parent* or the *ancestor* axis (see Table 2.1).

XPath expressions include several other features such as built-in functions, enabling wildcards in step expressions, using predicates for advanced conditions, testing nodes for names or values, and many others. With these functionalities we can easily build complex path expressions.

We can use the XPath expressions also as the major component in the XSLT transformations [11] or in XQuery. For our purpose, we will use XPath as the query description language.

2.2 Graph-structured Data

If we take XML files, they can be easily transformed into oriented graphs. The elements and attributes correspond to graph nodes and the edges are derived from the parent-child (or element-subelement) relationships. In practise, it means that anytime we visit an element, we create a new node in the graph and connect the new node with its parent. Every element, except for the root element, has a preceding element that we will treat as the node's parent. The orientation of the edge that connects two nodes will be always from the parent to the child.

Without loss of generality, we can use also unoriented graphs to represent the XML files instead of oriented ones. Everything will remain the same but we

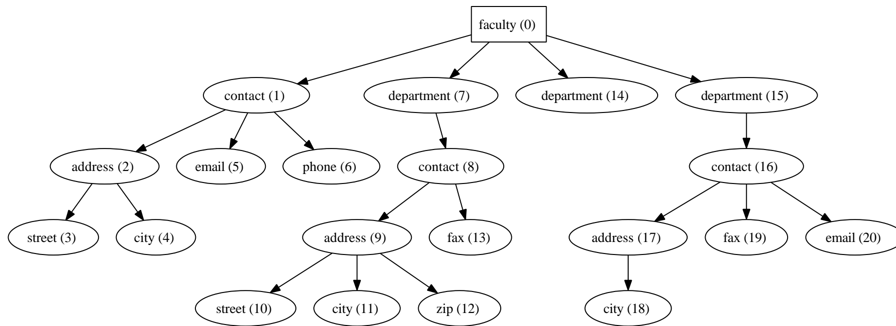


Figure 2.2: The sample XML file displayed as the oriented graph

think that oriented graphs are more transparent and we can clearly and easily see the parent-child relationships. Therefore in our method we will use the directed graphs and the terms elements/nodes or the parent-child relationships/edges will be interchangeable.

Generally, the graph from an XML file might contain loops but if we discard the `IDREF` attributes, we will get a *tree* (see Figure 2.2). The tree is a special type of a connected graph in which there are no loops and for each node, there is only a single path from the root.

Because our concept expects the tree structure as the input, we will focus primarily on indexing tree structures. Also some of the existing methods that we discuss in the following sections, index trees only.

2.3 Indexing XML Data

To find the right information in an XML file, we need to have a fast and an effective access to data. Similar to relational databases, we can create an *index* in order to speed up the querying for the information. Creating a new indexing concept means to define particular structures that hold some information about the input XML, to specify how we acquire and store the data, and to describe the algorithm that will use the structures when we evaluate a query.

There are various distinct approaches of indexing XML files. The main difference between them is that each method focuses on a specific topic such as decreasing the number of I/O operations [1], converting the XML format into tables in a relational DBMS to leverage the database engine ([21], [31], [45], [9], or GRIPP [41]), or relying on the simplicity of numbering schemes and joining elements (XISS [29] or twig joins [8]). The indexes might be based on a known data structure such as Patricia tries [24] that are used in [18] or [1]; but more often they use custom structures.

Specializing on paths, the DataGuide [20] handles raw paths and provides the basis for future path indexing methods such as XDG (Extended DataGuide [6]), ViST [42], or Index Fabric [18].

We have a lot of possibilities how to divide the existing methods in categories according to common criteria. Instead of describing all solutions, we decided to focus on three types of solutions that influenced us while designing our indexing concept: using *numbering methods*, *mapping into relational databases*, and transforming into *multi-dimensional tuples*. Later, in Section 3.1, we provide details about how our concept relates to existing solutions.

2.3.1 Numbering Schemes

One of the ways how to discover ancestor-descendant (A-D) relationships in the input XML tree is to assign each node in the graph some numbers during the tree traversal. In our best knowledge, the first approach how to determine these numbers was the work of Dietz [19] in which the author proposed that *for two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after postorder traversal*:

$$x = ancestor(y) \Leftrightarrow pre(x) < pre(y) \wedge post(x) > post(y) \qquad (2.1)$$

So when we assign, during the tree traversal, each node a pair of numbers that will correspond to the preorder and postorder numbers, we can easily determine the A-D relationships in the constant time. The disadvantage is that anytime a node is added to the tree, we must recompute all assigned values.

Many other numbering schemes stem from this method and usually they add other numbers to acquire more features.

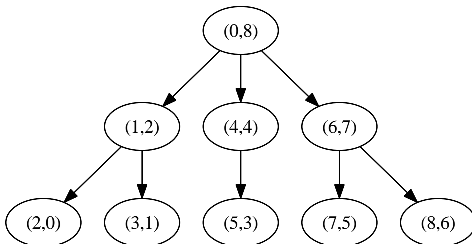


Figure 2.3: Dietz's numbering schema with preorder and postorder numbers

Example 2. *We take a simple XML tree and assign nodes the numbers according to the Dietz's proposition (see Figure 2.3). In this case, the node (0,8) is the ancestor of the node (3,1) because the node (0,8) comes before the node (3,1) in the preorder (0 < 3) and after the node (3,1) in the postorder (8 > 1).*

2.3.2 Structural Joins

The easiest way of evaluating a path query is to divide the path expression into step expressions, to find all elements for each step, and to join these partial results to get the final result. We call the joins *structural joins*, and the result of a join is an unordered set of elements that satisfy the A-D relationships between steps. The concept is similar to joining tables in relational databases where each table corresponds to a partial result (an unordered set of elements).

The advantage is that joining sets is simple and straightforward when we know how to find all nodes that form the partial result. So the solutions that use structural joins such as [23] or [29] focus mostly on how to quickly find these nodes, and provide several algorithms how to evaluate the relationships between them.

The main drawback is that we need to validate the A-D relationship for each pair of nodes and although the partial results might contain numerous nodes, the final result might be only a small subset of these nodes. So we have to test a lot of nodes that do not meet the prior criteria, which is time-consuming and inefficient.

XISS

XML Indexing and Storage System (XISS [29]) is an indexing method based on a numbering scheme and three major index structures (for elements, attributes, and the structure index) combined with several algorithms that process regular path expressions.

The numbering method is inspired by Dietz's numbering scheme. However, instead of preorder and postorder numbers, the authors suggest using an *extended preorder* and a *range for descendants*. Therefore, each node gets two numbers $<order,size>$ as follows.

- For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. It other words, interval $[order(y), order(y) + size(y)]$ is contained in interval $[order(x), order(x) + size(x)]$.
- For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, $order(x) + size(x) < order(y)$.
- For a tree node x, $size(x) \geq \sum_y size(y)$ for all nodes y that are direct children for x.

Evaluating the A-D relationships is still in the constant time because *for two given nodes x and y of a tree T, x is an ancestor of y if and only if*

$$order(x) < order(y) \leq order(x) + size(x) \tag{2.2}$$

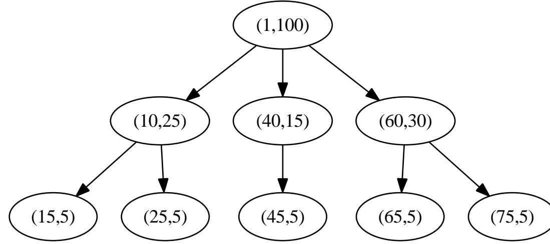


Figure 2.4: Applied XISS numbering scheme

Example 3. *See the sample numbering for a given tree with a proposed numbering scheme in Figure 2.4 and compare it with the Dietz's numbering scheme in Figure 2.3.*

As the complement to the major structures, there is a *name index* implemented as the B^+-tree which stores all distinct name strings. The strings are identified by the unique name identifier (*NID*) which is a key to the *element index* and the *attribute index* that are implemented also as B^+-trees.

When analyzing a complex path expression, authors first decompose it into several simple path expressions (called *subexpressions*) which produce intermediate results and then they combine or join them into the final result. Subexpressions might be a single element or an attribute, an element with an attribute ($\mathcal{EA}$), two elements ($\mathcal{EE}$), a Kleene closure (+,*), or a union of two subexpressions. For each of these situations, a path-join algorithm is proposed:

- **$\mathcal{EA}$-Join**
  This join takes a list of elements and a list of attributes as intermediate results for corresponding subexpressions and join them if they belong to the same document. Both lists are scanned sequentially.

- **$\mathcal{EE}$-Join**
  It joins two lists of elements which are intermediate results for two subexpressions that are in the same document. Both algorithms ($\mathcal{EA}$ and $\mathcal{EE}$) use *two-stage sort-merge* operation - merging according to the common documents in the first step and according to the A-D relationships in the second step.

- **$\mathcal{KC}$-Join**
  The algorithm of Kleene closure is used when a regular path expression contains zero, one, or more occurrences of a subexpression (using + or *). It sequentially applies the $\mathcal{EE}$-Join to the previous result until the join does not produce more results.

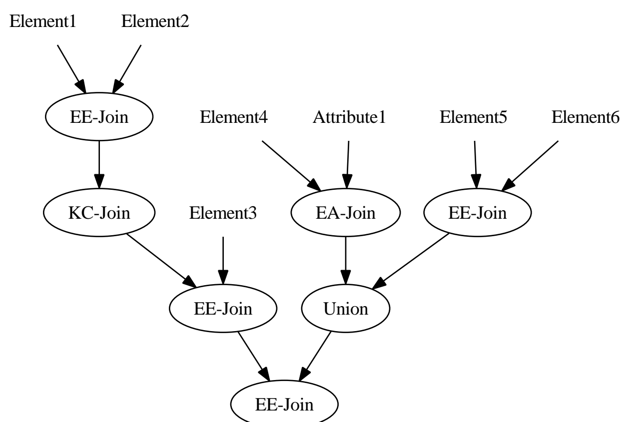


Figure 2.5: The decomposition of a sample path expression into single elements and attributes; and the application of distinct path-join algorithms

Example 4. *If we take a sample path expression, we are able to decompose it into several simple expressions (single elements and attributes). According to the expression types, we apply the desired path-join algorithms to acquire the final result (see Figure 2.5 where the edges determine the data flow).*

The solution brings a new numbering scheme that is more flexible and suitable also for dynamic updates, adding or deleting nodes. Even though authors provide good experimental results, the decomposed path expression might still produce quite extensive intermediate results.

2.3.3 Mapping to Relational Databases

The combination of indexing XML with relational databases seems natural. If we succeed in transforming the given XML into an RDBMS[1], we can leverage all features of databases such as ACID [22] (atomicity, consistency, isolation, and durability).

If we had a schema, we could create corresponding database tables and map the values from the XML file into tables. Then we will use the indices for database tables mostly based on B^+-trees. The problem appears when we have to process files with no schema assigned.

[1] Relational DataBase Management System

GRIPP

GRIPP (GRaph Indexing based on Pre- and Postorder numbering [41]) works exclusively with database tables and expects the input graph to be stored in the database prior the evaluation. It suggests an extended labeling scheme that assigns each node in the graph at least one pair of preorder and postorder numbers (similar to Dietz's numbering schema). The node might have several ancestor because GRIPP handles general graphs. At least one node must be without incoming edges. If such node does not exist, authors create a new virtual root node.

They distinguish tree and non-tree edges and use the *hop technique* to find, recursively, all reachable nodes from a given node in the graph.

The major issue is to choose the order of labeling nodes which has a great impact on the query evaluation. There exists a known solution how to find the best ordering but it relies on finding Hamilton paths in the graph which is NP-complete problem, so it is not feasible for real applications. Therefore authors suggest the heuristics how to compute the good ordering.

