Aus dem Institut für Informationssysteme
der Universität zu Lübeck
Direktor:
Prof. Dr. rer. nat. Volker Linnemann

KeyX: Selective Key-Oriented Indexing in Native XML-Databases

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
- Aus der Technisch-Naturwissenschaftlichen Fakultät -

Vorgelegt von
Dipl.-Inf. Beda Christoph Hammerschmidt
aus Göttingen

Lübeck, November 2005

Beda Christoph Hammerschmidt
Institut für Informationssysteme
Universität zu Lübeck
Ratzeburger Allee 160
D-23538 Lübeck
E-Mail: bchammer@ifis.uni-luebeck.de

Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technisch-Naturwissenschaftlichen Fakultät
der Universität zu Lübeck

Dekan:	Prof. Dr. Enno Hartmann
Gutachter:	Prof. Dr. Volker Linneman
	Prof. Dr. Stefan Fischer

Tag der Promotion:	26. Oktober 2005

Acknowledgments

I would like to thank my colleague Dr. Martin Kempa and my supervisor Prof. Dr. Volker Linnemann for the excellent cooperation, many helpful suggestions, and reviews. Special acknowledgments go to Volker Linnemann for the steady support even besides this project and to Henrike Schuhart for the fruitful discussions we had in the last years.

A larger project like a doctoral thesis in computer science cannot be realized without many helping hands from student workers. I greatly appreciate their results that we gained together in many meetings, discussions and code walk-throughs leading to several bachelor and diploma theses. Especially, I would like to thank (in alphabetic order) Patrick Bär, Konstantin Ens, Chu Danxing, Timm Gehrmann, Khaled al Hayja, Ivo Iken, Florian Massel, Nina Moebius, Alexander Pfalzgraf and Philipp Stursberg.

Many thanks go to Prof. Dr. Stefan Fischer for his co-review and to Prof. Dr. Erik Maehle for chairing my final examination.

Last but not least, special thanks go to to the whole team of the Institute of Information Systems for the support I received during my stay at the University of Lübeck, Germany.

Lübeck, November 2005. Beda C. Hammerschmidt

Contents

1 Introduction **5**

2 Fundamentals **7**

2.1 XML . . . 7
2.1.1 Historical Overview . . . 7
2.1.2 Technical Introduction to XML . . . 9
2.1.3 XMark Sample Data . . . 12
2.2 Document Type Definitions and XML Schema . . . 14
2.2.1 DTD: Document Type Definition . . . 14
2.2.2 XML Schema . . . 16
2.2.3 Other Schema Languages . . . 19
2.2.4 XHTML . . . 19
2.2.5 The Document Object Model . . . 19
2.2.6 Future Perspective of XML . . . 20
2.3 XML Query Languages . . . 22
2.3.1 XPath . . . 22
2.3.2 XPath Query Types . . . 28
2.3.3 XPath Evaluation . . . 29
2.3.4 XQuery . . . 31
2.4 XUpdate: XML Modification Language . . . 35
2.5 XML and Databases . . . 38
2.5.1 XML in Conventional Relational Databases . . . 38
2.5.2 XML in Extended Relational Databases . . . 41
2.5.3 Native XML Database Management Systems . . . 42
2.5.4 Hybrid Approaches . . . 43

3 Formal models for XML and XPath **45**

3.1 A Model for XML Data . . . 45
3.2 A Model for Path Expressions . . . 47
3.2.1 Regular Expressions in Formal Languages . . . 52
3.2.2 Tailing Predicates and Normalization . . . 53

4 Introduction to Recent Approaches in XML Indexing 55
4.1 Structural Indexes 58
4.1.1 The Strong DataGuide 58
4.1.2 1-Index 60
4.1.3 2-Index 60
4.1.4 T-Index 62
4.1.5 Apex 63
4.1.6 Numbering Schemes and Tree Signatures 65
4.1.7 Further Structural Summaries 66
4.2 Value Indexes 66
4.2.1 Inverted Lists 66
4.2.2 Lore Value Index 66
4.2.3 SEQL 67
4.3 Hybrid Approaches 67
4.3.1 Structural Summary plus Inverted List 67
4.3.2 Content-Aware DataGuide 68
4.3.3 ViST 69
4.3.4 Index Fabric 71
4.3.5 System RX Index 73
4.4 XML Indexes and Updates 74
4.5 Conclusion 75

5 The Key-Oriented XML Index KeyX 77
5.1 KeyX Formally 78
5.1.1 Index Declaration 78
5.1.2 Path Extraction Functions 79
5.1.3 KeyX Search Structure 82
5.2 Index Creation Algorithm 82
5.2.1 Structural Indexes in KeyX 84
5.3 KeyX by Examples 85
5.3.1 Single-Key Indexes 86
5.3.2 Multi-Key Indexes 87
5.3.3 Selective Structural Indexes 88
5.4 Query Processing 89
5.4.1 Query Execution with matching KeyX Indexes 89
5.4.2 Index Usage with Deviating Return Values 89
5.4.3 Containment Problem 90
5.4.4 Rating of Indexes for the Query Execution Plan 91
5.4.5 Algorithm for the Query Execution 94
5.5 Performance Measurements 97

6 The Index Selection Problem 103
6.1 Introduction to the Index Selection Problem 104
6.2 Index Selection in Relational DBMS 105
6.3 Index Selection Problem Applied to KeyX Indexes 108
6.3.1 Index Candidates 108
6.3.2 Index Configuration 110
6.3.3 Cost Functions 110
6.3.4 Costs of a configuration 112
6.3.5 Index Selection Problem 113
6.3.6 Exact Algorithm and Heuristics 113
6.3.7 Evaluation and Experiments 115
6.4 Autonomous XML Indexing 117
6.4.1 Architecture and Implementation 117

7 The XML Index Update Problem 123
7.1 Introduction 124
7.2 Intersection of Two Path Expressions 125
7.2.1 Formalization 126
7.2.2 Automaton for $Mod(p)$ 129
7.3 NP-Completeness for Path Expressions with NOT 133
7.3.1 Algorithm and Complexity of XIP^{NOT} 135
7.4 Evaluation 136
7.4.1 Evaluation of Intersection 136
7.4.2 Evaluation of Satisfiability 137
7.4.3 Evaluation of Satisfiability II 139
7.5 Related Work 139
7.5.1 XPath 2.0 140
7.5.2 Containment and Satisfiability of XPath Expressions 140
7.6 Updating KeyX Indexes 141
7.6.1 Update Algorithm 142
7.6.2 Index Maintenance Algorithms 143

8 KeyX Implementation Details 147
8.1 Architecture 147
8.2 XPath expressions 150
8.3 The Index Selection Tool 151
8.4 Index Update Problem 151
8.5 XDLT - A Graphical User Interface for KeyX 153

9 Conclusion and Future Work 157

10 Appendix 161
10.1 XQuery Example 161
10.2 Method testQualifier 161

10.3 XML Schema 162
10.4 List of Publications 163

Bibliography **168**

Index **168**

Index **180**

Chapter 1

Introduction

The eXtended Markup Language XML nowadays (2005) is an important standard in IT industries for the platform and language independent expression of structured data. In so-called service oriented architectures with business logic encapsulated in web services, self describing XML messages are exchanged between a service provider and a requester. This approach allows building loosely coupled information systems which tend to be easier to modify and maintain.
Although XML is used mainly for data exchange, there is an emerging need for enabling the persistent storage of XML data in databases. For example, a clearing house interacting with member brokers using web services is legally obliged to store the XML messages for non-repudiation.
When XML data is stored in a database management system many questions arise: How to express queries and updates? How can XML data be indexed to speed up frequent queries? How to detect if a modifying operation affects an established index and must therefore be updated to keep it consistent? And last but not least: How can indexes that are best for a given application be determined automatically?

Because of the semistructured (tree-like) model of XML data the usage of relational databases is mostly not possible or accompanied by severe problems like many expensive Joins and non-trivial XML to SQL query rewriting. Therefore *native* XML database management systems are becoming more and more popular. In this case *native* means that the structure of the XML data is not mapped to flat tables but is reflected in the internal persistent data model.

This thesis introduces an approach for indexing XML data stored in a native XML database management system. The indexes are used to accelerate XPath based queries. Only the parts of the XML data that are selected by a given set of XPath queries are reflected in the materialized indexes. This feature - called *selectivity* - reduces the space consumption of all indexes and leads to less update operations if the original XML data is modified.

Due to the complex path expressions in XPath that may use structural and value conditions the decision whether an XML index is affected by an operation that modifies the underlying original XML data is not trivial. This problem is reduced to the emptiness of the intersection of two XPath expressions and can be decided efficiently by an algorithm that relies on finite automata.

The well known *Index Selection Problem* that searches for an optimal set of indexes for a given set of querying and modifying database operations is extended and transferred from the relational world to XML indexes. Existing heuristics for finding a good solution efficiently can be reused in the index selection tool of the implementation. An autonomous XML database management system is realized if the index selection tool is called periodically and creates and drops indexes automatically.

Basically, this work summarizes the publications of the workshops and conferences GVD 2004 [43], DEXA 2004 [44], SMDB 2005 [45], ICEIS 2005 [47], ED-MEDIA 2005 [48], IDEAS 2005 [46] and two technical reports [41] and [42]. For a reader with previous knowledge who is only interested in a specific aspect of this work it may be more efficient to read just the relevant parts in one of these papers. The abstracts of all publications can be found in the appendix of this thesis.
First of all the thesis gives a brief introduction to XML and related fundamentals as deep as required to understand further definitions and realizations. In addition, an extensive survey with illustrative examples about related work in XML indexing is provided and compared with the own approach.

The thesis is structured as follows: Chapter 2 introduces the fundamentals required to understand his thesis; it includes a brief introduction to XML and to relevant technologies like XML databases and query languages. Chapter 3 defines a formal representation of XML data and path expressions. Related work on XML indexing is introduced and discussed in the following chapter 4. Our own approach - called KeyX - is presented in chapter 5. The problem of finding good indexes for a given database and keeping them consistent when the database is modified is discussed in chapter 6 and 7. Details about the architecture of the indexing system and its implementation is given in section 8. Section 9 concludes this thesis.

Chapter 2

Fundamentals

This chapter introduces XML and related technologies that are used within this thesis as basis for the KeyX index approach. Due to lack of space we do not provide a full introduction to all technologies but concentrate on the aspects that are relevant and required to understand this work. For more detailed information we refer the reader to books or tutorials.

2.1 XML

In this section we give a brief and informal introduction to the eXtended Markup Language (XML) starting with a retrospection on the roots of XML. Within this thesis we use sample data of an auction scenario for illustration purposes.