XPath Accelerator

The work [21] presents a method designed specially for XPath queries and handling regular path expressions. It encodes parts of the input file into *XML document regions* and assigns each node v a 5-dimensional *descriptor*:

$$desc(v) = \{pre(v), post(v), par(v), att(v), tag(v)\} \tag{2.3}$$

The descriptor consists of preorder $pre(v)$ and postorder $post(v)$ numbers, parent's preorder ranking $par(v)$, the boolean attribute flag $att(v)$, and the element or attribute name $tag(v)$.

The evaluation of XPath queries is based on *query windows* and the inductive generation of the SQL query. For each XPath axis, the concept defines the query window according to the descriptors. Searching for all nodes that belong to a specific query window is translated into an SQL query. The result provide the context nodes for the next step, so the nested SQL query is generated.

Example 5. *If we take a sample XML tree from the Figure 2.6(a) and assign each node preorder and postorder numbers, we can visualize the nodes in the table; see Figure 2.6(b). The context node f determines four disjoint regions:*

1. *the lower-right region contains all* `descendants` *of f*
2. *the upper-left region consists of* `ancestors` *of f*
3. *the lower-left region includes nodes* `preceding` *f in the document order*
4. *the upper-right region represents nodes* `following` *f in the document order*

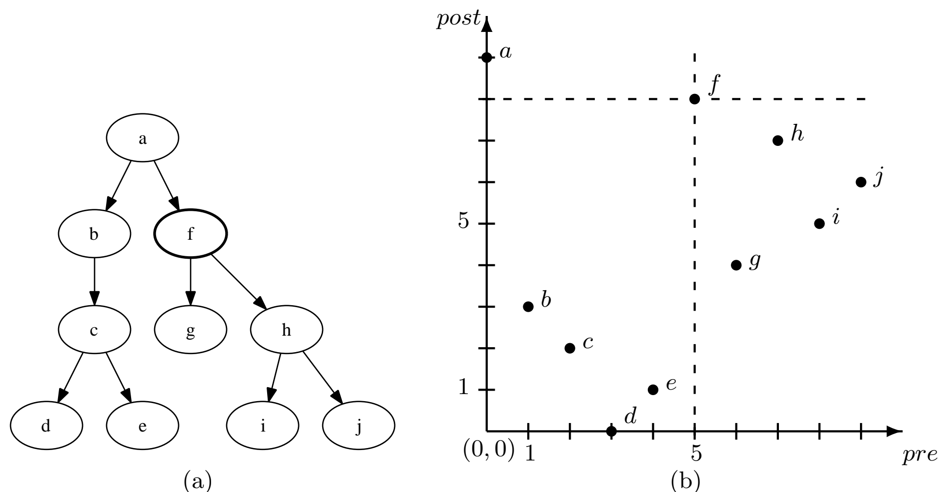


Figure 2.6: A sample XML tree for *XPath accelerator*. (a) Graph-structured data, (b) XML document regions according to preorder and postorder numbers

Because the node descriptors are in the 5-dimensional space, the solution suggests using R-trees [5] as the database indices because they provide good results in XPath step evaluations.

2.3.4 Multi-dimensional Approaches

Focusing on indexing paths, these methods try to transform the arbitrary long paths from the input XML tree to multi-dimensional tuples. The main idea is to avoid structural joins which might be slow and inefficient and leverage the multi-dimensional data structures such as UB-trees [28] or R-trees [5].

Having the labeled paths transformed to points in multi-dimensional spaces, we can use *point* or *range queries* to get the expected result as stated in [25]. Because these queries are a bit special, we need to either adjust the multi-dimensional structures or create new structures.

The benefit of multi-dimensional points is that enabling the update actions such as inserting or deleting is simple.

The main problem is that we do not know the dimension of spaces where we place the multi-dimensional points before parsing a file. One can always generate a file which has the maximal path length longer than expected, so we cannot rely on the upper bound of the path lengths or the number of dimensions. Also not every path has the same length, so we must take into account that the multi-dimensional points might have different numbers of dimensions.

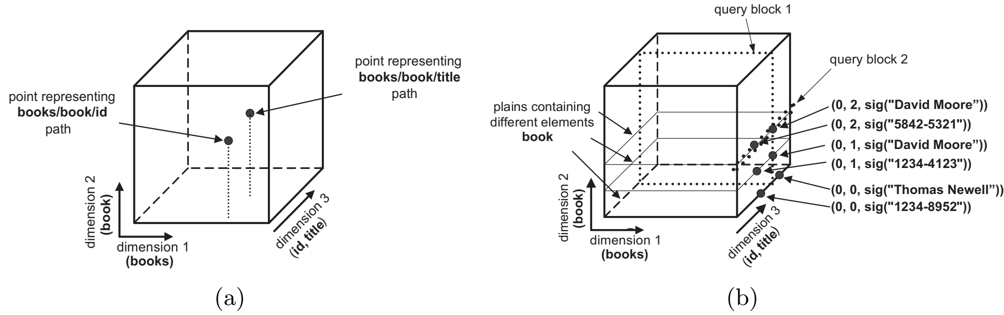


Figure 2.7: The 3-dimensional space with indexed paths in the MDX approach.

Multidimensional Approach to Indexing XML (MDX)

The works of [25] and [26] propose a new indexing method for XML trees based on multi-dimensional tuples without using structural joins, MDX. It models the XML tree as a set of root-to-leaf paths using vectors (based on the work [27]; see Figure 2.7). The method differentiates between *paths* (the sequence of unique node numbers; identified by the unique number `PID`) and *labeled paths* (the sequence of node tags; identified by the unique number `LPID`).

It assigns unique numbers to nodes (node identifiers) according to some numbering scheme. The numbering scheme might be easily adjusted to generate unique node identifiers with *gaps* (e.g. incrementing by 100) to cover the update operations such as inserting new nodes or delete existing nodes without a big effort.

MDX contains three index structures:

1. **Term index** that includes unique numbers $id_T(s_i)$ for terms s_i (names, text values of elements, or attributes). It provides the conversion between these two values and might be implemented as the B-tree.

2. **Labeled path index** contains points, that represent labeled paths, together with corresponding labeled paths' unique numbers (`LPIDs`).

3. **Path index** consists of points that denote all root-to-leaf paths.

Example 6. *Suppose we take a root-to-leaf path that consists of "books/book/title" elements, the corresponding tags are* $\{0, 1, 4\}$*, and the unique labeled path number might be* $LPID = 2$*. So we insert the tuple* $\{1, false, 0, 1, 4\}$ *into the Labeled path index. The first value, docId, denotes the unique number of an XML collection, the second value, attrFlag, is the boolean flag whether the path ends in an attribute or not, and the rest is the list of unique node identifiers. If the node identifiers are* $\{0, 3, 8\}$*, we also generate the tuple* $\{2, 0, 3, 8\}$*, where the first coordinate is the* `LPID`*, and we store it in the Path index.*

The proposed query processing consists of several steps: find unique numbers $id_T(s_i)$ for all terms, find labeled paths' `LPID` (the point query), and search for points in the path index (the range query). Branch queries are divided into path queries and the algorithm applies a *path join*. It is different from the structural join that must be executed for each pair of nodes because authors use it for branch nodes only.

Although the results show faster query execution for sample queries compared to element-based approaches such as XISS, the evaluation of complex queries with relative paths might still force to full search of several whole dimensions for resulting multi-dimensional points.

Indexing XML Data with UB-trees

Universal B-trees (UB-trees) combine the *Z-ordering* and the B-tree which is the balanced, persistent, and paging tree, into a structure that accesses multi-dimensional data. If we have the n-dimensional space, we can compute the Z-address for any tuple (see Figure 2.8). The UB-trees then define the *Z-regions* which make clusters of spatial neighbors and which provide a totally ordered disjunctive partitioning of the given space.

This method takes paths from the root to the leaf and compute the *path content* for each path $e_1/e_2/\ldots/e_k$ as a sequence of string values $s_1/s_2/\ldots/s_k$. The string values are further transformed into *signatures*, the binary numbers of the same lengths, so we get the n-dimensional point that represents the path content $(sig(s_1), sig(s_2), \ldots, sig(s_k), \ldots)$ where $sig(s_i)$ is the signature of s_i for $1 \leq i \leq k$, or $sig(s_i) = 0$ for $n > k$. Then we insert these points into the UB-tree.

Before the evaluation, the given query is transformed to a *range query* which finds all points that includes the n-dimensional *query cube* (for 2-dimensional space it means the rectangle). Authors propose *query window* as its name.

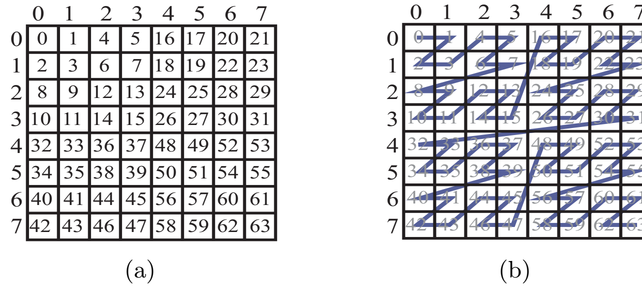


Figure 2.8: Sample Z-ordering (a) Z-addresses for all points in 2-dimensional space 8 x 8, (b) The Z-curve that fills the entire 2-dimensional space 8 x 8

When indexing the paths, the n-dimensional point is similar but the dimension of the space will be $n + 1$ because we add the unique path number to the coordinates.

The work [28] showed that although the method is correct and viable, inserting and finding points in UB-trees was not so demanding.

Chapter 3

iXUPT: Indexing

In this section, we outline the new proposed indexing method *iXUPT* (Indexing XML Using Path Templates), we explain how it differs from existing solutions, and how it enhances present techniques, followed by the detailed description of how to build and initialize the index structures. Finally, because we offer two variations of our concept, we will compare them and depict the differences.

3.1 Overview

The aim of our proposed method is to build a native XML indexing scheme which improves the evaluation of XPath queries in terms of the time complexity. We follow the multi-dimensional techniques and leverage the path-based indexing with focus on grouping paths according to common characteristics, *path labels*. We expect this idea to eliminate many unsuitable paths and, as the result, to speed up the evaluation of any query.

The interesting work of mapping all root-to-leaf paths into multi-dimensional points, MDX [26], influenced us on designing our indexing method. Similar as in this work, we take the root-to-leaf paths and label them. MDX tries to avoid using structural joins because of their time complexity and so does our solution. We also use joins only for branch queries.

We found also motivation in XISS [29] for the decomposition of XPath expressions into simpler subexpressions and producing the intermediate results (called candidates in our approach) that might be combined or joined to produce the final result. Although the inspiration came from structural joins, we try to eliminate such joins that does not add anything to the final result in order to accelerate the querying. The method of *path labeling* helps us with doing so because we make joins only on similar paths.

We understand the inefficiency of element-based approaches with structural

joins but we also see the potential slowdown for the multi-dimensional mapping methods when evaluating queries with multiple wildcards and relative paths. In this case, the domain used for finding matching tuples might grow faster. Therefore we focus on these special types of queries as well.

The Patricia trie data structure [24] inspired us while building the `PrefixTree` (see Section 3.3).

Our approach considers only tree structures as the input. We also exclude attributes and text values from the indexing and querying methods and focus primarily on the elements and their relationships. The support for indexing and evaluating attributes is straightforward because any attribute of an element might be considered as a specific subelement with a specific edge. There will be no big difference in querying as shown in the MDX method which handles also the attributes.

3.2 Indexing Paths

When we get the input XML file, we need to parse the file, initialize the structures and build the initial graph. We use the SAX (Simple API for XML) parsing method. In this case, this method is more effective than the DOM (Document Object Model) approach because we do not need to build the whole representation of the XML file. We take only information and the data we need and store them in our own structures. Because we process the file as a stream, we fill the structures on-the-fly while parsing the file.

First of all, we assign elements in the source file the unique identification numbers (`NodeIDs`) and convert the element (tag) names into integers (`TagIDs`). We use the numbering method that assigns a node the `NodeID` when the corresponding element is visited for the first time. In terms of the tree traversing, we use the in-order traversal sequence. The root element has the number 0, while the numbers of following nodes are always incremented by 1 (see the sample XML file with element names and `NodeIDs` in Figure 2.2).

Because element names might repeat, we create the `TagID` only for the first occurrence of the specific name. The next time we visit a node with the same element name, we use the previously created `TagID`. Therefore multiple `NodeIDs` might have the same `TagID`. Similar to `NodeIDs`, the first element name has the number 0, while the next unique element name is always incremented by 1, and the conversion between element names and corresponding `TagIDs` might be implemented with B-trees.

We prefer numbers over strings because we need to compare the element names while evaluating and the comparison of integers is faster than comparing strings with variable lengths even though it brings some memory overhead.

Table 3.1: Terms definition and description

Term	Description
`NodeID`	The unique node number given by the numbering method.
`TagName`	The name of an element.
`TagID`	The unique number for the `TagName` given by the numbering method.
`Path Label`	The sequence of `TagIDs` of the nodes on the path from the root to the leaf.
`Path Prefix`	The sequence of `TagIDs` that identifies a prefix of at least one path.
`Path Template ID` (`PTID`)	The unique number for a path label. The path labels are converted into integers (`PTIDs`).
`Path Reference`	The sequence of `NodeIDs` of the nodes on the path from the root to the leaf.

Next, we store all root-to-leaf paths according to their *path labels*. We determine the path labels by the sequence of `TagIDs` of the nodes on the path from the root. Whenever we reach a leaf node, a new root-to-leaf path occurs, and we store the `Path reference` according to its path label. The `Path reference` contains the sequence of `NodeIDs` of the nodes on the path from the root to the current (leaf) node and it is unique.

The difference between the `Path reference` and the *path label* is that the first one contains `NodeIDs` while the second one uses `TagIDs`. The path label is stored only once but one path label can contain several `Path references`. Moreover, we convert each path label into corresponding *Path Template ID* (`PTID`) that groups similar `Path references` together.

For the comprehension, we provide the list of new terms with their descriptions in Table 3.1.

Example 7. *If we take the sample XML file from the Figure 2.2 and we visit a leaf node* fax (13)*, the* `Path reference` *(sequence of* `NodeIDs`*) will be* `(0,7,8,13)`*. Converting element names* `"/faculty/department/contact/fax"` *to* `TagIDs`*, we get the path label '*`/0/7/1/9`*'. For details about the conversion, see Section 3.3 and Table 3.2 (*`NodeTags` *table).*

3.3 Structures

We maintain several in-memory structures that store the root-to-leaf paths and other necessary information that we will later use when we evaluate queries. There are three major data structures: the `NodeTags` table, the `Paths` table, and the `PrefixTree`. While the first two structures are more like database tables, the last one is definitely a tree structure.

Figure 3.1 shows the overview of the iXUPT Index and includes also temporal structures for query evaluation (`XPathTree`; see Section 4.2.1).

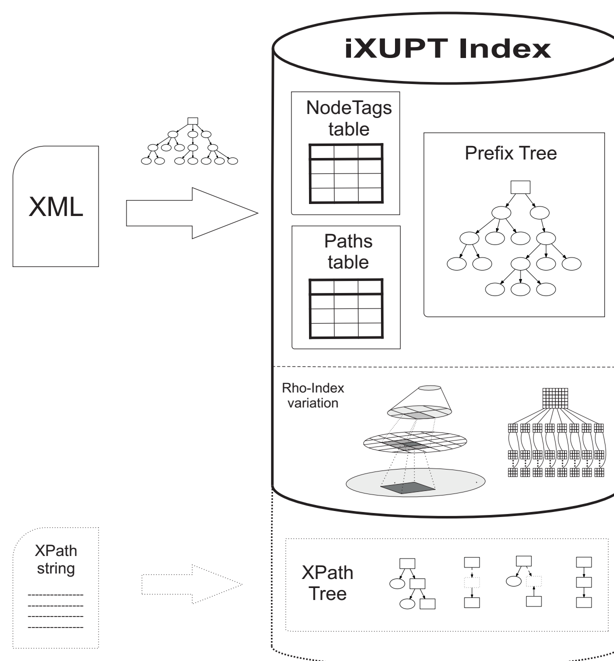


Figure 3.1: Overview of the *iXUPT* Index and its data structures

3.3.1 NodeTags Table

This table provides the conversion between the `TagName` and the `TagID`. The `TagNames` are listed according to their first occurrence in the input XML file. We save also all Path Template IDs (`PTIDs`) where the given `TagID` appear; see Table 3.2. As we assign the `TagIDs` only to visited nodes, there must be always at least one `PTID` for a given `TagID` because each node is included in at least one root-to-leaf path.

3.3.2 Paths Table

This table contains the conversion between the *path label* and corresponding `PTID`. We delimit the `TagIDs` in the *path label* with the '`/`' character similarly to the directory navigation in common file systems. As mentioned before, a single *path label* provides grouping for several `Path references` that we also store here; see Table 3.3.

Table 3.2: *NodeTags* table for the sample XML in Figure 2.2

TagName	TagID	Path Template IDs
faculty	0	{0,1,2,3,4,5,6,7,8,9}
contact	1	{0,1,2,3,4,5,6,7,9}
address	2	{0,1,4,5,6}
street	3	{0,4}
city	4	{1,5}
email	5	{2,9}
phone	6	{3}
department	7	{4,5,6,7,8,9}
zip	8	{6}
fax	9	{7}

Table 3.3: *Paths* table for the sample XML in Figure 2.2

PTID	Path label	Path references
0	0/1/2/3	{(0,1,2,3)}
1	0/1/2/4	{(0,1,2,4)}
2	0/1/5	{(0,1,5)}
3	0/1/6	{(0,1,6)}
4	0/7/1/2/3	{(0,7,8,9,10)}
5	0/7/1/2/4	{(0,7,8,9,11); (0,15,16,17,18)}
6	0/7/1/2/8	{(0,7,8,9,12)}
7	0/7/1/9	{(0,7,8,13); (0,15,16,19)}
8	0/7	{(0,14)}
9	0/7/1/5	{(0,15,16,20)}

3.3.3 Prefix Tree

The `PrefixTree` covers all distinct *prefixes of path labels* from the input file. Each node in this tree has the `TagID` and the path from the root to a given node represents the path prefix. Nodes also contain the list of all `PTIDs` which do have the selected prefix. We use this structure to find all `PTIDs` to which a `NodeID` might belong. Because a given `NodeID` determines its path from the root (the prefix of a path label), we can use the sequence of `TagIDs` of nodes on this path for the navigation in the `PrefixTree` and we get all possible `PTIDs` for this node (for details about using this method within the query evaluation see Section 4).

Example 8. *If we take the node* contact (1) *from the sample XML (see Figure 2.2), it belongs, according to the* `NodeTags` *table, to almost all* `PTIDs`*. And using the* `PrefixTree`*, we find its prefix* `'/0/1'` *that matches the* `PTIDs` *{0,1,2,3}. So the chosen node occurs only on paths with these path labels.*

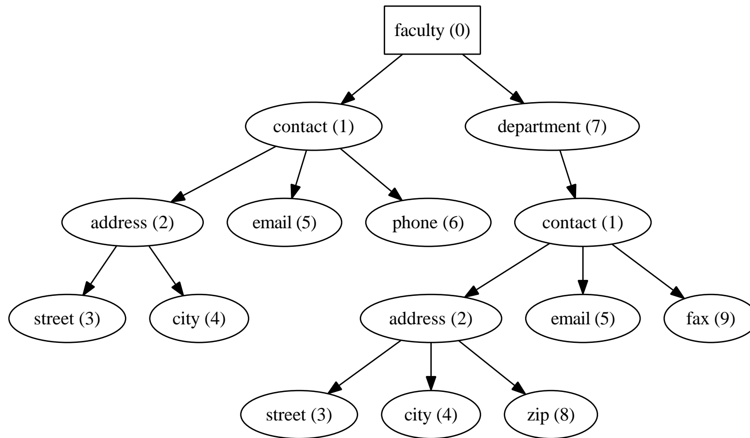


Figure 3.2: *PrefixTree* for the sample XML file in Figure 2.2

3.4 Build Index

In previous sections, we introduced our own data structures that we will use while indexing the XML files. To demonstrate how we initialize, build, and use the data structures, we provide a simple algorithm which describes the main steps and actions in the process of parsing the source file (see Algorithm 1).

After initializing our data structures, and creating a SAX reader, we read sequentially the whole file. We expect the XML file to be well-formed, otherwise an exception is thrown and the parsing fails. A document is well-formed if it meets several constraints (declared in [15]) such as it contains only a single root element, the start tag and its corresponding end tag are in the content of the same element, and many others.

We handle two events when traversing the input XML file: when we visit a new element (*start element*) and when we leave an element (*end element*). All other events are skipped. Depending on the type of the event, we execute several actions that we explain in the following text. We do not consider attributes or text values as we stated at the beginning.

3.4.1 Start Element

When we visit an element for the first time, we need to assign the element the unique identification number (`NodeID`) and we add the element name into the `NodeTags` table. The table assures that the element name is appended and gets a new `TagID` only if it is not present in the table yet. Otherwise we get the `TagID` for an existing item from the table.

Algorithm 1 Build iXUPT Index algorithm

Require: *path* to the source XML file
Ensure: Initialized iXUPT index

```
 1: nodeID = 0
 2: NodeTags ← new NodeTagsTable()
 3: Paths ← new PathsTable()
 4: PrefixTree ← new PrefixTree()
 5: reader ← new SAXReader(path)
 6: while reader.Read() do
 7:    if reader.IsStartElement then {first visit}
 8:       NodeTags.Add(reader.ElementName)
 9:       tagID = NodeTags.GetTagID(reader.ElementName)
10:       pathRefStack.Push(nodeID)
11:       pathLabelStack.Push(tagID)
12:       nodeID = nodeID + 1
13:       isLeafNode = true
14:    else if reader.IsEndElement then {last visit}
15:       if isLeafNode then
16:          pathLabel = GetPathLabel(pathLabelStack)
17:          if not Paths.Contains(pathLabel) then
18:             Paths.Add(pathLabel)
19:             PTID = Paths.GetPTID(pathLabel)
20:             for all tagID in pathLabelStack do
21:                NodeTags[tagID].AddPTID(PTID)
22:             end for
23:             PrefixTree.Process(pathLabelStack, PTID)
24:          end if
25:          Paths.AddPathReference(pathLabel, pathRefStack)
26:       end if
27:       pathLabelStack.Pop()
28:       pathRefStack.Pop()
29:       isLeafNode = false
30:    end if
31: end while
```

We save the `NodeIDs` in the stack (*PathRefStack*) and whenever we visit a leaf node, the stack represents a `Path Reference` that we consequently add to the `Paths` table. Similarly, saving `TagIDs` in another stack (*PathLabelStack*) provides a basis for creating the *path label*.

The boolean variable `isLeafNode` determines whether the path from the root to the current node is a root-to-leaf path. When we traverse the tree downwards (similar to Depth First Search), any new node that we visit might be a leaf node, therefore we assign it `true`. When we visit a non-leaf node on the way back while backtracking, the value will be `false`. It is assured by the *End Element* method.