2.1.1 Historical Overview

In 1986 the *Standard Generalized Markup Language* (SGML) (e.g. [36, 122]) became an ISO standard (ISO 8879) for defining the structure and the content of electronic documents. The goal of *markup languages* like SGML is to describe the logical organization of documents independently from platforms and applications. Historically, the word *markup* has been used to describe annotations within a text intended to instruct a typist how a particular passage should be printed. As the formatting of texts was automated, the term was extended to cover all sorts of special markup codes inserted into electronic documents.

There are three characteristics which distinguish SGML from other markup languages like the scientific text processors LaTeX[69] or Troff [109]: it is descriptive rather than procedural, there is a document type concept and it is independent of particular hardware and software systems. The three characteristics are described in the following:
A *descriptive markup* system uses markup codes (so-called *tags*) which categorize parts of a document. Tags like `<book>` or `<title>` simply identify a part of a document so that it is even readable and in most cases understandable for humans.

In contrast, a *procedural markup* system defines what processing has to be performed at particular points in a document. In SGML, the instructions to process a document are separate from the descriptive markup. Usually, they are collected outside the document in separate procedures or programs. With descriptive instead of procedural markup the same document can easily be processed by many different systems with different processing instructions and different results. For instance, the same content can be processed to an internet page and a paper manual. SGML does not define the graphical layout or the presentation of these documents.

Secondly, SGML has introduced the concept of *document types*, and therefore *document type definitions (DTDs)*. Documents are regarded as having a type, just as other objects processed by computers do. Roughly, a DTD defines the tags that may be used in a document and their structure, e.g. a `<book>` tag may have many `<author>` tags but only one `<title>` tag. By the use of a validating parser it can be determined whether a document belongs to a DTD. The main advantage of the document type concept is that different documents of the same type can be processed in a uniform way.
A fundamental goal of SGML was to ensure that documents are compatible in all software and hardware environments without loss of information. The two concepts discussed so far address this requirement at an abstract level; the third feature addresses it at the level of the encodings of which documents are composed and addresses the notorious inability of different systems to understand each other's character sets.

One popular example for the successful usage of SGML are the Linux HOWTO manuals of the Linuxdoc project [71] describing the Linux operating system. With SGML the texts can be processed to online documentations in HTML format as well as printed books.
Another example for the usage of SGML is the *CALS - Computer Aided Acquisition and Logistics Support* project of the U.S. Department of Defense. With CALS the distribution and storage of technical documentation is standardized and relieved. Suppliers of military products were forced to represent their documentation in SGML.
The most known application of SGML is the page description language *HTML* [120] that is used to send content over the internet and to represent it graphically in a web browser. The permitted tags of a HTML document and their structure is well-defined and restricted - a user may not add new tags to express a custom concept. Most tags deal with visually formatting the page. Therefore, a HTML document mixes content and structure.

In the mid 1990s there was an emerging need to provide more flexible and customized documents over the internet. The visualization-centric approach of HTML did not suit the demands of distributed enterprise information systems with data

being processed without any graphical aspects.
SGML was judged to be too complex to be used for web-based information processing [100]: SGML contains many features that are very rarely used. Its support for different character sets is weak which causes problems on the web where people use many different platforms and programming languages. It is also difficult to interpret a SGML document without having the definition of the markup language (the DTD) available. These difficulties and the lack of SGML-related software like editors have condemned SGML to being a niche technology rather than a mainstream approach in document managing. Indeed some cynics have renamed SGML to 'Sounds Good Maybe Later'.

To solve the complexity issue the *eXtended Markup Language (XML)* was designed to be a simplified subset of SGML. It eliminates the features that make SGML difficult to learn and parse while retaining most of the power of SGML. XML was designed by the The World Wide Web Consortium (W3C) to be, in their own words, "straightforwardly usable over the internet". The W3C's XML 1.0 recommendation was first issued in 1998. The goals of XML as defined by the XML W3C Working Group in the XML specifications [91] are:

- XML shall be straightforwardly usable over the internet.
- XML shall support a wide variety of applications.
- XML shall be compatible with SGML.
- It shall be easy to write programs which process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
- XML documents should be human-legible and reasonably clear.
- The XML design should be prepared quickly.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance.

2.1.2 Technical Introduction to XML

For space restrictions this thesis gives only a brief and informal introduction to the eXtended Markup Language (XML). For more extensive information about XML the reader is referred to popular online tutorials (e.g. [113]), books (e.g. [49]) and the W3C's XML Specification [119].
For reasons of understandability XML is motivated and described with sample documents - taken from the *XMark* project. XMark models an auction scenario consisting of items, categories, sellers, and bidders. This way, a particular XMark

data is somehow similar to a current state of the internet auction *eBay*. More information about XMark is given in section 2.1.3.

Elements and Tags

The core of all XML documents are the *tags* which identify *elements*. A tag is just a generic label for an element. An opening tag looks like `<element>` while a closing tag has a slash that is placed before the element's label: `</element>`. All contents belonging to an element must be contained between the opening and closing tags of the element. An element may contain other elements, a text value, or a mixture of both (so-called mixed content). The following example shows one element `<name>` that has the content `Sinus MP3 Player`.

```
1  <name>
2    Sinus MP3 Player
3  </name>
```

Each opening tag must be closed by a corresponding closing tag with the same name. The resulting nested structure is comparable to the structure of brackets of mathematical formulas. In this case we speak of a *well-formed* document. In general, if a document is not well-formed it is not an XML document and it will cause errors when read by a software that tries to parse it.

Elements can be nested recursively to express the relationships between them. In the second example the `<name>` element is positioned in an `<item>` element that contains further elements:

```
1  <item>
2    <name>
3      Sinus MP3 Player
4    </name>
5    <location>
6      Lübeck, Germany
7    </location>
8    <quantity>
9      25
10   </quantity>
11   <payment>
12     Cash
13   </payment>
14   <payment>
15     Creditcard
16   </payment>
17   <description>
18     <text>
19       This is a portable <emph> MP3 Player </emph> with 512 MB internal memory.
20     </text>
21   </description>
22 </item>
```

An element a that is located under an element b is called *child* of b. For instance, in the example `<quantity>` is a child of `<item>`. Analogously, we say that an element c is a *parent* of b if b is a child of c. Elements that have the same parent are called *siblings*: in the example the elements `<payment>` and `>location>` are

siblings. An element d is a *descendant* of e if it is a recursive child of e. Analogously, we speak of an *ancestor*.

It is possible and usual that elements with the same label are siblings, i.e. the two `<payment>` elements. Usually their content is different but this is not required.

In XML the label of elements and their structure is not predefined so that a user can define her own elements corresponding to the given application. In a XML document the order of the elements matters; this is an important feature that guarantees that the chapters of an electronic book appear in the right order.

Attributes

Attributes are used to specify additional information about an element. An attribute is assigned to an element if it appears within the opening tag:

```
1 <item id = "item0" >
```

An element may have arbitrarily many attributes but they are not allowed to have the same label. In addition, attributes have no order. In general, attributes are replaceable by elements that have a higher expressiveness. For example, the equivalent XML document without attributes is:

```
1  <item>
2    <id>
3      item0
4    </id>
5  </item>
```

The question when to use an element and when to use an attribute is not always easy to answer. In principle, an attribute cannot be used to express an entity if it has children or if it may appear more than once per parent or if the order matters. For all other cases we can choose between an attribute or an element. In most cases, attributes lead to more compact documents. From the theoretic point of view, attributes do not enhance the expressiveness of XML documents. Any document containing attributes can be transformed to an equivalent document without attributes by creating a new element for each attribute. Therefore, this work omits attributes and focuses on elements.

References

Basically, with XML it is possible to express *tree-like* data structures with a node having multiple children but only one parent node (except the root). For a significant number of scenarios tree-like data are too restricted. For instance, a book may have several authors and an author may have written several books. In a tree representation either the authors or the books must be repeated as done in the DBLP data [70]. The redundant repetition of data can only be avoided by using references between nodes.

XML supports references by the use of `ID`, `IDREF` and `IDREFS` statements: An attribute can be assigned as an identifier or key of the element by adding the `ID` key word to the attribute's definition in the DTD (see next section). The value of this attribute is a text value that is unique within the whole XML document and identifies the corresponding element. An attribute that is assigned the `IDREF` property may now refer to an element that has an attribute with an ID. The reference is realized by using the same values for both attributes. Because an attribute with a given name may only be assigned once to an element, we cannot create multiple references from one element to several others. Therefore, another keyword `-IDREFS-` is needed. An attribute that has the `IDREFS` property may contain more than one reference, separated by whitespace characters.

The parser of the document checks whether all elements with an `IDREF` or `IDREFS` attribute are referring to an existing element that has the same attribute value. Anyhow, with the described reference mechanism it is not possible to restrict the references to certain types. For instance, let us assume that both books and authors in a document have unique keys and that authors are referencing to books and books are referencing to authors (bidirectional references). It is not possible to prevent books from referencing books and authors from referencing other authors as only the existence of the `ID-IDREF` pairs is checked and not their type.

2.1.3 XMark Sample Data

XMark is a research project initiated in 2002 by several institutions including Microsoft, INRIA (France), CWI Amsterdam and FhG-IPSI Darmstadt. The goal of XMark is to provide scalable XML data that can be used for benchmarks and tests of XML processing systems like database management systems. XMark comes with a document generator that produces XML sample documents of custom size. The documents are valid according to a given schema (see below).
We use XMark data in the examples of this thesis because the structure of the data is complex enough to express meaningful path expressions in queries and updates. Additionally, the auction scenario is very demonstrative and motivating. Other popular XML data like the DBLP [70] could also be used but are less structured and uniform over larger sections. In contrast to an own unknown data format that might be optimized for this thesis the queries and results are checkable.
The most important elements and their relationships are illustrated in figure 2.1. The figure is taken from [104].
The information about all items that are offered in the auctions are stored below the `regions` element. For each continent there is an own region element (e.g. `europe`). Each item is assigned to one or more categories that are listed and described under the `categories` element. Categories have relations (e.g. convertibles is a sub-category of cars) that are expressed by edges. This information