We omit some actions such as maintaining the current `NodeID`, creating the initial graph (if we use the Rho-Index), or handling the array of already processed elements with corresponding `TagIDs`. These actions are obvious and they depend mainly on the chosen implementation.

3.4.2 End Element

This event represents the last visit of a node in the graph within the traversal before backtracking. If a node is a leaf node, a new root-to-leaf path occurs and we need to handle it. We create the *path label* from the `TagIDs` in the stack and check whether it exists in the `Paths` table. If the *path label* is not present, we add it to the table and it gets a corresponding `PTID` (Path Template ID). Furthermore, we need to update the `NodeTags` table and for each `TagID`, we append this new `PTID` to the existing list of `PTIDs`.

We also have to update the `PrefixTree` with the new `PTID` and add it to all corresponding nodes. `PrefixTree` ensures that if any node does not exist, it will be created.

No matter whether the *path label* occurred in the `Paths` table before or not, we store a new `Path Reference` (sequence of `NodeIDs`) to the corresponding item in the `Paths` table determined by the *path label*.

Because there are no more childnodes and we leave the node, we remove the current `NodeID` and `TagID` from the stack because we will not use them in the future.

Finally, we assign the `isLeafNode` variable the `false` value because we backtrack from the leaf node, so other nodes on the way back to the root cannot be leaves. A new leaf might occur only if a node has not been visited yet, and we handle this situation with the previous phase (*Start Element*) when we visit the node for the first time.

3.5 Variations

As we stated at the beginning, we provide two variations of our new concept of indexing XML files. Both of them provide a different approach to handle the Ancestor-Descendant relationships when evaluating queries. They are interchangeable and fully comparable, when we talk about the functionality.

The first variation is based on a new numbering schema while the second one uses the *Rho-Index*, the indexing structure that is primarily used for processing path queries and discovering relationships between elements. If we choose any variation, the structures described in previous sections will remain the same.

3.5.1 Numbering Scheme

When using the numbering scheme for revealing A-D relationships, we need to add one more number to each node in the initial graph. We mark it as `LastNodeID` and it is the `NodeID` of the last visited leaf node in the subtree determined by the current node. We assign the number when we visit a node for the last time during the tree traversal.

So we get the node interval (`NodeID,LastNodeID`) for each node in the tree. This interval is similar to Dietz's preorder and postorder numbers and it covers all subelements. Precisely, the node x is an ancestor of the node y if and only if the $y.NodeID$ belongs to the node interval of the node x:

$$(y.NodeID > x.NodeID) \wedge (y.NodeID \leq x.LastNodeID) \tag{3.1}$$

Remark that using node intervals, we are able to compute the A-D relationships for any two nodes in the tree in the constant time.

3.5.2 Rho-Index Structure

Rho-Index (also ρ-index [2], [3]) is a multi-level balanced tree structure that was created to handle special path queries in graph structured data. When we take a general graph, that comprises of vertices and edges, we can use this index to find all paths between arbitrary two vertices in the graph[1].

As the input, Rho-Index expects an existing graph, that, in our case, we create while parsing the XML file. Then we need to compute required values before the structure is ready to use. The main idea is to take the initial graph and transform it to a new simplified graph with similar properties. Authors call this transformation *graph segmentation* and the product of the transformation is the *Segment graph*.

Furthermore, they use a modification of the adjacency matrix to describe any node in the tree structure. This modified Path Type Adjacency Matrix (PTAM) stores the whole paths between two nodes instead of path counts that are saved in the usual adjacency matrix. Together with customized matrix operations, $+$ and $*$, we can compute the transitive closure of this matrix. As the result, we get a matrix that contains at each position $[i, j]$ all paths between the node i and the node j. Because path lengths in a graph might be arbitrary, computing the transitive closure might be time-consuming. Therefore the index expects the maximal path length as the input parameter in order to limit path lengths. This bound increases the efficiency because it also determines the number of iterations when we compute the transitive closure of the modified adjacency matrix.

[1] Although the Rho-index allows several other special path queries, we will use the structure only to search for all paths between any two vertices in the graph to check A-D relationships.

Algorithm 2 Create Rho-Index algorithm

Require: $G = (V, E)$, minSegmentSizes, maxSegmentSizes, maximal length L
Ensure: Rho-Index for the initial graph G
1: $CG \leftarrow G$ {CG is the Current Graph being processed}
2: **for** $i = 1$ to $SegmentSizes$ **do**
3: S = CreateSegments(CG,$minSegmentSizes[i]$,$maxSegmentSizes[i]$)
4: SG = CreateSegmentGraph(S)
5: **for all** ($segment$ in S) **do**
6: $segment$.CreatePTAM() {Path Type Adjacency Matrix}
7: $segment$.ComputeTransitiveClosure(L)
8: **end for**
9: $CG \leftarrow SG$ {The segmentation graph becomes the current graph}
10: **end for**
11: CG.CreatePTAM()
12: CG.ComputeTransitiveClosure(L)

When we create the index (see Algorithm 2), we are able to search for all paths between any two vertices. The searching uses special *transcription graph* with specific edge types that initially contains only the start and the end vertex (as the input). While processing this graph, we transform the edges until we get only edges of one particular type. At that time, the transcription graph contains all paths between those two input vertices. We cover the details of the search algorithm later in this section.

Graph Segmentations

The graph segmentation divides all vertices from the input graph into several segments where each segment contains at least one vertex and each vertex belongs to a single segment. Once we assign a vertex to a segment, it will remain in the segment with no further changes. The difference between a segment and a subgraph is that a segment can contain also an edge defined by two vertices possibly in different segments.

When we get a set of segments, we connect the segments with edges applying the rule that two segments S_1, S_2 are connected with an edge whenever there exists two vertices v_1, v_2 such that $v_1 \in S_1$ and $v_2 \in S_2$. Because there might be multiple edges, we can replace them with a single edge and save the multiple edges in special *Tables of Transitions* between segments (based on the inverted file index). We also discard the inner edges within the segments. Then, if we take segments as the vertices and edges that connects segments as edges, we get a new *segment graph* $S(G)$ that has similar properties as the initial graph G but

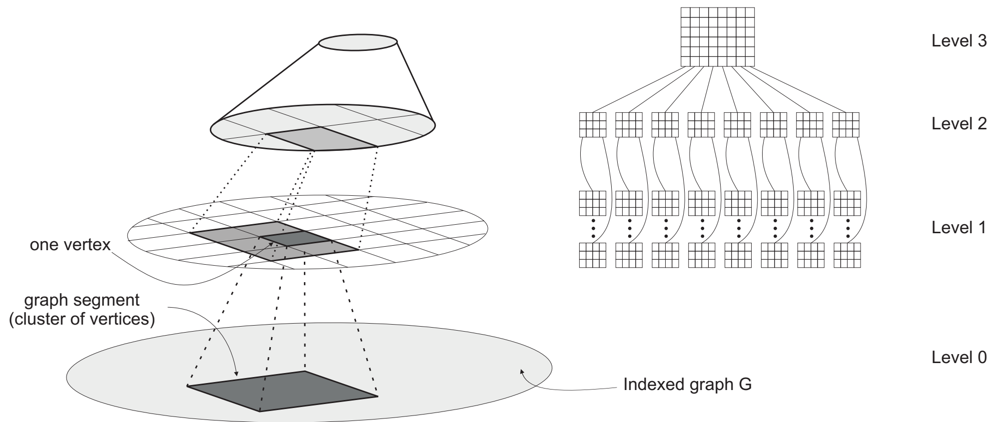


Figure 3.3: *Rho-Index* - Graph segmentations

it is smaller (see Figure 3.3). Furthermore, if there exists a path in an initial graph, the path will remain in the segment graph but will be simplified. Instead of vertices and edges from the initial graph, the path in the segment graph consists of segments and edges between them and it is called the *sequence of segments*.

We create the path type adjacency matrix (`PTAM`) for each segment (line 6 in Algorithm 2). Because segments have a smaller number of vertices, computing the transitive closure for each matrix will be faster than computing the transitive closure for a matrix covering the whole graph. The transitions between segments will be represented by the graph at the higher level.

The previous steps describe only one iteration of the graph segmentation which takes the initial graph G and transforms it to the segment graph $S(G)$. Because the resulting graph might be still large, we apply the same process of graph segmentation to the segment graph $S(G)$ and we get another segment graph $S(S(G))$. The Figure 3.3 visualizes the graph segmentation method.

Number of graph segmentation iterations is another input parameter together with minimal (`minSegmentSizes`) and maximal (`maxSegmentSizes`) number of vertices in one segment. These bounds might differ for each iteration.

There are several segmentation methods available that divide vertices into segments such as graph to forest of trees with or without vertex clustering, or the segmentation using the topological order. The selection of a segmentation method has an impact on the creation complexity of the Rho-Index. We implemented the second method that uses the topological order because the input graph will be a DAG (directed acyclic graph) as we focus on indexing trees only.

Algorithm 3 Rho-Index search algorithm

Require: starting vertex s, ending vertex e

Ensure: Graph of all paths between s and e stored in Rho-Index

1: Get segments $[S_1, S_2, \ldots S_k]$ for s
2: Get segments $[E_1, E_2, \ldots E_k]$ for e
3: TG = BuildInitialTranscriptionGraph()
4: TG.TransformEdges()

Search Algorithm

After we process the initial graph and create several segment graphs, we are ready to search for all paths between any two vertices using the transcription graph. At any time, the transcription graph is a DAG. The search algorithm (Algorithm 3) leverages the principle of the segmentation that each vertex on the higher level corresponds to a graph on a lower level.

Initially the transcription graph contains the start vertex s and a list of segments $S_1, S_2, \ldots S_k$ to which this vertex belongs on higher levels (k denotes number of segmentation iterations or levels). Then, we append the end vertex e with its list of segments $E_1, E_2, \ldots E_k$ together with an entry from the top-most path type adjacency matrix $P[S_k, E_k]$.

There are four different edge types in the transcription graph and each edge type is determined by the level at which the two connected nodes occur. The search algorithm transforms these edge types, so we get only vertices at the lowest level (vertices from the initial graph) that might be connected with edges (also from the initial graph). If the resulting transcription graph is connected, it contains all paths between vertices s and e with the limited path lengths. Otherwise, it includes only the start and the end vertex and no edge. In this case, no path between specified vertices exists.

3.6 Implementation Details

We develop the indexing technique that is completely stored in the main memory. The advantage is that we can easily and relatively fast access the data we need but the drawback is that it has higher memory requirements, especially for larger files.

We understand the temporal inefficiency of loading the whole structures to the main memory, and we propose to study this issue as the future work (see Section 6.1). We see the perspective of saving the data structures to the secondary memory for both variations. Considering the numbering scheme, we need to additionally store only a single integer (`LastNodeID`), and for Rho-Index, we might use

some of the persistent memory structures designed for tree structures. Other data structures such as `Paths` table or `NodeTags` table must be studied more properly to create a serialization method that takes into account variable sizes of some items.

Chapter 4

iXUPT: Evaluating XPath Queries

When we build and initialize the *iXUPT* index for a specific XML file, we are able to use the defined structures for evaluating queries. Our algorithm supports not only regular path expressions but also the branch queries. To provide a standardized way of defining queries, we choose the W3C recommendation of the XPath language [13] as the query description language that is capable of representing any query we need.

The process of evaluating a given XPath query has several steps. In the following sections, we describe the concept of evaluating and then we explain all these actions in more detail.

4.1 Evaluating Queries

Evaluating an XPath query is divided into a number of subsequent steps (see Algorithm 4). First we need to parse and prepare the query. In this step, we convert the string expression into the appropriate tree structure. Then we prepare and initialize the created tree structure for the evaluation.

As the most important step, we evaluate the query by traversing the tree structure in correct order.

The last step gets the result node and provides the final result as the set of nodes in the candidates table for this node.

Any of these steps might produce an exception for unexpected values or incorrect data. If something goes wrong, we display appropriate error message and abort the evaluation. In order to get the final results, all steps have to finish successfully.

Algorithm 4 Evaluate XPath query algorithm

Require: XPath string expression *expr*, iXUPT index $iXUPT$

1: XPathTree *treeQuery* = XPathAnalyzer.Parse(*expr*);
2: PreProcessVisitor.Visit(*treeQuery*, $iXUPT$)
3: EvaluateVisitor.Visit(*treeQuery*, $iXUPT$);
4: XPathNode *resultNode* = EvaluateVisitor.GetResultNode();
5: **return** $resultNode$.Candidates

4.2 Parsing XPath Expressions

In the beginning, we take an XPath string, that includes a regular path expression, and convert it into the structure that is appropriate for the evaluation. We selected our own tree structure `XPathTree`.

The implemented prototype which proves the feasibility of our indexing concept supports only some of the XPath axes: `self`, `child`, `descendant`, `parent`, `ancestor`, `descendant-or-self`, and `ancestor-or-self`. Other XPath axes[1] might be studied as the future work.

4.2.1 XPathTree

`XPathTree` is our own tree structure implemented with the *Composite* design pattern [37] and created by two types of nodes (`XPathNodes`): *steps* and *predicates*. Although they have different meanings and we evaluate them in different ways, considering the data structures, they are implemented in the same way and distinguished only by the boolean flag. This structure is temporal for evaluating the query and we create it on-demand for a given XPath expression.

The root node is the `XPathNode` that corresponds to the first step expression.

Steps If we divide the path expression into the sequence of step expressions, they will be represented by these nodes. We handle the *Step* nodes as the major nodes during the evaluation. Each node has at most one *Step* childnode and we call it `NextStepNode`.

Predicates We express the further filter expressions by the *Predicate* nodes. While the *Step* nodes cannot be added as childnodes to *Predicate* nodes, each node in the `XPathTree` might contain one or more *Predicate* childnodes. If a node has more predicates as childnodes, we combine them using *logical conjunction* (AND). It means that all the predicates must have a non-empty resulting set of nodes in order to meet the filtering criteria of the given node.

[1] For the whole list of existing XPath axes see the Table 2.1

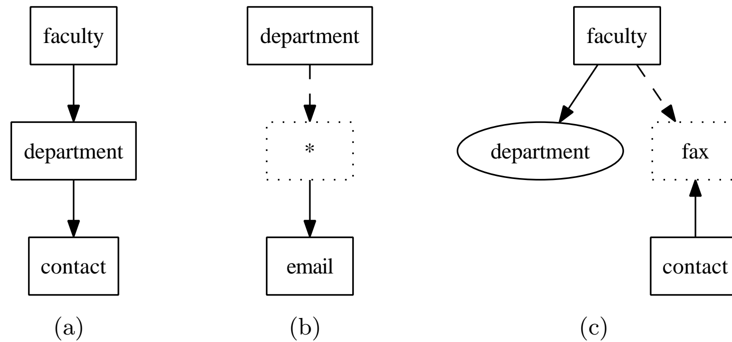


Figure 4.1: The visualized *XPathTree* structure of sample queries. The *Steps* are represented by boxes, while the *Predicates* are shown as ellipses. We use oriented edges to determine the forward and reverse axes (the edge leads always from an ancestor to a descendant). We put dashed edges and dotted boxes or ellipses to indicate that the target node is generally a descendant (not necessarily a childnode). The corresponding XPath expressions are: (a) `faculty/department/contact`, (b) `department//*/email`, and (c) `faculty[department]//fax/ancestor::contact`

Example 9. *For better imagination, we can visualize the* `XPathTree` *that we derive from an XPath expression (see Figure 4.1).*

4.2.2 XPathNodes

Each node in the `XPathTree` contains several values:

- `TagName` The element name in the step or predicate expression assigned during parsing the string expression.
- `TagID` The corresponding unique number for the element name if it exists in the `NodeTags` table; otherwise the value is undefined (e.g. -100). We also need a special value for `TagID` which represents the wildcard expression '$*$' if it is used as the element name (e.g. -3). The value is assigned during the pre-processing phase (see Section 4.3).
- `Order` The integer that corresponds to the order of evaluating the major nodes (applicable only for *Step* nodes).
- `IsMajor` The boolean flag that is true for *Step* nodes and false for *Predicates*.

- `Axis` The description of the XPath axis. Because the `XPathNodes` are in the tree structure, each node has exactly one parent (except for the root). The *Axis* defines the relation between a node and its parent in the `XPathTree` and we use it as a designator when validating the A-D relationships.

Definition 1 (*Alternating XPathNode*)**.**
Alternating XPathNode is a node that *changes* the axis direction. It means that the incoming edge has different direction than the outgoing edge (*forward-reverse* or *reverse-forward*)[2].

4.3 XPathTree Pre-processing

After we parse the XPath expression and create the corresponding `XPathTree`, we need to convert the stored element names into `TagIDs`. While converting, we check whether the element names exist in the index. If a name does not occur in the `NodeTags` table, the result is instantly available because no nodes will be in the result set and no further evaluation is needed (as mentioned before, we suppose all predicates to co-exist at the same time).

During the tree traversal, we also assign `XPathNodes` the integer `Order`, so we know the order in which they are visited and evaluated. We will use this number later while traversing the `XPathTree` to determine whether the minimal set of `PTIDs` for a given node has been already computed or not.

As the result, we get the validated and properly initialized `XPathTree` that is ready for the evaluation.

4.4 XPathTree Evaluating

When we build and validate the `XPathTree`, we can start evaluating a given query by traversing the tree structure. As we outlined before, we use slightly different evaluating methods according to the type of the current `XPathNode`. We evaluate the *Step* nodes with the top-down approach (from the first to the last *Step* node according to the `Order`), while the *Predicate* nodes are processed in the bottom-up style (from the lowest level upwards using backtracking in the Depth-First Search). The reason for distinct methods is that we have to evaluate step expressions in the same order as they occur in the query because the intermediate result from one step defines the context for the next step. On the other hand, all predicates must be resolved before we can continue with the next `XPathNode`.

[2] For details see the types of the XPath axis in Table 2.1

Algorithm 5 Visit procedure for evaluating an *XPathNode* in the *XPathTree*

Require: *xnode* is the current *XPathNode* in the *XPathTree* being visited
1: xnode.Candidates = GetCandidates()
2: **for all** (*predicate* in *xnode.Predicates*) **do**
3: VISIT(predicate) {recursive call for a predicate}
4: xnode.VoteForCandidateNodes(predicate.Candidates)
5: **end for**
6: **if** xnode.HasPredicates **then**
7: xnode.FilterCandidateNodes()
8: **end if**
9: **if** (xnode.IsMajor) **then** {it is the *Step* node}
10: MergeCandidates (lastNode, xnode)
11: lastNode ← xnode
12: **if** (xnode.NextStepNode is not null) **then**
13: VISIT(xnode.NextStepNode)
14: **end if**
15: **end if**

Even though we use the general tree traversal, we are able to apply different approaches for distinct node types. We evaluate *Step* nodes when we first visit them but we process *Predicate* nodes when we leave them.

The Algorithm 5 describes the VISIT procedure that we apply to all `XPathNodes` as we visit them. There are several steps that we need to explain in more detail but the main principle is that we save candidate nodes (`NodeIDs`) for each `XPathNode` we visit, process the candidates if needed, merge the candidates with the previous node if any, and continue with the next node.

The set of candidate nodes of the last *Step* node represents the final result.

4.4.1 Get and Save Candidates

The first step is to get and save candidate nodes for the current node (line 1). We store them in the table called *candidates*. It contains `PTIDs`, where the current `XPathNode` occur, with corresponding `NodeIDs`. Each `PTID` is stored only once but one `NodeID` might be saved for more `PTIDs`. It is because we focus later on merging *candidates* according to `PTIDs` rather than `NodeIDs`.

Get the Minimal PTID Set

To identify the set of `PTIDs` to be stored in the candidates table, we try to find the smallest `PTID` set that is common for as many nodes as possible (using the `NodeTags` table). For a *predicate* node, this means to take only `PTIDs` that are

common for `XPathNodes` on the path from the current *Predicate* node to the first *Step* node on the path from the current node to the root. Because we process these nodes as we leave them (bottom-up), we might pre-compute the smallest `PTID` set on the way down.

For a *Step* node we take the path from the first *Step* node. This holds only if all axis directions on the path are the same; otherwise we take the last alternating node instead. We try to find the `PTID` set for as many *Step* nodes as possible at a time. It means that we "look forward" (process the following *Step* nodes) and store the maximal `Order` number for which the current set is valid. Then, when we visit a node, its `Order` determines whether we have to compute the `PTID` set or not.

If the `XPathTree` does not contain any alternating nodes, the minimal `PTID` set is computed for all *Step* nodes at once. If we have an alternating node, it divides the `XPathNodes` on the path from the first *Step* node into two groups for which we have to compute the smallest `PTID` set separately.

Another option is to compute the minimal `PTID` set for each node in the preprocessing phase which might be studied as a part of the future work. In our concept, we prefer computing it on-demand when it is really needed, even though it might slow down the evaluating.

Example 10. *If we take the Query 3 in Figure 4.1(c), the alternating* `XPathNode` *is the* `fax` *node. Therefore the minimal* `PTID` *subset can be computed only for set of nodes* {`faculty`,`fax`} *and* {`fax`,`contact`}*. The candidates for this* `XPathNode` *are shown in Table 4.1.*

Table 4.1: Candidates table for the *fax* node in Figure 4.1(c)

PTID	**NodeIDs**
2	{5}
9	{20}

Get Candidates

When we obtain the minimal `PTID` set, we use the `Paths` table to find the candidate nodes (`NodeIDs`) according to the `Path references` for any `PTID`. If the `TagID` does not represent `'*'` (wildcard), we find the positions of the `TagID` in the `Path label` identified by the current `PTID`. We search only for positions that are either *after* (forward axes) or *before* (reverse axes) the position of the last node reflecting the minimal and maximal skipping bounds (explained later) of the previous axis. We take all `NodeIDs` on those positions from the `Path references` and store them to the Candidates table for the specific `PTID`.

If the `TagID` reflects `'*'` and the `XPathNode` does not have any predicates, we skip it and save the minimal and the maximal number of positions to be skipped when searching for positions in the next `XPathNode`. The numbers are determined by the current axis:

- $(1, 1)$ for the direct relative (parent, child)
- $(1, \infty)$ for other axes (ancestor, descendant)

We cannot skip the node if it has predicates because the predicates eliminate some nodes, so skipping the node might produce an incorrect result.

4.4.2 Evaluate Predicates and Voting

If there are any predicates for the current `XPathNode`, we need to handle them before we continue with the next node. Because predicates give additional filtering criteria, not all candidate nodes from the current `XPathNode`'s candidates will meet the new criteria.

Therefore we use the *voting* mechanism to eliminate such candidate nodes that do not meet filtering options (line 4). Every predicate node gives a *vote* for all candidate nodes (`NodeIDs`) in the current `XPathNode`'s candidates table (no matter of their `PTIDs`) that are reachable from a `NodeID` stored in the predicate's candidates. The reachability is dedicated from the axis type of the predicate node and the given `NodeIDs`.

Because we provide two variations of our concept, the testing for reachability for a pair of given `NodeIDs` and the axis type depends on the selected variation. If we use the numbering scheme (explained in Section 3.5.1), we apply the interval method (Equation 3.1). The second option considers the search algorithm of the Rho-Index (see Section 3.5.2). Both approaches provide the same result: *true* if there is a specific relation between two `NodeIDs`; *false* otherwise.