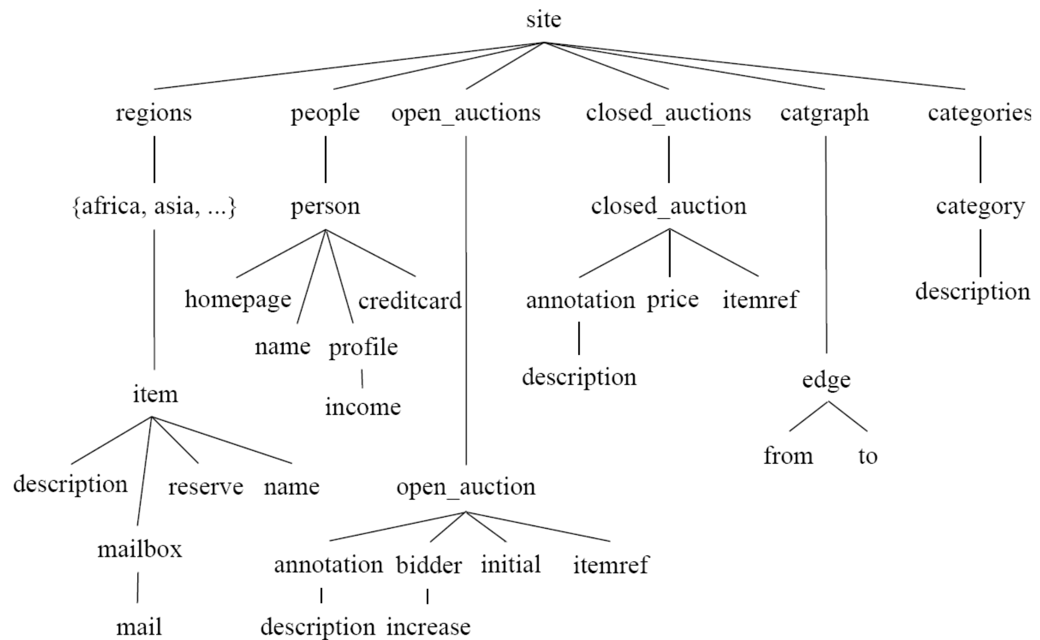


Figure 2.1: Relationships between XMark elements

can be found under the `catgraph` element.
The data of persons are kept under the `people` element. A person can be a bidder or a seller.
The auctions are divided into open auctions where the final buyer is not yet found and closed auctions that do not change anymore. This information can be found under the `open_auctions` and `closed_auctions` elements.
The references between elements are realized with unique identifiers stored in attributes values of auctions, persons, and items. For instance, every closed auction stores the final bidding price and a reference to the sold item. The references rely on the `ID - IDREF` mechanism.

2.2 Document Type Definitions and XML Schema

So far, we made no restriction on the elements' labels and their structures. For most applications not every well-formed XML document is understandable and processable: For example, an auction system that expects XMark data will not be able to process an XML formatted list of publications. Technically it is possible to read and parse the elements but semantically the application is not aware how to deal with it.
Therefore, we need a mechanism to declare a class or type of documents. This is done by schema languages like *Document Type Definitions* and *XML Schema* documents. The idea is to predefine the allowed element labels and to declare how they are allowed to be nested. Schemas are comparable to grammars for programming languages, however, context-free grammars describe sets of words whereas we need to describe sets of trees. The term "schema" comes from the database community.
If an XML document satisfies all constraints of a schema it is *valid*. Validity implies that a document is well-formed and is checked by validating parsers.

2.2.1 DTD: Document Type Definition

A significant feature that XML inherits from its predecessor SGML is the concept of a Document Type Definition (DTD). The DTD is an optional feature which provides a formal set of rules to define a document structure. It defines the elements that may be used and states where they may be applied in relation to each other. Therefore, the DTD defines the document's hierarchy and granularity.
In the following figure the DTD for an XMark fragment is presented.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!ELEMENT item (name, location, quantity, payment+,description?)>
3  <!ELEMENT name (#PCDATA)>
4  <!ELEMENT location (#PCDATA)>
5  <!ELEMENT quantity (#PCDATA)>
6  <!ELEMENT payment (#PCDATA)>
7  <!ELEMENT description (text)>
8  <!ELEMENT text (#PCDATA | bold | emph)*>
9  <!ELEMENT bold (#PCDATA)>
10 <!ELEMENT emph (#PCDATA)>
11 <!ATTLIST item        id ID #REQUIRED>
```

Figure 2.2: The DTD for an XMark fragment

Line 2 states that the root element is an `<item>` containing a sequence of `<name>`, `<location>`, `<quantity>`, `<payment>` and `<description>` elements. The $+$ symbol indicates that the `payment>` element may appear more than once. A $*$ symbol states that zero to many elements are allowed. The $?$ symbol indicates that an element may appear zero times or once. In the example an item may have a description but it does not need to have one. If no symbol is attached to an element it may appear exactly once as child.

The definition of the new elements is carried out in a similar manner: Line 3 states that a <name> element contains one text value (a string) indicated by #PCDATA. The same applies in lines $4-6$.
Line 7 states that a <description> element has exactly one <text> child consisting of a mixed content of strings and text markups. The mixed content is formally expressed by a sequence of unlimited length ($*$) of choices ($|$).
Line 11 defines an attribute with the name id and assigns it to the <item> element. The keyword #REQUIRED states that the attribute is not optional but must appear in the element.

Validity is checked by a validating parser by a simple top-down traversal of the particular XML document

At first look the Document Type Definitions look somehow similar to regular or context-free grammars of formal languages with #PCDATA as terminals and the element labels as non-terminals. But because the element labels appear in the XML document they are not equivalent to non-terminals. This is the reason why DTDs are not closed under union: For two DTDs a and b sharing at least one identical element label n with different rules there is no third DTD c that defines all documents which are valid for a or b. The usual approach of formal languages to rename the ambiguous non-terminals cannot be applied in DTDs because renaming changes not only the DTD but also the derived XML documents. This fact is illustrated by the following example:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!ELEMENT book (title, author)>
3  <!ELEMENT author (#PCDATA)>
4  <!ELEMENT title (#PCDATA)>
```

Figure 2.3: DTD a

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!ELEMENT book (title, author, price)>
3  <!ELEMENT author (first,last)>
4  <!ELEMENT first (#PCDATA)>
5  <!ELEMENT last (#PCDATA)>
6  <!ELEMENT title (#PCDATA)>
7  <!ELEMENT price (#PCDATA)>
```

Figure 2.4: DTD b

The first DTD declares a <book> element to have a <title> and an <author> child. The <book> elements defined by the second DTD have an additional <price> child and the <author> element is not a single text value but consists of the two elements <first> and <last>.

When unifying both DTDs to a single DTD something like the following may

be constructed: In this DTD a book has an optional `<title>` element and an `<author>` may be an atomic text value or a sequence of `<first>` and `<last>`.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!ELEMENT book (title , author , price?)>
3 <!ELEMENT author (#PCDATA | (first ,last))>
4 <!ELEMENT first (#PCDATA)>
5 <!ELEMENT last (#PCDATA)>
6 <!ELEMENT title (#PCDATA)>
7 <!ELEMENT price (#PCDATA)>
```

Figure 2.5: DTD c

At first sight this DTD c seems to express the union of both DTDs a and b but the correlation between the elements `<price>`, `<first>`, and `<last>` is lost. Therefore, the following XML document is valid concerning DTD c although it is not valid for a and b:

```
1 <book>
2   <title> A book title </title>
3   <author> An author </author>
4   <price> 4.99 </price>
5 </book>
```

Beside this more theoretical problem (in practice one can solve the union problem with two separate validation processes) DTDs lack in many points: The most significant disadvantage of DTDs is that they are not formatted in XML syntax themselves. This implies that separate parsers and software tools are required. A tool that is able to store and visualize XML documents can therefore not support DTDs if it offers no separate implementation. Second, a learner of XML has to learn the syntax of XML and the syntax of DTDs. The top 15 reasons for avoiding DTDs are listed in [5].

2.2.2 XML Schema

XML Schema[127] was introduced as a working draft in 2000 by the W3C to be the successor of the Document Type Definition. One of the greatest improvements of XML Schema is the support of data types allowing to specify the values of elements and attributes. The most common atomic data types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

With the support for data types it is easier to validate the correctness of data and to work with data from a database with typed columns. With XML Schema a user is able to define his own data type by the composition of atomic types. The inheritance mechanism of XML Schema enables the definition of new types by enhancing existing ones.