As the further optimization, we compare the `NodeIDs` between each other before checking the A-D relationship because for any two nodes n_1, n_2 it holds that if the node n_1 belongs to ancestors of n_2, the node n_1 must have been visited before the node n_2 in the very first tree traversal while parsing the input file:

$$n_1 = ancestor(n_2) \Leftrightarrow n_1.NodeID < n_2.NodeID \tag{4.1}$$

We also optimize the testing for parent-child relationships. We leverage the structure of the initial graph and check whether the first node is the parent or child of the second node. Depending on the implementation, if we save the list of childnodes as an associative array, we can reveal these relationships in the constant time. Otherwise we get the linear time complexity $O(N)$ where N denotes the number of childnodes for the parent node.

4.4.3 Filter Candidates

Whenever we use the voting mechanism, we need to finalize the candidates. We call this action *filtering* candidates (line 7). The candidate nodes that have not received enough *votes* will be removed from the candidates table. Number of votes needed for being kept equals the number of predicates that has been included in voting.

Suppose that the current node in the `XPathTree` is *xnode*. We define a function $xnode.Votes[nid]$ that returns the number of votes for a specific *nid* (`NodeID`) receives, a function $Result(xnode)$ that returns the result set of nodes for the *xnode*, and a variable $xnode.PredicatesCount$ which provides the total number of predicates for a given `XPathNode`. Then we will get:

$$nid \in Result(xnode) \leftrightarrow xnode.Votes[nid] = xnode.PredicatesCount \qquad (4.2)$$

After we eliminate unsuitable candidate nodes, we need to update the `PTIDs` set for the next step because we expect it to decrease. By updating `PTIDs` we mean find all `PTIDs` where a `NodeID` might occur. If the axis of the next `XPathNode` has the same direction, we take the `PTIDs` only from the already computed smallest `PTID` set for the current `XPathNode`. Otherwise, we need to consider potentially all `PTIDs`. To take only some `PTIDs`, we use the `PrefixTree` that determines only such `PTIDs` in which the given `NodeID` might occur.

To navigate in the `PrefixTree`, we need to find out the *path prefix*. We expect the tree structure as the input, so revealing the *path prefix* is quite easy because each `NodeID` uniquely determines the path to the root in the initial graph. While traversing this path upwards, we store the `TagIDs` of the nodes in the stack. Then we examine the stored `TagIDs`, which represent the *path prefix*. Because we use the stack, the `TagIDs` are already in the reversed order as we reach the root node. So, we can use them in the `PrefixTree` directly for the downward navigation from the root and we finally get a node which holds all `PTIDs` with the given *path prefix*. This is the set of `PTIDs` we need.

We cannot use only the `Paths` table because we will find `PTIDs` for any `NodeID` with the same `TagName` and that produces bigger set than we need for a specific `NodeID`. Therefore we search in the `PrefixTree` instead.

4.4.4 Merge Candidates

If we have two *Step* nodes, we need to merge their candidate nodes (line 10). Usually we take the candidates of the last *Step* node and apply the same voting mechanism on the current `XPathNode` as with predicates. After voting, we automatically filter candidates. The result candidates for the current `XPathNode` contains previous candidate nodes that received votes.

Chapter 5

Experimental Results

We implemented the prototype of the *iXUPT* native indexing method in .NET Framework 2.0 [34] and used the laptop machine with Intel Core Duo processor (1.66GHz), 3GB main memory, and Windows XP SP2 installed to execute the experiments. We take standardized data sets as the input and choose sample queries to evaluate.

We compare both variations of our concept between each other and also with other existing solutions in terms of time complexity when evaluating sample XPath queries.

5.1 Data Sets

For our experiments, we use *XMark* [39] (the XML benchmark database) as the data set. *XMark* is designed to generate XML documents with multiple parts meeting various XML approaches (data-centric and document-centric). Although the generated documents are valid to a specific DTD, we do not use this schema as the hint for indexing or evaluating.

To acquire the data set, we use the tool *xmlgen* [38] which is the implementation of the *XMark*. We generated several documents with different characteristics (see Table 5.1 for details). The main driver of generation is the *Factor* which determines the size of the XML document and number of elements.

Because we focus on indexing paths, we also studied other properties such as number of *Real paths* (the total number of root-to-leaf paths) or number of `PTIDs` (the total number of root-to-leaf paths with different path labels). In the following sections, you find out how the data interrelates.

Table 5.1: Characteristics of the XMark data set

Factor	Size [MB]	Nodes	# Real paths	# PTIDs
0.01	1.12	17 132	12 504	338
0.1	11.32	167 865	122 026	422
0.3	34.05	501 498	364 481	434
0.5	56.28	832 911	605 546	434
1	113.06	1 666 315	1 211 774	434

5.2 Indexing

For testing purposes, we created a simple framework that enables us to automatically run the indexing tests. The configuration file contains the number of iterations for each test, minimal and maximal segment sizes for the graph segmentation if the Rho-Index structure is used, the output CSV[1] file that will contain the results, and the source file to be tested. We chose the CSV format for the output because it easy to parse with other programs (e.g. R or MS Excel).

We observed several properties and in the following sections we provide the results that we acquired.

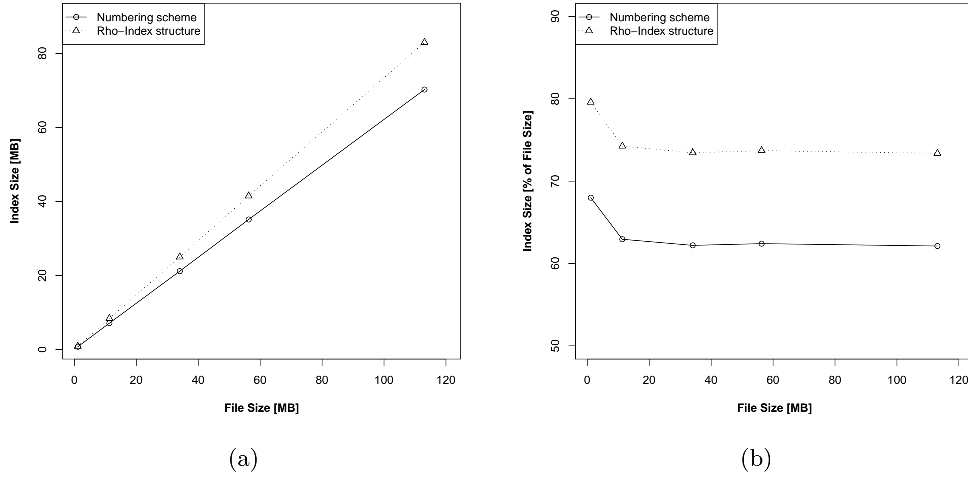
5.2.1 Index Size

First, we focus on the size of the whole index. It means the total size of all major data structures that we use to store the information while parsing the file (for more detail see Section 3.3).

We can see the results in Figure 5.1. There are some differences between the two variations. First, the numbering scheme occupies less memory than the variation that uses the Rho-Index. The reason is that Rho-Index needs more space to store all the graph segmentations and modified adjacency matrices. Although any Path Type Adjacency Matrix contains less than the half of the values (because we take only trees, so there are no reverse edges as in general graphs), there is an overhead; see Figure 5.1(a).

The next thing we want to accent is that the index sizes are always less than the original file size. Although the reason might be the skipped attributes and text values, we are able to build the index that stores all information needed for query evaluation and occupies around 62% (using the Numbering scheme) or 73% of the source file size; see Figure 5.1(b).

[1]Comma Separated Values.



(a) (b)

Figure 5.1: (a) The comparison between the total Index size and the File size, (b) The percentage relation between the Index size and the File size

5.2.2 Creation Time

Except for the index size, we want to compare also the time needed to initialize and build the index. We noticed that the two variations are almost incomparable in terms of creating the index (see Figure 5.2).

The numbering scheme is really fast because we compute the intervals as we traverse the tree. So there is no special dependency on the file size.

On the other hand, the variation that uses the Rho-Index structure is too slow. The reason is that after parsing the file, we need to apply the segmentation, initialize all segments, and create segment graphs. Because there are several iterations of the segmentation, the time complexity dramatically increases.

Table 5.2: Settings for the graph segmentation when building Rho-Index

Factor	**Levels**	**Max Path Length**	**Max Segment Sizes**
0.01	3	11	{20,10,5}
0.1	4	11	{35,20,10,5}
0.3	4	11	{40,25,10,5}
0.5	4	11	{40,25,10,5}
1	5	11	{60,40,25,10,5}

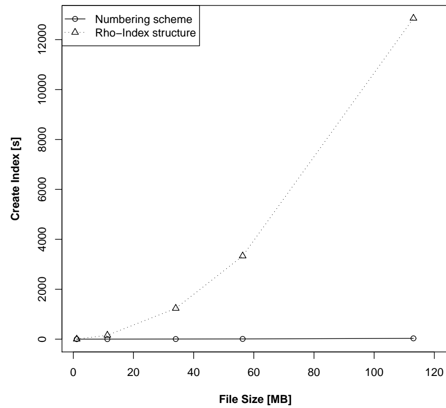


Figure 5.2: The total time of creating the index according to the File Size for both variations

Build Rho-Index

We focus more on the variation with the Rho-Index variation trying to discover the relation between the time we need to parse the file and time needed for building the specific Rho-Index structures. We surprisingly found out that parsing the XML file adds only a small value of 0.3–1% to the total time (see Figure 5.3). It means that even though we are able to quickly parse the file and get the initial graph, we need a lot of time to build and process the Rho-Index structure. This does not seem like an advantage for further usage.

For our testing purposes, we use several different settings for the graph segmentation when building the Rho-Index (see Table 5.2). As the basis, we take the suggested values from the authors of the structure which they show in their work [2]. Their approach considers only smaller graphs (5000–30 000 nodes) which is not sufficient for us because it does not cover all files. Only the first one is capable of being in this interval. Therefore we have to estimate the maximal segment sizes. We tried some settings and took always the one that gave as the most promising results for each file.

Although the proposed segmentation methods have both upper and lower bounds, we do not apply the minimal number of nodes to be in the segment. Without loss of generality, the segmented graph still delivers the expected functionality.

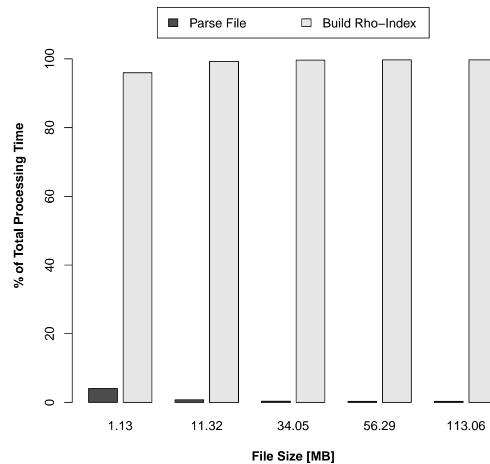
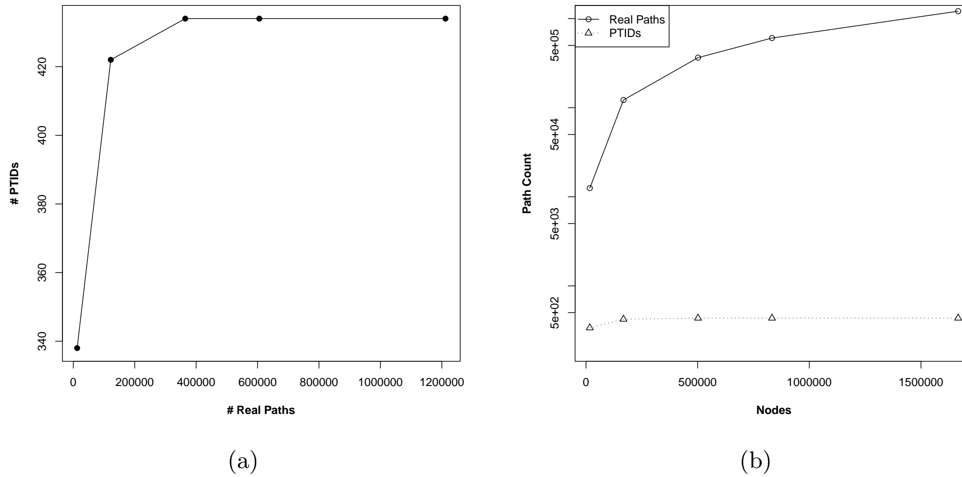


Figure 5.3: The comparison between the time to parse the source file and the time to build the Rho-Index.