The following listing shows an XML Schema for the XMark sample of figure 2.6:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
       qualified" attributeFormDefault="unqualified">
3      <xs:element name="item">
4          <xs:complexType>
5              <xs:sequence>
6                  <xs:element name="name" type="xs:string"/>
7                  <xs:element name="location" type="xs:string"/>
8                  <xs:element name="quantity" type="
                       positiveNumber_type"/>
9                  <xs:element name="payment" type="xs:string"
                       minOccurs="1" maxOccurs="unbounded"/>
10                 <xs:element name="description" type="
                       description_type"/>
11             </xs:sequence>
12             <xs:attribute name="id" type="xs:string" use="required"/>
13         </xs:complexType>
14     </xs:element>
15     <xs:simpleType name="positiveNumber_type">
16         <xs:restriction base="xs:integer">
17             <xs:minInclusive value="0"/>
18             <xs:maxInclusive value="1000"/>
19         </xs:restriction>
20     </xs:simpleType>
21     <xs:complexType name="description_type">
22         <xs:sequence>
23             <xs:element name="text" type="desc_text_type"/>
24         </xs:sequence>
25     </xs:complexType>
26     <xs:complexType name="desc_text_type" mixed="true">
27         <xs:sequence>
28             <xs:element name="emph" type="xs:string" minOccurs="0"
                   maxOccurs="unbounded"/>
29             <xs:element name="bold" type="xs:string" minOccurs="0"
                   maxOccurs="unbounded"/>
30         </xs:sequence>
31     </xs:complexType>
32 </xs:schema>
```

Figure 2.6: An XML Schema document

The `<schema>` element is the root element of every XML Schema. Like in Document Type Definitions the root element `<item>` is defined in the first lines of the schema. Here the `<item>` element is defined as a *complex type* consisting of the elements `<name>`, `<location>`, `<quantity>`, `<payment>`, and `<description>`. The keywords `minOccurs="1"` and `maxOccurs="unbounded"` state that the `<payment>` element has to appear one to many times in the sequence. If an element has no `minOccurs` and `maxOccurs` attributes the value 1 is assigned implicitly. This im-

plies that the element has to appear exactly once.
The element `<quantity>` is assigned to the user-defined type `positiveNumber_type` that is defined in line 15 as a restriction of the `integer` type. Another user-defined type is assigned to the element `<description>`. A description contains a `<text>` element that is of a mixed content type consisting of `<emph>` and `<bold>` elements. Due to the keyword `mixed="true"` in line 26 an atomic text value can appear between the elements `<emph>` and `<bold>`.

A great strength of XML Schemas is that they are written in XML syntax, so that existing XML editors and parsers can be reused. XML Schema can be created and processed by any software that works with XML documents. Especially an XML Schema can be validated against the XML Schema that defines valid XML Schema documents.

Because XML Schemas tend to be large XML documents (especially larger than DTDs) it is difficult to perceive them. Therefore, a graphical representation is often more comfortable. Figure 2.7 presents *XMLSpy*s [4] visualization of the sample schema.

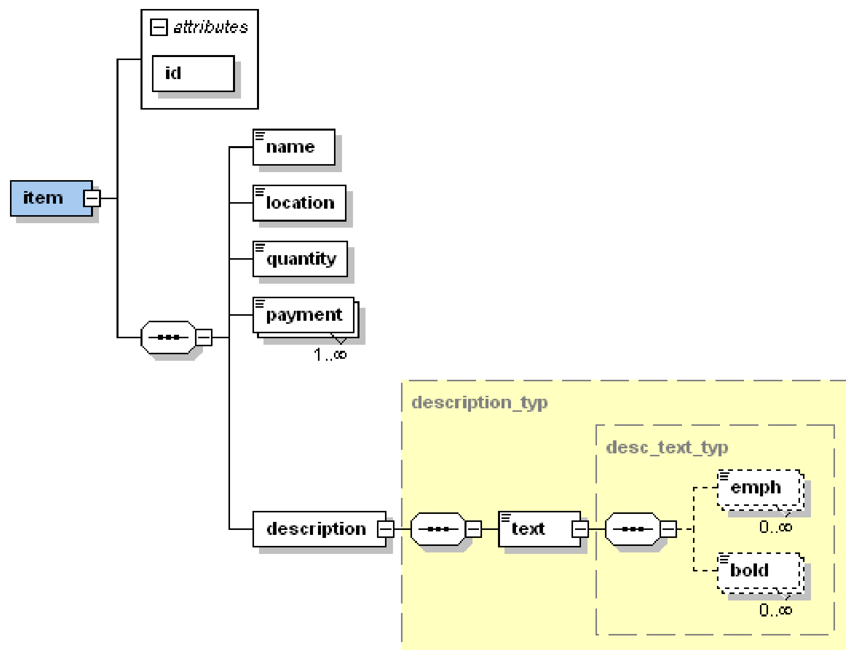


Figure 2.7: A graphical representation of an XML Schema

Although it is semantically possible to create an XML Schema c that defines the union of two XML Schemas a and b this is not processed as expected by the validating parsers like XMLSpy . Syntactically, c can be defined using the inheritance feature of XML Schema (see appendix 10.3). Therefore, as DTDs, XML Schema are not closed under union; details can be found in [86, 87].

2.2.3 Other Schema Languages

There are a multitude of further schema languages to define types of XML documents. Examples include *Schematron* [99], the *Document Structure Description (DSD)* [65], the *Document Definition Markup Language(DDML)* [130], and *Schema for Object-oriented XML (SOX)* [131]. Most approaches gained no or only little importance. The current state-of-the-art is XML Schema, although DTD is still very common because of its more compact form.

2.2.4 XHTML

XHTML [125] is a working draft for the formulation of HTML 4.0 Internet pages and can be seen as an application of XML and DTDs. The goal of XHTML is to guarantee that websites are well-formed and valid according to a given XHTML DTD. XHTML is designed for portability: There will be an increasing use of non-desktop user agents to access HTML documents. In some cases they will not have the computing power of a desktop platform, and will not be designed to accommodate badly formed HTML as current browsers do. Indeed if these programs do not receive well-formed XHTML they may simply not display or process the document.

With XHTML the overlapping of elements is banned (e.g. `<p>an emphasized <em>paragraph.</p></em>`). A document with overlapping elements can never be well-formed. Instead the elements have to be nested (e.g. `<p>an emphasized <em>paragraph.</em></p>`).
Technically, the author of a static website uses the XHTML DTD and checks his HTML code with a validating XML parser. If the parser accepts the HTML code the author may publish it.
For dynamically generated HTML with data coming from a database this approach is not optimal, because it implies that each data has to be validated before sending it to the user. This may cause high costs of computation and there are few chances to automatically fix an HTML document that is rated as invalid by the parser.
The problem of validating dynamic HTML pages is faced in many works including Xtatic [31], XDuce [53], XAct [64], JWig [18], XJ [50], XOBE[61] and many others. The XOBE project faces the problem with a static check of the source code that generates the HTML code. With a positive static check it can be guaranteed that each run of the program generates a valid HTML page. Therefore, the check at runtime is avoided.

2.2.5 The Document Object Model

Until now all XML documents were represented as text documents consisting of lines with a textual representation of the elements and their contents. This facilitates the readability for humans but a text file is not appropriate for navigating in

the structure of the document. Therefore, computer systems usually do not handle an XML document as a string but represent it in a tree-like data structure. The transformation of the textual representation is done by a parser and implies that the XML document is well-formed. One common representation is the *Document Object Model* (DOM) - a proposal of the W3C for platform-independent access on XML documents.
In the following figure 2.8 the DOM tree for the XML example is presented.

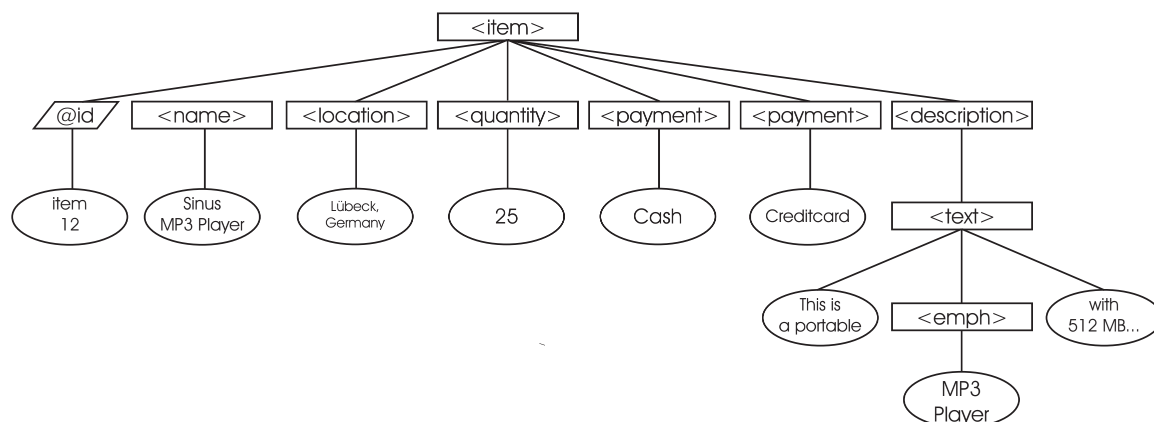


Figure 2.8: A DOM-tree

The DOM tree consists of three node types: elements (illustrated as boxes), attributes (rhombus) and the text values (ovals). In object-oriented programming languages like Java the node types are represented as classes. Each node of the DOM tree is an object/instance of the corresponding class and offers methods to access its content. For instance, the W3C's Java DOM API offers methods like `Boolean hasAttribute(String name)`, `String getTagName()` or `NodeList getChildNodes()` for elements in the DOM tree. This way a computer program can easily process and generate XML documents.

2.2.6 Future Perspective of XML

After its introduction in 1999 XML was hyped and considered as the universal solution solving all data related problems. Of course, this is not the case, especially XML is not replacing the relational data model. However, today (2005) XML is an important and widespread technology that is mostly used for data interchange.
The success of XML can be seen in the huge amount of XML languages that can be found in almost every field of knowledge: Besides the usage as exchange format XML is used to build visualization independent documents in content management systems, to express mathematical or chemical formulas [88], MathML [121] , or in the e-learning context to model learning materials [93], for instance.
Another emerging technology that relies strongly on XML are *web services* . In this context XML is used to describe the functionality of web services using the *Web Service Description Language* (WSDL) [124], to advertise web service providers us-

ing the *Universal Description, Discovery and Integration* (UDDI) registry [110] and for the communication between a web service consumer and the provider using the *Simple Object Access Protocol* (SOAP) [123].

Recapitulating, one can say that the requirements of the XML specification of 1999 were accomplished and that the introduction of XML in real applications was fruitful.

2.3 XML Query Languages

The DOM API provides a universal way of accessing an XML document by a programming language. However, the explicit navigation to nodes by a programming language is not very comfortable. For instance, the following Java code selects all `<name>` elements of items that have the value `'MP3 Player'` in its description.

```
1  Document doc = (new DOMBuilder()).build(XMLSource);
2  Element root = doc.getRootElement();
3  Element description = root.getChild("description")
4  if (description!=null){
5    Element text = description.getChild("text")
6    if (text!=null){
7      Element emph = text.getChild("emph")
8      if (emph != null && emph.getText().equals("MP3 Player")){ // found
9        Element name = root.getChild("name");
10       return name;
11     }
12   }
13 }
14 return null; // not found
```

Figure 2.9: A Java program for accessing the DOM-tree

As one can see quickly the code is not intuitive and descriptive. Rather than defining *which* nodes are selected, the programmer has to express *how* to select them. For more complex queries with more than one value comparison the code will quickly become unmanageable. For this reason a more descriptive and compact query language for XML is needed.

2.3.1 XPath

One common approach is *XPath* [126] a language for expressing navigational steps and conditions for selecting nodes (elements and attributes) in an XML document. XPath 1.0 was introduced in 1999 by the W3C with a definition for the syntax and the semantics. Specific XPath implementations are offered by third parties like [4, 7, 101, 108] or are part of a database management system, e.g. [8, 55].

The following expression motivates XPath: It selects the same nodes as the previous Java code but is much more compact:

```
1   \item\name[..\description\text\emph='MP3 Player']
```

XPath is a navigation language for selecting parts of an XML document modeled as a tree of nodes. XPath discriminates between seven types of nodes: The documents root-node, normal elements, text nodes, attributes and, less important, comment-nodes, processing instructions, and XML Namespaces .

XPath Expression

The primary syntactic construct in XPath is the *expression*. An expression is evaluated by an XPath engine to yield an object, which has one of the following four basic types:

- a node-set (an unordered collection of nodes without duplicates)
- a Boolean value
- a number or a floating-point number
- a string

Location Path

The most important expressions are the *location paths*: A location path selects a set of nodes relative to the context-node(s). When a location path is evaluated it returns a node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter the nodes.

There are two kinds of location paths: *relative* location paths and *absolute* location paths: An absolute location path starts with a / identifying the root element of the XML document as the context-node. An absolute location path may be optionally followed by a relative location path.
A relative location path consists of a sequence of one or more *location steps* separated by *axes*. The steps in a relative location path are composed together from left to right. Each step selects a set of nodes relative to a context-node.

Location Step

Each location step has three parts: An *axis* which specifies the relationship between the nodes selected by the location step and the context-node, second, a *node test*, which specifies the node type and the name of the nodes selected by the location step, and third, arbitrary many *predicates* which use expressions to filter the set of nodes selected by the location step.

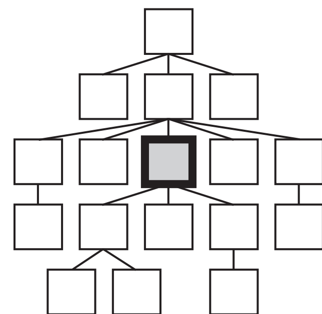
The syntax for a location step in the unabbreviated syntax is the axis name and node test separated by a double colon, followed by the predicates each in square brackets. For example, in `child::para[position()=1]`, `child` is the name of the axis, `para` is the node test and `[position()=1]` is a predicate. The axes, predicates and their semantics are described in the following.

Axes for navigation

XPath provides a multitude of axes for navigating in an XML document. The initial point of the navigation is the context-node. For absolute XPath expressions the context-node is the document's root element. In the following we present the axes in the unabbreviated and, if available, in the abbreviated syntax, a description and an illustration. The context-node is highlighted with bold lines. The selected nodes are colored in light gray. The idea of this illustration is taken from [60].

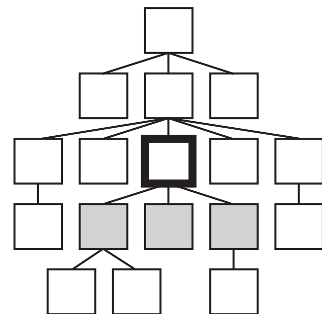
Self-Axis:
XPath: `self::`
Short: `.`

The self-axis is the simplest step: It just selects the context-node without further navigation.



Child-Axis:
XPath: `child::`
Short: `/`

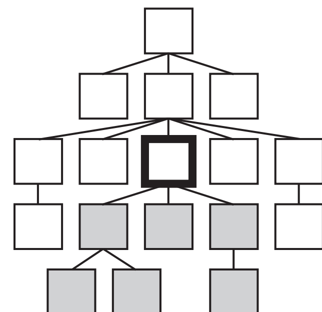
The most frequently used axis is the child-axis that contains the children of the context-node.



The `attribute` axis is comparable to the child-axis with the difference that the attributes of an element are selected. The XPath syntax for this axis is `attribute::` or abbreviated `@`.

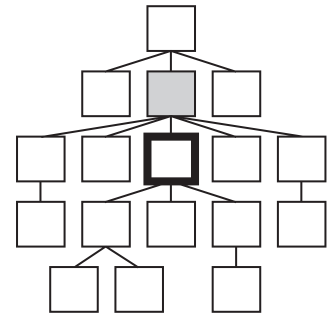
Descendant-Axis:
XPath: `descendant::`
Short: `//`

The descendant-axis contains the descendants of the context-node. A descendant is a child or a child of a child and so on.



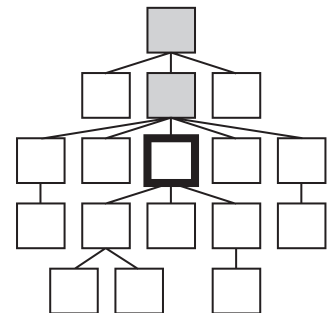
Parent-Axis:
XPath: `parent::`
Short: `..`

The counterpart of the child-axis is the parent-axis that contains the parent node of the context-node, if there is one



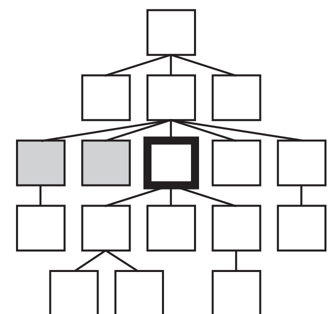
Ancestor-Axis:
XPath: `ancestor::`
Short: NA

The ancestor-axis contains the ancestors of the context-node; the ancestors of the context-node consist of the parent of the context-node and the parent's parent and so on; thus, the ancestor-axis will always include the root-node, unless the context-node is the root-node.



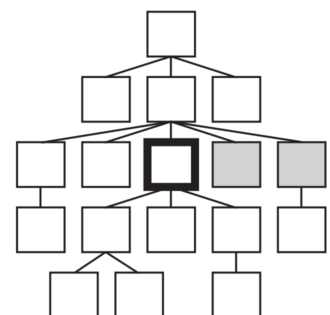
Preceding-Axis:
XPath: `preceding::`
Short: NA

The preceding-axis contains all nodes that appear before the context-node in document order, excluding any ancestors and excluding any attribute nodes.



Following-Axis:
XPath: `following::`
Short: NA

Similar to the preceding-axis the following-axis contains all nodes appearing after the context-node in document order, excluding any descendants and excluding attribute nodes.



In addition to these axes there are some more that consist of the union of two axes like *descendant-or-self* or *ancestor-or-self*.

Node Test

Every axis has a *principal node type*: it is the type of nodes that the axis can contain. For the attribute-axis, the principal node type is `attribute`. For most other axes, the principal node type is `element`.

A *node test* is true if and only if the type of the node is the principal node type and has a name equal to the specified name in the location step. For example, `child::item` selects the `item` children of the context-node. If the context-node has no `item` child element, it will select an empty set of nodes.
A node test `*` is true for any node of the principal node type. For example, `child::*` will select all element children of the context-node, and `attribute::*` will select all attributes of the context-node.
The node test `text()` is true for any text node. For example, `child::text()` will select all text nodes of the context-node. Similarly, the node test `comment()` is true for any comment-node.

Predicates

A *predicate* filters the node-set that is selected by the location step and drops all nodes that do not fulfill a condition. The general format of a predicate in a location step is `[condition]` where the square brackets are required, and condition represents some test resulting in a Boolean true or false value. For each node in the node-set to be filtered, an expression is evaluated with that node as the context-node. If the expression evaluates to true for that node, the node is kept in the node-set, otherwise it is dropped.

The conditions in predicates are expressed by value comparisons and by *node-set functions*. For instance, the predicate`[name='MP3 Player']` checks all `name` children if their content is equal to 'MP3 Player'. The context-node itself can be filtered with the `self` axis, e.g. `[.='MP3 Player']`.

Value comparisons are used to check for equality (=) and inequality ($\neq, <, \leq, >, \geq$) of textual and numeric values. Conditions can be put together with the Boolean operators AND, OR and NOT. String functions like `starts-with()` are provided for textual comparisons. With node-set functions like *last()* or *count* properties of the node-set an be checked. A complete list of the string functions and node-set functions can be found in [126] respectively [132].

Predicates can be used recursively in the expressions of a predicate, e.g.

$$/item[description[text = \text{'This item is ...'}]].$$

XPath expression Examples

In this section we describe XPath expressions by examples operating on the XMark sample document:

`/site`
This absolute path expression navigates to the root-node (/) and selects the one `site` element.

`//description`
Selects all elements that have the name `description` without regarding the leading path. Because of the descendant-axis (//) every node in the XML document is checked whether it has the requested name.

`/site/closed_auctions/closed_auction`
Selects all `closed_auction` elements that are child of `closed_auctions` elements that are children of the `site` element that is the root of the document. This path expression contains no predicates.

`//item/@id`
Selects the `id` attributes of every `item` element.

`//item[@id='item12']`
Selects the one `item` element that has an attribute `id` with the value `item12`.

`//item/@id[.='item12']`
Same as the previous one with the difference that the attribute itself is selected by a comparison with the node itself (.).

`/site/closed_auctions/closed_auction[price]`
This expression selects all `closed_auction` elements that have a child named `price`. The `closed_auction` elements must be children of the nodes selected by the expression `/site/closed_auctions` . The particular value of the price is not relevant, only the existence of a price child matters.

`/site/closed_auctions/closed_auction[price>'8.49']`
Similar to the previous expression with the difference that the value of the price must be greater than 8.49.

`/site/regions/*`
This expression selects all children of the nodes selected by `/site/regions`. Due to the $*$ the names of the selected elements is not regarded.

`//item/name[../location='Europe']`
Selects all `name`s of `item`s with the `item` having a `location` child with the given value. Please note the parent-axis that is required for this expression.

`//item[contains(name,'MP3')]`
The expression selects all `item` elements that have a `name` child whose textual content contains the string `MP3`.

`//item[count(incategory)>=3]`
Selects all `item` elements that have at least three `incategory` children.

2.3.2 XPath Query Types

In the context of databases it makes sense to classify XPath expressions into different query types. This allows us to compare the possibilities and the power of the various indexing approaches. We list the most common query types here:

- **Structural query** or **pure-path query:**
  Pure-path queries navigate in the data without paying attention to element values; e.g. $p_1 = /site/people$ selects all `people` elements below the `site` element.

- **Single-key query:**
  With a single-key query we can select elements fulfilling one condition. For instance, the query $p_2 = /site/people/person/[name =' x_1']$ selects all `person` elements of a given name with the value x_1. In the literature, this query type is sometimes called **predicate query** or **value query**. If the operators $>$ or $<$ are used for key value comparison we speak of a **range query**.

- **Multi-key query:**
  A multi-key query is a generalization of a single-key query allowing multiple key-value comparisons. The expression
  $p_3 = /site/regions/asia/item[name =' MP3\ Player'$ *and* $quantity > 10]$ queries the data for MP3 Player `items` with an available quantity greater than 10.

- **Wildcard query:**
  With the wildcard operator * we can select arbitrary elements without testing their labels. For instance, $p_4 = /site/regions/*/item$ selects all `item` elements without regarding their particular region.

- **Descendant query:**
  The *self-or-descendant* operator `//` is a generalization of a wildcard and selects elements by its name without paying attention to the leading path. The following expression $p_5 = //name$ selects all elements with the label `name` wherever they appear in the data.

- **Combinations:**
  It is possible to combine the different query types as the following self-explaining valid expressions show:
  $p_6 = //name[. =' x']$
  $p_7 = //item[name. =' x_2'$ *and* $quantity > 10]$

XPath 2.0

XPath 1.0 was introduced by the W3C in 1999. It is not a full programming language, in particular it is not Turing-complete because every location step and function terminates. This is very similar to the SQL language. XPath was designed to be embedded in a host language such as *XSLT 2.0* [129] or *XQuery* [128]. XPath 2.0 was presented as a working draft in 2001 and is a much more powerful language that operates on a much larger domain of data types. XPath 1.0 supports only the four expression types node-set, Boolean, number and string. This solution is very simple but is very limited when processing typed values such as dates. XPath 2.0 introduces support for the XML Schema [127] primitive types which immediately gives the user access to 19 simple types, including dates, years, months, etc. Besides, a number of functions and operators are provided for processing, casting and constructing these different data types.
Unlike XPath 1.0 that operates with unordered node-sets, XPath 2.0 supports ordered sequences of nodes. Sequences may contain duplicates. Sequences can be unified and intersected by new functions.
Further enhancements of XPath 2.0 are additional functions like `min()` and `max()` for numeric values and string matching with regular expressions. But as XPath 1.0 the successor is no full programming language. A major application of XPath 2.0 is the XML query language *XQuery*.

2.3.3 XPath Evaluation

For evaluating a particular XPath expression we need an XPath processor that operates on a given XML document. A straightforward (naive) implementation would evaluate the location steps of the XPath expression step by step. The resulting nodes of a previous location step are used as context-nodes for the ongoing location step. The pseudocode of the naive XPath evaluation algorithm is presented in figure 2.10.

```
1  NodeSet evalXPath (XPathExpr e, ContextNode cn){
2    if (e.isEmpty()) return cn;           //no more steps

4    LocationStep step = e.getNextStep();
5    e = e - step;                         //remove step from expression

7    Axis axis = step.getAxis();           //extract the axis
8    NodeTest nTest = step.getNodeTest();  //extract the node test
9    Predicate[] preds = step.getPreds();  //extract the predicates

11   NodeSet nodes = cn.eval(axis);        //evaluates the axis

13   for (n in nodes){
14     if (nText!=* && nTest !=n.label)    //check the nodes label
15       nodes.remove(n);                  //Node test failed
16   }

18   for (n in nodes){                     //Check each predicate
19     for (p in preds){                   //for each node
20       NodeSet nodes2 = evalXPath(p,n);
```

```
21        if (nodes2.length==0){
22          nodes.remove(n);                //predicate not fulfilled
23          break;
24        }
25      }
26    }
27    NodeSet result = new NodeSet();       //create empty set of nodes
28    for (n in nodes){
29      result.add(evalXPath(e,n));         //recursive call
30    }
31    return result;                        //remaining nodes
32  }
```

Figure 2.10: A naive evaluation algorithm for XPath expressions

The recursive method gets an XPath expression e and one context-node cn as input and returns a set of nodes that is selected by e when evaluated upon cn. If the path expression e is empty the context-node is the selected node. This is the termination case of the recursion. If e is not empty (it contains at least one location step) the leading location step is removed from e and stored in the variable $step$. This is done in lines 4 and 5.

In lines $7-9$ the axis, node text, and predicates are extracted from the location step. The axis is evaluated in line 11 leading to a set of nodes that have a relationship to the context-node specified by the axis. Each of these nodes must be checked if it fulfills the node test, i.e. if it has the requested name. This is done in lines $13-16$. If the node test is the wildcard operator * then every node is accepted.
Each of the remaining nodes in $nodes$ must fulfill all conditions expressed in the predicates. This selection is done in lines $18-26$. For reasons of simplicity the pseudocode concentrates on structural conditions and omits value comparisons and functions like `count`. A structural condition is an XPath expression that returns a non-empty set of nodes if evaluated on a node $\in nodes$. This is performed in line 20. If one node $\in nodes$ does not fulfill one condition it is removed from node $nodes$.
The remaining nodes $\in nodes$ have passed the node test and the predicates. Now they are used as context-node for the recursive call of the method with the tailing part of the original XPath expression e. The result of this evaluation is the final result set.

Gottlob, Koch and Pichler have shown in [34] that this naive algorithm may lead to an exponential runtime in the length of the query: Each application of a location step may result in a set of selected nodes with a size linear to the number of elements. If the method is called recursively with the previously selected nodes the algorithm ends up in consuming exponential time in the worst case. For example, let $t =$ `<a><\b><\b><\a>` be an XML data consisting of an a element with two b children. We query this XML data with a path expression p of the variable length

m. p consists of two leading `/a/b` location steps followed by $m-1$ expressions of the form `/../b`. The first part selects the two b elements. The remainder of p is executed for both b. Each `/../b` step returns the two b elements, so that the number of b elements is doubled on each execution of the `/../b` path expression. The m-th execution step returns 2^m b elements. Of course these are redundant repetitions of the original two b elements.

This naive approach is implemented in most XPath processors like *XALAN* [7] and *Microsoft Internet Explorer* [1] Gottlob, Koch and Pichler propose a more efficient implementation that uses polynomial runtime in the worst case. Details can be taken from [34, 35].

2.3.4 XQuery

XQuery is a functional programming language that uses XPath expressions for selecting nodes. More precisely, XQuery is a superset of XPath; this implies that each XPath expression is already a valid XQuery expression and will return the same result in both languages. In contrast to XPath that only selects nodes in a given document XQuery allows to transform the nodes and to create new structures with XML templates.

XQuery is a proposal of the W3C that is not standardized yet. The latest working draft was published in April 2005. However, many systems that use an XQuery implementation are realized. Examples include the *Microsoft SQL Server 2005 Express* with XQuery support [79] or the *Berkeley DB XML 2.0* [107], an embedded native XML database with support for XQuery 1.0.

Sequences

The basic construct of XQuery are *sequences*. All XQuery expressions and functions operate on one or multiple sequences and returns a new sequence. Sequences consist of items which can be atomic values or XML nodes. Both types can be mixed in a sequence. A sequence may not contain another sequence but can contain nested nodes. The following example shows how a sequence of values and elements is created syntactically in XQuery:

```
1  (1, <a><b/></a> , 2, 1, <c/>)
```

The items in a sequence are ordered and may appear more than once. Therefore, duplicates are allowed in a sequence. The concept of the sequence is different to the node-sets of XPath that have no ordered items and no duplicates.

[1]The Internet Explorer uses XPath in its embedded XSLT engine generating HTML pages from an XML document and a stylesheet.

FLWOR-Expressions

In contrast to XPath that only selects existing nodes, XQuery allows the construction of XML results that have a different structure than the original data. The central framework of XQuery are the *FLWOR*-expressions. The acronym FLOWR stands for `for-let-where-order by-return` and is pronounced like the word 'flower'. FLWOR-expressions support the binding on variables, the iteration over sequences and templates that define the structure of the resulting XML data. This way FLWOR can be compared to SQL in relational databases.

A FLWOR-expression creates a sequence of nodes and values by binding the results of a path expression to a variable by the use of `for` and `let` . The items of the sequence are filtered by the conditions expressed in the `where` clause. Optionally, the remaining items can be ordered with `order by` and returned with the `return` statement. This process is illustrated in figure 2.11.

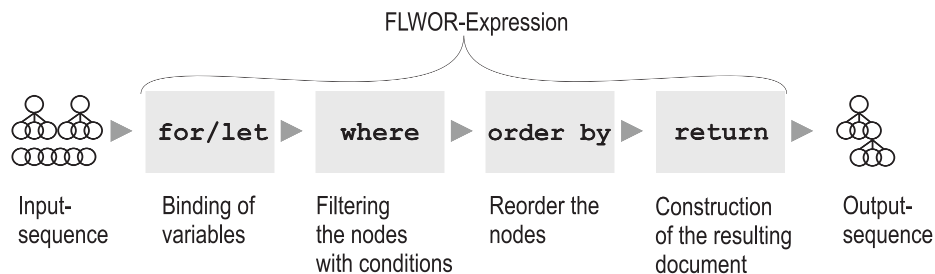


Figure 2.11: Construction of a FLWOR-Expression

The `let` and `for` clause bind the results of an XQuery expression (the result is a sequence) to a specific variable that can be addressed by further processing steps. The semantic of `let` and `for` is different: The `let` clause binds the sequence as a whole to the variable. Therefore, the further processing steps are performed once for the sequence.
The `for` clause binds all items of the sequence to the variable so that the further processing is performed with each item separately.

The different semantic of `let` and `for` is illustrated in a small example: The first XQuery expression binds a sequence of three element-nodes to the variable x. The return statement creates an `item` element and encloses the content of x.

```
1 let $x := (<name/>, <location/>, <description/>)
2 return <item> { $x } </item>
```

The resulting XML data is as follows:

```
1 <item>
2   <name/>
3   <location/>
4   <description/>
5 </item>
```

If we change the `let-` clause to a `for` clause each of the three nodes of the sequence are enclosed with the `item` element:

```
<item>
  <name/>
</item>
<item>
  <location/>
</item>
<item>
  <description/>
</item>
```

Analogously to predicates in XPath or the `WHERE` clause in SQL it is possible to restrict the result of an XQuery expression by using conditions that must be fulfilled by the sequence or its items. A `where` clause typically references the declared variables of `let` and `for`.
The following XQuery expression filters all items that are located in the United States and returns their names in an enclosing `<result>` tag:

```
 let $item := doc("auctions.xml")//item
 where $item/location ="United States"
 return <result> { $item/name } </result>
```

With XQuery it is possible to express more complex queries that produce an XML output from data that is distributed over the whole original data. The next XQuery expression creates compact information for each sold item consisting of the item's name, the name of the buyer, and the seller and the price. The data is distributed over three different parts of the original XML data: The name of the seller and buyer is in the `people` branch, the item's name can be found below the `regions` element and the final price is a value that can be found under the `closed_auctions` element.

```
let $data := doc("c:\xml\xmark\out.xml")
for $auction in $data//closed_auction
for $item in  $data//item
for $seller in $data//person
for $buyer in $data//person
where
        data($auction/seller/@person) = data($seller/@id) and
        data($auction/buyer/@person) = data($buyer/@id) and
        data($auction/itemref/@item) = data($item/@id)
return
 <summary>
        <item>
                {data($item/name)}
        </item>
        <seller>
                {data($seller/name)}
        </seller>
        <buyer>
                {data($buyer/name)}
        </buyer>
```