5.2.3 Paths Count

We examine also the relation between a number of all root-to-leaf paths and a number of different `Path labels` for the given data sets; see Figure 5.4. As the number of rooted paths in the document rises, the number of paths with unique `Path labels` also increases but in the decreasing manner. The reason is that the XML documents have usually pre-defined structure, so the `Path labels` repeat. Because we assign the new `PTID` (that uniquely identifies a `Path label`) only for the `Path label` that has not appeared yet, the number of different `PTIDs` is smaller than the number of all paths (and in our case, it has an upper bound). This is the fact, why we prefer searching for candidate nodes according to common `PTIDs` which also enhances the speed of the evaluation.

Yet we understand that for general XML files we can easily generate documents where the number of `PTIDs` equals the number of real paths which does not have to improve the querying time. Though it holds that the number of `PTIDs` does not exceed the number of real paths and usually the difference between those numbers is in orders of magnitude.



(a) (b)

Figure 5.4: (a) The relation between the number of different *Path labels* (or `PTIDs` and the number of real paths), (b) The comparison between the path counts and the total number of nodes in the logarithm scale

5.3 Querying

In this section, we focus on the choosing and evaluating sample queries to prove the feasibility of our study and acquire some results, so we are able to compare the accomplished results with other existing solutions.

5.3.1 Sample Queries

We choose two sample queries according to the given DTD of the generated documents that will cover as many features and functionality as possible. The first one is simple while the second one includes different XPath axes and predicates. The aim is to demonstrate our technique and its efficiency on these queries.

After parsing each query, we build the corresponding `XPathTree` structure, and traverse it to get the results. For better imagination, we also provide the visualization of the sample queries in Figure 5.5. We use boxes for the *Step* nodes and ellipses for the *Predicates*. The edge is always oriented and it leads from an ancestor to a descendant. Whenever we use a dashed edge, it means the general ancestor-descendant relationship, while the solid edges cover the parent-child relationships.

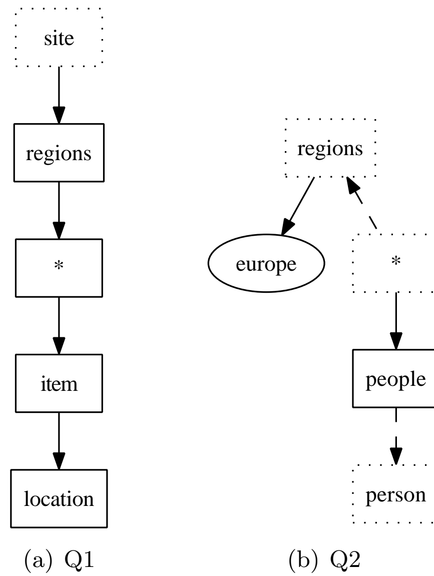


Figure 5.5: The visualization of `XPathTrees` for sample queries.

Query 1 (`site/regions/*/item/location`)

The first query (Q1) is simple, there are no predicates and it uses only direct relatives (the `child` XPath axis) but it contains a wildcard; see Figure 5.5(a). It represents a single path expression.

Query 2 (`//regions[europe]/ancestor::*/people//person`)

This query (Q2) is more complex, there is a predicate, a branching node, and an alternating node. Furthermore, the second *Step* node matches several elements, and different axis directions are used; see Figure 5.5(b).

5.3.2 Query Results

To eliminate any negative effects of the managed code, we run the experiments as several subsequent steps and then we generalize the results. More precisely, we present the average time of ten subsequent executions for each query. For the evaluation time, we used a simple method of obtaining the system time before and after the evaluation of a query and providing their difference. Considering the memory usage, we utilized the *CLR Profiler* [33] tool.

Figure 5.6 displays the evaluation times for both sample queries. We see that the simpler query (Q1) is evaluated much faster than the complex one (Q2). We

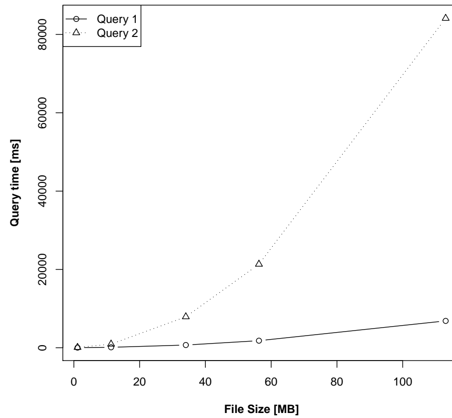


Figure 5.6: The evaluation times for Query 1 and Query 2

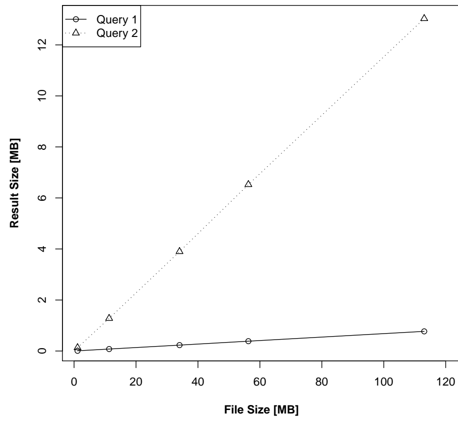
also expect the evaluation time of Query 2 to rise at a faster rate for larger files.

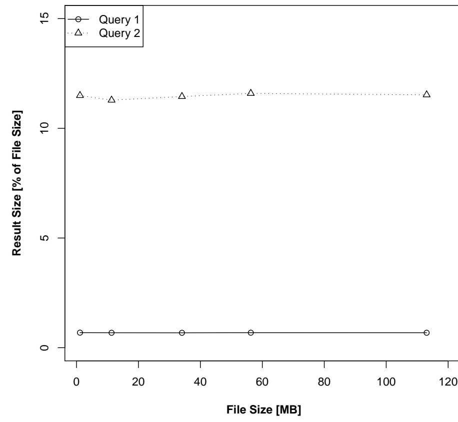
Table 5.3 provides the characteristics of the query results. Although the result node count for Query 1 is about 1.3%, the result size is less than 0.70%. For Query 2 the result node count is similarly around 1.5%, however, the result size is much higher, around 11.5%; for details see Figure 5.7.

Yet we still need to consider that our indexing method is memory-based and it indexes elements only when we compare it with other existing solutions. Therefore we studied the memory usage for query evaluation in more details (see Figure 5.8). We found out that even though the memory usage for evaluating queries increases with larger files, in long term, we are able to approximately set the memory limit for the query evaluation. Figure 5.8(b) shows that for the first query, the bound is about 10–11%, while for the second query it is about 19–20% of the source file size. So, the bounds highly depend on the query.

Table 5.3: Characteristics of the result sets

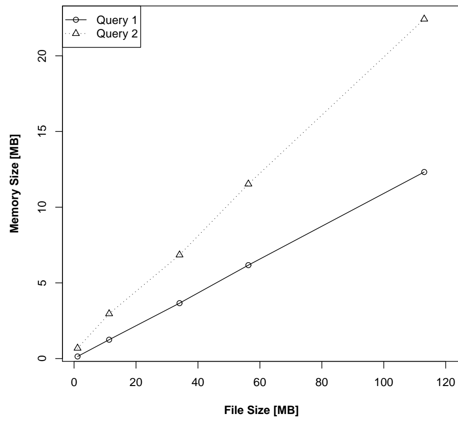
Factor	**Nodes**	**Result Nodes (Q1)**	**Result Nodes (Q2)**
0.01	17 132	217	255
0.1	167 865	2 175	2 550
0.3	501 498	6 525	7 650
0.5	832 911	10 875	12 750
1	1 666 315	21 750	25 550

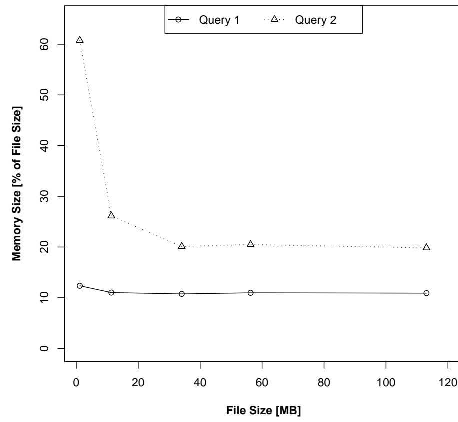




(a) (b)

Figure 5.7: (a) The relation between the result size and the file size, (b) The percentage relation between the result size and the file size





(a) (b)

Figure 5.8: (a) The relation between the query memory usage and the file size, (b) The percentage relation between the query memory usage and the file size

5.3.3 Competitors

As other solutions have different indexing methods than the one we propose and it is not always easy to find out which approach they use and how they use it, we chose the competitors randomly.

We compare the *iXUPT* prototype with several other products such as *eXist* [32], *Qizx* [44], *MS SQL Server 2005* [35], or the built-in *XPathNavigator* in .NET Framework 2.0 (marked as XPN 2.0). While the first two solutions use native indexing of the XML file only because they have been build primarily to store and index XML documents, we have not found out the exact indexing approach the others use.

The Figures 5.9 and 5.10 show the comparison of our *iXUPT* prototype with other solutions in the terms of the time needed to get the result for each query. Also notice that the result values are in the logarithm scale.

Query 1

The first, simple, query proved that there is an improvement of the query evaluation using the common `Path labels`. For the smaller files with less nodes, our concept provides good results.

As the number of nodes rises, the improvements decrease, and for the very last file with more than a million of nodes, our method has been outperformed by almost all selected competitors. We think that this downturn might be partially also the result of using the managed code.

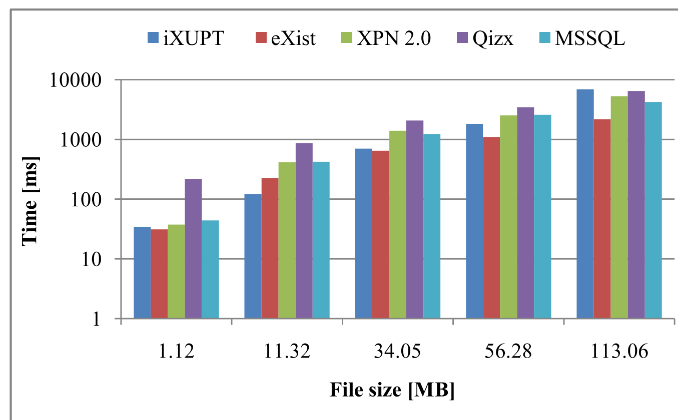


Figure 5.9: The evaluation times for Query 1 in the logarithm time scale

Query 2

As we mentioned earlier, the second query is far more complicated than the first one because it combines several advanced features. So the evaluation times increased.

We had to exclude *MS SQL Server* from evaluating because it does not provide support for `ancestor` axis, so we cannot execute the Query 2.

For the first two files, we are able to compare our solution only with the *Qizx* product. All other methods gave better results. Furthermore, for larger files other solutions outperformed our prototype by almost an order of magnitude.

We expected better results for Query 2 but our solution has not been highly optimized as other existing products. We developed *iXUPT* to prove the feasibility of our indexing concept, so we still see the potential to optimize it in order to gain better performance and provide better results. We leave this optimizations for the future work.

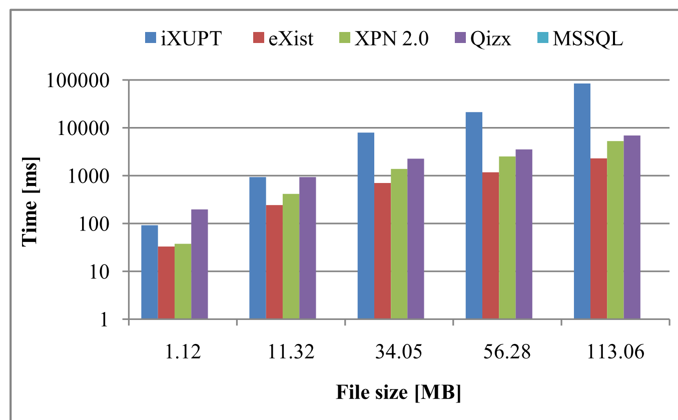


Figure 5.10: The evaluation times for Query 2 in the logarithm time scale

Chapter 6

Conclusion

The main goal of this thesis was to create a native indexing method that we might apply to larger XML databases, or precisely the XML files with a large number of elements.

We started by studying the existing solutions (Section 2.3) which gave us an overview of existing approaches. Although several different methods exist, we found out that one of the most effective ways of indexing XML data is using the paths. The approaches are usually not as simple as element-based concepts but they provide better experimental results in terms of time complexity while evaluating the queries.

We were mainly inspired by the widely known indexing approaches such as using numbering schemes, transformations of paths into multi-dimensional spaces, or structural joins. We studied all these areas in more detail and tried to take only the advantages of each solution. Even though we noticed that structural joins do not provide the best results, the idea of intermediate results fit exactly into our concept.

Taking into account the existing solutions and leveraging their advantages, we described our own indexing method that tries to combine the best ideas of all mentioned approaches. The substantial part of our concept has been described in the work [4] where authors define the main indexing principles.

We propose two variations of the indexing method that differ in the way they handle ancestor-descendant relationships. The first one uses a new numbering scheme, while the second one relies on the *Rho-Index*, the existing data structure used mainly to discover paths in a graph between a given pair of nodes. We build the whole concept on the basis of indexing and labeling paths.

To be able to evaluate regular path expressions, we selected the *XPath* to be the language for path patterns because it is standardized and provides all necessary features we needed.

The main idea of query evaluation is to decompose the *XPath* query into several simple step expressions that we subsequently evaluate producing the intermediate results and then we combine them to build the final result. We use the idea of structural joins but instead of full joins we search for nodes only within the common path labels. This method eliminates several joins that will not be parts of the final result.

We implemented the *iXUPT* prototype to prove the feasibility of our study for both variations. In the Section 5 we compared these two variations from various aspects such as the time of creating the index or the size of the index. Considering the variations, the Rho-Index did not achieve the expected results and has been totally outperformed by the simple numbering scheme. We leave the ideas of how to simplify the Rho-Index for better performance for the future work, especially for larger graphs with a large number of nodes because the structure did not expect such huge graphs.

We also provided the comparison of experimental results between our concept and some other existing solutions and showed how they differentiated from our concept in terms of time complexity while evaluating the queries. Analyzing the results, our methods improved some queries, especially on the files with small or medium number of nodes. But it did not improve the evaluation of complex queries such as branch queries as we expected.

6.1 Future Work

While we were creating and developing the indexing concept, we found out several topics that might be interesting to study in the future. The first possible area might be the modification the proposed numbering scheme, so we can apply it also for general graphs as a replacement for the Rho-Index structure because, as we saw, Rho-Index needs a lot of time to be fully initialized and built and it is not probably the best choice here.

We might also focus on more technical background and rewrite the implementation into other programming language (e.g. `C++`) to avoid the managed code and all the disadvantages it brings. There is still also the potential to optimize the solution and make things ”smarter”.

We might also study a selected subset of XPath axes in more detail and optimize the solution for some of them, or we could add the support for all remaining XPath axes. At least the branch queries or twig queries are a good topic for the future study.

As far as the memory is concerned, an interesting part will be storing the created index with all major data structure to the secondary memory. This brings additional requirements and issues such as how to efficiently load and store the

data, or how to manage them throughout processing queries. But it lowers memory requirements.

Because most data structures we proposed have the characters of tables, we might consider using database tables to implement our approach using the capabilities of relational databases.

We see all of the mentioned areas with the potential to improve our concept and also as good ideas for future work.

Bibliography

[1] Dmitry Barashev and Boris Novikov. Indexing XML to support path expressions. In Manolopoulos and Návrat [30], pages 1–10.

[2] Stanislav Bartoň. *Indexing Graph Structured Data*. PhD thesis, Faculty of Informatics, Masaryk University, 2007.

[3] Stanislav Bartoň and Pavel Zezula. Indexing structure for graph-structured data. In *Mining Complex Data*, volume 165 of *Studies in Computational Intelligence*, pages 167–188. Springer Berlin / Heidelberg, 2009.

[4] Tomáš Bartoš and Ján Kasarda. iXUPT: Indexing XML Using Path Templates. In *DATESO 2010*, CEUR Workshop Proceedings, 2010.

[5] Christian Böhm, Stefan Berchtold, Hans-Peter Kriegel, and Urs Michel. Multidimensional index structures in relational databases. *J. Intell. Inf. Syst.*, 15(1):51–70, 2000.

[6] Jan-Marco Bremer and Michael Gertz. An efficient XML node identification and indexing scheme. Technical report, Dept. of Computer Science, University of California at Davis, 2003.

[7] David Brownell. SAX. http://www.saxproject.org, 2004.

[8] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 455–466, New York, NY, USA, 2005. ACM.

[9] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, and Divesh Srivastava. Index structures for matching XML twigs using relational query processors. *Data Knowl. Eng.*, 60(2):283–302, 2007.

[10] W3C Consortium. W3C XML Specification DTD. http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm, 1998.

[11] W3C Consortium. XSL Transformations (XSLT) 1.0. http://www.w3.org/TR/xslt, 1999.

[12] W3C Consortium. Document Object Model (DOM). http://www.w3.org/DOM, 2005.

[13] W3C Consortium. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20, 1 2007.

[14] W3C Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, 1 2007.

[15] W3C Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/REC-xml, 2008.

[16] W3C Consortium. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. http://www.w3.org/TR/xmlschema11-1, 2009.

[17] W3C Consortium. XForms 1.1. http://www.w3.org/TR/xforms11, 2009.

[18] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[19] Paul F. Dietz. Maintaining order in a linked list. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, New York, NY, USA, 1982. ACM.

[20] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[21] Torsten Grust. Accelerating xpath location steps. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, New York, NY, USA, 2002. ACM.

[22] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[23] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[24] Donald Ervin Knuth. *The art of computer programming*. Addison-Wiley, Reading, Mass., 1968.

[25] Michal Krátký. *Indexing Graph Structured Data*. PhD thesis, Faculty of Electrical Engineering and Computer Science, Technical University of Ostrava, 2004.

[26] Michal Krátký, Radim Bača, and Václav Snášel. On the efficient processing regular path expressions of an enormous volume of XML data. In *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2007.

[27] Michal Krátký, Jaroslav Pokorný, Tomáš Skopal, and Václav Snášel. The geometric framework for exact and similarity querying xml data. In *EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, pages 35–46, London, UK, 2002. Springer-Verlag.

[28] Michal Krátký, Jaroslav Pokorný, and Václav Snášel. Indexing XML data with UB-trees. In Manolopoulos and Návrat [30], pages 155–164.

[29] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[30] Yannis Manolopoulos and Pavol Návrat, editors. *Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002, Bratislava, Slovakia, September 8-11, 2002, Proceedings, Volume 2: Research Communications*. Slovak University of Technology, Bratislava, 2002.

[31] Gerard Marks and Mark Roantree. Pattern based processing of xpath queries. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 179–188, New York, NY, USA, 2008. ACM.

[32] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*, pages 169–183. Springer Berlin / Heidelberg, 2009.

[33] Microsoft. How To: Use CLR Profiler. http://msdn.microsoft.com/en-us/library/ms979205.aspx, 2004.

[34] Microsoft. .NET Framework 2.0. http://msdn.microsoft.com/en-us/netframework/default.aspx, 2006.

[35] Microsoft. Microsoft SQL Server 2005. http://www.microsoft.com/sqlserver/2005, 2009.

[36] OASIS. Reference Model for Service Oriented Architecture v1.0. http://www.oasis-open.org/specs/index.php#soa-rmv1.0, 2006.

[37] Dirk Riehle. Composite design patterns. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 218–228, New York, NY, USA, 1997. ACM.

[38] Albrecht Schmidt. xmlgen - The Benchmark Data Generator. http://www.xml-benchmark.org, 2009.

[39] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark a benchmark for XML data management. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[40] Dr. Harald Schöning. Tamino - a dbms designed for xml. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, page 149, Washington, DC, USA, 2001. IEEE Computer Society.

[41] Trißl, Silke and Leser, Ulf. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, New York, NY, USA, 2007. ACM.

[42] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: a dynamic index method for querying xml data by tree structures. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121, New York, NY, USA, 2003. ACM.

[43] Dave Winer. RSS 2.0 Specification. http://cyber.law.harvard.edu/rss/rss.html, 2003.

[44] XMLmind. Qizx, a fast XML repository and search engine fully supporting XQuery. http://www.xmlmind.com/qizx, 2 2010.

[45] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Technol.*, 1(1):110–141, 2001.

Appendix A

CD-ROM Content

A.1 Source Code

We implemented the indexing concept in the .NET Framework 2.0 using the `C#` language. So we attach the source code as the Visual `C#` 2005 solution. Because the project was originally named *xmlRho*, the source files are located in the `/source/xmlRho` directory.

Both variations are implemented in the same project because of common data structures and common logic. The choice of a variation depends only on the preprocessing directive. If we define the `RHO_INDEX_RELATIONSHIPS` constant, after compiling, we will get the variation using the Rho-Index for testing A-D relationships. Otherwise the numbering scheme will be used.

`/build` contains executable programs for the Windows platform. It contains console applications for

- the variation using the numbering scheme `/build/1.Numbering scheme`
- for the variation using the Rho-Index `/build/2.Rho-Index`.

A.2 Data Sets

The generated documents that we used for the indexing and querying tests are located in `/data_set` directory. We add also DTD for the XML files and the executable version of the program we use to generate the data sets `xmlgen_win32.exe`.

A.3 Test Results

We supply the experimental results for all indexing and querying tests in the folder `/test_results` mostly stored as spreadsheet or CSV files.

A.4 Documentation

We provide the documentation generated from the comments in source files in HTML, LATEX, and XML formats obtained with the Doxygen[1] program.

A.5 Supplemental Programs

While working on the project, we have written some additional programs for supplemental purposes. We supply some of them:

- `/source/xml2graph` Converts the source XML file into a graph structure and visualize it using the *GraphViz*[2] program.
- `/source/xmlStat` Provides statistics about the input XML file such as number of nodes, attributes, or paths.
- `/source/XPathTest` This graphical application implements the evaluation of XPath queries using the built-in *XPathNavigator*.

[1] see http://www.doxygen.org/
[2] see http://www.graphviz.org/