```
21        <price>
22                {data($auction /price)}
23        </price>
24 </summary>
```

Figure 2.12: XQuery expression for selecting information from a closed auction

In line 1 the root of the XML source is bound to the variable `data`. In lines $2-5$ separate variables are declared for each (closed) auction, item, buyer, and seller. The variables of the buyers and sellers are bound to the same XML nodes because a person can participate in an auction both as seller and as buyer. The `where` clause in lines $6-9$ performs a join-operation and removes all tuples that do not correspond.

The result of the expression is a sequence of `summary` elements that contains the most relevant information of a terminated auction. The `data()` function returns the value of an element or attribute. In this example, these are texts for the names and a float value for the price.

XQuery allows the definition of user-defined functions that can be called recursively. Therefore, XQuery is a Turing-complete programming language. XPath expressions always terminate (when executed on finite XML documents); this shows that XPath is not Turing-complete.

Because XQuery is more expressive than XPath it is very likely that XQuery will gain a significant relevance in the database context. Because XQuery relies strongly on XPath when addressing nodes the performant execution of XPath expressions is a major issue when optimizing XQuery. For instance, the XQuery expression of the last example has the computational complexity $O(n^4)$ because every combination of the four variables `auction`, `item`, `buyer`, and `seller` is analyzed whether it matches. Even for smaller documents this complexity is prohibitively high if the XPath expression is evaluated in a straightforward manner (see also 2.3.3). (With a more sophisticated XQuery expression it is possible to reduce the complexity up to $O(n^2)$, see section 10.1 in the appendix).

The popular and commercial XML editor *XMLSpy* [4] needs minutes of time and has a memory consumption of roughly 260 megabyte when evaluating the sample XQuery expression upon an XML document of only 124 kilobyte! This little experiment proves that optimization is a top-level issue in XML query languages. With an index that offers logarithmic time to find relevant nodes the complexity shrinks to $O(n \cdot \log(n))$. Even for a relatively small database with only 1000 auction elements the execution time with indexes is magnitudes faster than the straightforward implementation that has to select and check every combination in the `where` clause.

2.4 XUpdate: XML Modification Language

With XPath and XQuery we are able to select nodes and create an XML output with content from an XML document. Modifications in an existing document can be performed neither with XPath nor XQuery. With XSLT [129] we can transform one document A into a document B and perform some changes to B. But because XSLT works like a *stylesheet* the modifications do not take place in the original document. Furthermore, XSLT is Turing-complete and can be regarded as a cryptic but full programming language that does not allow comfortable ad-hoc queries like SQL for instance.

In this thesis *XUpdate* [134] is presented as a proposal for an XML modification language comparable to SQL. Although XUpdate relies strongly on XPath it was not designed by the W3C. The latest XUpdate Working Draft was published in 2000 by the XML:DB Initiative. No enhancement or updates of XUpdate were presented recently. Nonetheless, XUpdate is used by some XML database management systems like XIndice [8]. A successor project of XUpdate is *Lexus* [6], an update language also using XML syntax and XPath for navigation and selection.

In this thesis we give a brief and informal introduction to XUpdate - as one representative of an ad-hoc XML modification language. Basically, there are six different types of modifications that are presented in the following by using examples. For the reason of simplicity, elements and attributes are summarized to nodes.

Constructing and adding new nodes

XUpdate allows to add one or multiple new nodes to an existing document. The insert position can be *before* or *after* a designated node or the new node(s) may be appended to a list of existing nodes. This is shown in the next two examples.

```
1 <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2    <xupdate:append select="/site/categories" >
3       <xupdate:element name="category">Harddisks</xupdate:element>
4    </xupdate:append>
5 </xupdate:modifications>
```

Figure 2.13: XUpdate append operation

As one can see XUpdate expressions are well-formed XML documents themselves. The root-node is an element called `modifications` that belongs to the namespace xupdate. The second line selects the `/site/categories` element and adds one new `category` element with the value `Harddisks`. In this command an implicit element creation takes place.
The second example inserts a full XML fragment consisting of one item-node with an attribute and a complex content.

```
1  <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2    <xupdate:append select="/site/regions/europe" >
3      <xupdate:element name="item">
4        <xupdate:attribute name="id">item505</xupdate:attribute>
5        <name>DVD Recorder</name>
6        <location>Paris, France</location>
7        <quantity>1</quantity>
8        <payment>Cash</payment>
9        <description>
10         <text>This is a brand new DVD Recorder</text>
11       <description>
12     </xupdate:element>
13   </xupdate:append>
14 </xupdate:modifications>
```

Figure 2.14: XUpdate append operation with complex content

Deletion nodes

The deletion of nodes is performed using XUpdate's `remove` command. The node d to be deleted is selected with a normal XPath expression. Nodes that are descendants of d are deleted as well because without d they cannot be reached anymore from the document's root.

```
1  <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2    <xupdate:remove select="//item[@id = 'item101']"/>
3  </xupdate:modifications>
```

Figure 2.15: XUpdate remove operation

Renaming nodes

Nodes can be assigned a new name using the `rename` command. This modification does not affect the content of an element. In the example all `description` elements are renamed to `desc`.

```
1  <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2    <xupdate:rename select="//description" >
3      desc
4    </xupdate:rename>
5  </xupdate:modifications>
```

Figure 2.16: XUpdate rename operation

Changing the content of nodes

In contrast to the previous operation the `update` command changes the content of an element without affecting its name. The XUpdate expression in the example changes the price of all items that cost 14.99.

```
1 <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2   <xupdate:update select="//price[.='14.99']">
3             12.99
4   </xupdate:update>
5 </xupdate:modifications>
```

Figure 2.17: XUpdate update operation

Copying and moving nodes

XUpdate allows to bind the result of an XPath expression to a variable. In the following example a name of an `item` is bound to a variable i and the variable is inserted at another place. The value of the variable is addressed with $\$i$. This way one can copy a node in a document.

```
1 <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2   <xupdate:variable name="i" select="//item[@id␣=␣'item12']/name"/>
3   <xupdate:insert-after select="//item[@id␣=␣'item33']">
4     <xupdate:value-of select="$i"/>
5   </xupdate:insert-after>

7 </xupdate:modifications>
```

Figure 2.18: XUpdate operation for copying nodes

It is possible to move a node from one position to another if the node is bound to a variable and deleted before reinserted. This is shown in the next example.

```
1 <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2   <xupdate:variable name="i" select="//item[@id␣=␣'item12']/name"/>
3   <xupdate:remove select="//item[@id␣=␣'item12']/name"/>
4   <xupdate:insert-before select="//item[@id␣=␣'item33']">
5     <xupdate:value-of select="$i"/>
6   </xupdate:insert-before>

8 </xupdate:modifications>
```

Figure 2.19: XUpdate operation for moving nodes

2.5 XML and Databases

The emerging usage of XML technologies by applications in e-business, for instance, demands the close connection to database management systems because the latter provide a fast, robust, and application-independent way of storing and accessing data. For example, a clearing house interacting with member brokers using Web services is legally obliged to store the XML messages for non-repudiation. With the advent of Web services, it is quite common for business interactions to be processed via XML messages (e.g. SOAP). In such cases, the message is not only a transaction request but also a business entity like an order or an invoice. Such messages need to be preserved for many reasons including auditing, regulatory compliance, and non-repudiation. In addition, search facilities may be required to process the stored data. So, although XML was originally designed for data interchange, an increasing amount of XML data needs to be recorded and persistently stored.

In the context of databases the term *XML data* instead of *XML document* is more precise and generic. The term *XML document* fits better for document centric applications like content management systems, where documents are generated, transformed, and visually displayed.

Any software developer who designs an application requiring particular XML data to endure more than one run of the system has the problem of how to store the data persistently. As usual the file based approaches have many disadvantages: storing the XML data in its textual representation means parsing (to a DOM tree) whenever the data shall be processed. Even if minor changes in the DOM tree were performed the document has to be updated globally leading to performance degradation. A system crash in the meantime may lead to a total loss of the data if the document is not well-formed. And of course, storing XML data in the file system excludes the traditional database features like the support of a query language and multi-user and transactions controls.
Because of these reasons the persistency and database aspect is very important for applications dealing with larger amounts of XML data. In the following, a survey of database approaches for storing XML data is presented. In this thesis there is no basic difference between the storage of a multitude of small XML data versus the storage of one or few huge XML data.

2.5.1 XML in Conventional Relational Databases

A first approach is to try to store XML data in a relational database management system like *MySQL* or *PostgreSQL*. For this purpose, the XML structure and its content has to be reflected in a set of relational tables.
If the XML data is restricted by a DTD or an XML Schema one can define a corresponding relational schema and map the XML data into these custom relations.

The sample DTD of figure 2.2 could be transformed into the following three relations in a high normal form (3NF, BCNF):

ITEM

item_id	position	name	location	quantity
item0	1	Sinus MP3 Player	Lübeck, Germany	25
..	..	..	..	..
..	..	..	..	..
..	..	..	..	..

PAYMENT

payment_id	position	item_id	text_value
paym_0	1	item_0	Cash
Paym_1	2	item_0	Creditcard
..	..	..	..
..	..	..	..

TEXT

text_id	position	item_id	text_value	isBold	isEmph
text_0	1	item_0	This is a portable	false	false
text_1	2	item_0	MP3 Player	false	true
text_2	3	item_0	with 512 MB...	false	false
..	..	..	..	..	..

Figure 2.20: Relations representing the XML data

In general, each element that may have a text value is transformed to a column of a table. Elements with a 1:1 relationship (e.g. `<name>` and `<quantity>`) can be summarized in the same table. Elements with a 1:n relationship must be distributed over two tables if the second normal form shall not be harmed (no repetition of values). This applies to the `<payment>` element: they are stored in a second table that references the first one by the foreign key `item_id`. Because we have multiple tables with relations between them we have to add keys; for the first table we use the value of the attribute which was designed to be a key. But for the two other tables we have to assign artificial keys. Because the entries in a table have no order an extra column is needed to capture this information.
The third table expresses the mixed content of text values, `<bold>` and `<emph>` elements by using a flag. Especially in this table the order column is very important in order to keep the text structure consistent.

At first sight, this approach seems to be adequate for the storage of XML data but it has inherent and significant disadvantages: minor changes of the DTD (or XML Schema) may lead to major changes in the relational schema. For instance, if an element's cardinality in the DTD is changed from `?` to `+` or `*` the corresponding table has to be separated because this element may now appear more than once. Because the relational schema is now very different all existing SQL statements have become invalid and therefore must be adapted. Additionally, a relational schema cannot express exactly a DTD. The difference between `+` or `*` is not expressible.

An issue that is relevant for the performance of a database application is that the separation of one XML data to multiple tables implies several costly join-operations, in general. For instance, a relatively simple query that selects all items with a child `<payment>` that have the value `Cash` and a description with

the value `MP3 Player` the corresponding SQL statement already has three join-operations:

```
SELECT tab1.item_id FROM tab1, tab2, tab3
WHERE tab2.text_value = 'Cash' AND
      tab3.text_value = 'MP3 Player' AND
      tab2.item_id = tab1.item_id AND
      tab3.item_id = tab1.item_id;
```

For large XML data in a relational database the SQL based query evaluation may be very costly. Besides this, it is difficult and sometimes impossible to express queries that operate on the structure of an XML data in SQL. A comprehensive review of methods for XML-to-SQL query translation and their limitations is beyond the scope of this thesis and can be found in [67].

Conventional database management systems offer no direct support for XML. Therefore, the mapping to tables has to be done explicitly by the application. The same holds for the construction of XML data from values in the tables and for the generation of appropriate SQL statements.

These issues become even worse if no schema is provided for the XML data. Having no schema means that new elements may appear and disappear at any time during the databases lifetime. For some applications like research databases in bioinformatics this is a realistic constraint. Without a schema it is not possible to define relations with columns that correspond to the elements. Therefore, a more generic approach is required.
One way could be to express the parent-child relationship of all elements in one huge table. A simple generic table that is capable to keep an arbitrary XML document could be designed like this:
For each node (element, attribute, text node) of the XML data one entry is inserted into the table. The type of each node is kept in an extra column. The element's name and its content cannot be summarized in one row of the table because XML allows to express mixed content with an element having multiple text values as children. Again the position of the elements has to be recorded in an extra column. The null values are omitted to increase the readability.
This approach is very generic but leads to even more join-operations when processing a query. Each parent-child lookup requires a join-operation. Our sample query would look like this:

```
SELECT x.id FROM a,b,c,d,e tab
WHERE  a.value ='Cash'  AND         // a: text value of <payment> element
       a.parent_id = b.id AND
       b.name = 'payment' AND       // b: <payment> element
       b.parent_id = x.id AND       // x: requested <item> element
       c.value = 'MP3 Player' AND   // c: text value of <text> element
       c.parent_id = d.id AND
       d.name = 'text' AND          // d: <text> element
       d.parent_id = e.id AND
```

node_id	parent_id	position	name	value	type
1		1	item		element
2	1	1	id		attribute
3	2	1		item0	text
4	1	2	name		element
5	4	1		Sinus MP3 Player	text
6	1	3	location		element
7	6	1		Lübeck, Germany	text
8	1	4	quantity		element
9	8	1		25	text
10	1	5	payment		element
11	10	1		Cash	text
12	1	6	payment		element
13	12	1		Creditcard	text
14	1	7	description		element
15	14	1	text		element
16	15	1		This is a portable	text
17	15	2	emph		element
18	17	1		MP3 Player	text
19	15	3		with 512 MB...	text

Figure 2.21: Generic relations for storing arbitrary XML data

```
e.name = 'description' AND  // e: <description> element
e.parent_id = x.id;
```

For large data and more complex navigation steps the computation of the join-operations quickly becomes prohibitively expensive. In SQL transitive closure is not expressible, therefore we cannot navigate arbitrarily deep into the XML structure and check all descendants of a node.

Conventional relational database management systems do not support mechanisms to validate the XML data. Therefore, minor changes in the XML data (e.g. deleting the name of an item) may lead to invalid data. For validation, the XML data has to be reconstructed from the relations and validated apart; these are expensive operations so that validity cannot be guaranteed if a performant system is required.

2.5.2 XML in Extended Relational Databases

In recent years the major relational and object-relational database management systems (RDBMS) were extended and now include some support for XML data. Examples include the Microsoft SQL Server 2000 [80], Oracle XML DB [95], and the IBM *XML Extender* [73, 54] which is described here. In principle, two approaches are offered: XML data with a schema is shreddered and mapped into relations and afterwards processed with SQL statements. The XML Extender calls this method *side tables*. The only difference to the self-made tables of the previous section is that a user may define a *Document Access Definition* that automates the creation of suitable tables and the mapping process. The content in the side tables is accessed with SQL statements only. Compromising one can say that side

tables are more a convenience function than a new approach.

A second common approach of the extended RDBMS is to provide a column type for the exclusive storage of XML. In most cases the base of the XML type is a BLOB or a CLOB (Binary/Character Large Objects). In the XML Extender this column type is called *XMLClob* and can be seen more or less as a conventional CLOB which gets XML in its textual representation as input. When accessing the data an external parser gets the full data and transforms it to the memory resident representation. Because of the textual representation in the CLOB the parser always gets the full document even if only a small portion is processed afterwards. This approach may suit for a collection of very small XML data but fails if the data becomes larger. Additionally small modifications in the XML data lead to a full replacement of the textual representation in the CLOB. In other words the CLOB approach offers no granularity beside the full data. Therefore, standard database features like multi-user access on the same data with fine granular locks are impossible.

The main advantage of the CLOB approach is the full schema flexibility, as any XML document, regardless of its schema, can be stored. Because the XML data is not shreddered to relations but stored in its native form it is possible to apply XML query languages like XPath (see next section). The XML Extender uses stored procedures to invoke an external XPath processor which usually gets the entire XML document before processing. Again, this severely limits the size of the data and optimization possibilities. As a result, search and retrieval of XML data in a CLOB is relatively slow.

2.5.3 Native XML Database Management Systems

Summarizing one can say that conventional relations (aka side tables) and CLOBS are not a fully satisfying and universal solution to store and access XML data in a database.
In contrast to the relational approaches a new paradigm called *native XML database management systems* (XDBMS) was introduced in the last years. This approach is designed exclusively for the management of XML data. Examples include *Tamino* [105], Natix [29, 30], eXist [78], InfonyteDB [55], TIMBER [56], Xindice [8], and others. Native XDBMS store the XML data persistently in its tree-like structure, avoiding a costly transformation into relations and vice versa.

In contrast to the CLOB based storage that requires the entire data to be fetched from the database, native XDBMS offer a finer granularity. In general, single elements can be selected, changed, or added to the XML data without affecting the other parts. The parent-child relationship, sibling relationship, etc. is reflected by the internal storage structure. Simplified, one can think of a native XDBMS as a persistent DOM tree that resides on the harddisk. Indeed, the *PDOM* [28] was

an early step towards an XDBMS.
At least a native XDBMS provides an efficient storage of the XML data, support of a query language like XPath or XQuery, and an interface for programming languages like the DOM API [118] or XML:DB [133]. Advanced XDBMS may support an XML modification language like XUpdate, indexes to improve the query performance, and locking mechanisms to support multi-user interaction.

The technical functionality of native XDBMS is described here on the basis of Natix [29, 30]: Subtrees of the original XML data are stored together in a single (physical) record (and, hence, are clustered). Thereby, the inner structure of the subtrees is retained. The XML segment's interface allows to access an unordered set of trees. New nodes can be inserted as children or siblings of existing nodes, and any node (including its induced subtree) can be removed. The individual documents are represented as ordered trees with non-leaf-nodes labeled with a symbol taken from an alphabet. Elements are mapped one-to-one to tree nodes of the logical data model. Attributes are mapped to child-nodes of an additional attribute container child node, which is always the first child of the element-node the attributes belong to. Attributes, PCDATA, CDATA nodes and comments are stored as leaf-nodes.

Figure 2.22 illustrates how the logical tree (an XML data) is mapped to the physical tree. The relationships between elements are preserved. The image is taken from [30].

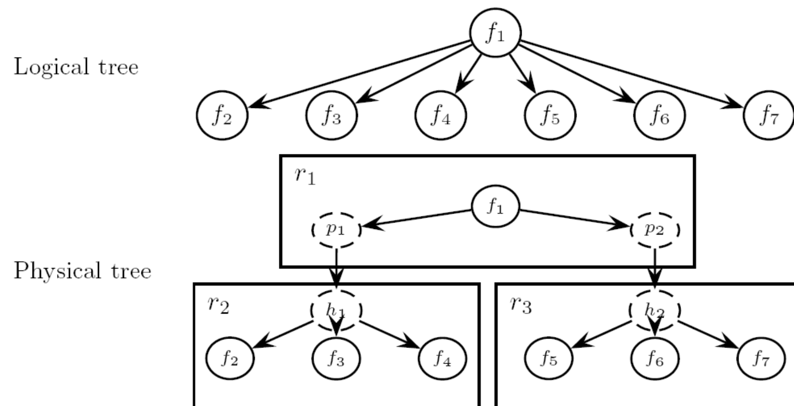


Figure 2.22: One possibility for the distribution of logical nodes onto records.

2.5.4 Hybrid Approaches

The example of figure 2.22 shows that the storage technique differs significantly from the relational approach. Therefore, existing implementations of relational DBMS can be reused (if at all) to a limited extent only. It is feared that native XDBMS will meet the fate of pure object-oriented databases that never gained

significant importance in real applications in industrial environments. However, the object-oriented paradigm lead to the extension of relational DBMS to object-relational DBMS supporting a multitude of the object-oriented features, like user-defined types and methods. The same may apply to native XDMBS so that future relational DBMS may provide a native support of XML storage and access.

A first prototype of a hybrid database can be seen in the *System RX* [11] from IBM Almaden Research Center and the IBM Silicon Valley Lab. SystemRX unifies a native XML storage and indexing and query processing technologies with the existing relational approaches of the DB2 implementation. The main idea is a new column type that keeps the XML data in a native form. Conceptually, the entry in each row can be interpreted as a reference to the root of a persistent DOM tree. Path expressions are evaluated directly over the native format; Therefore, the node of an XML document can be accessed and inserted without reading and manipulating the whole document. Like in every DBMS the persistent data (here the nodes) are distributed over multiple pages on the harddisk because the whole data cannot be kept in the main memory. For an accelerated access to relevant child-nodes of an element-node SystemRX provides an encoded XML storage and compression that is comparable to the Natix approach [29, 30].
Basically, because the XML data is stored in its native form any XML index can be used. SystemRX provides an indexing mechanism that is based on keys stored in a B+ tree. This approach is discussed more extensively in section 4.
The architecture is presented in figure 2.23. The figure is taken from [11].

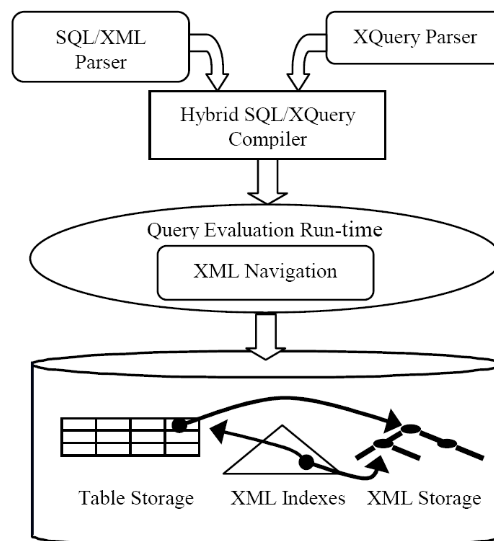


Figure 2.23: System Architecture of the hybrid DBMS System RX

With SystemRX existing SQL applications can be augmented with additional XML data. At this time is is too early to judge the impact of the System RX approach.

Chapter 3

Formal models for XML and XPath

In this chapter a formal model for XML data and XPath expressions is defined. The Document Object Model (DOM) of XML is a common model for XML documents that offers a multitude of node types - some of them, e.g. namespace nodes and comment nodes are used very rarely. Therefore, we restrict the DOM model without losing relevant parts of its expressiveness.
XPath expressions may have a very complex structure if multiple predicates are part of the expression. A special problem is that two XPath expressions that seem different at the first look may be semantically equal in the sense that the set of selected nodes is the same for all XML data. Therefore, a restricted model for XPath with the aspect of normal forms is the second step in this chapter.

3.1 A Model for XML Data

In this thesis a simplified DOM model is used to represent XML data. It is less expressive but suits well for the majority of use cases. Especially in the context of XML indexing the restricted model meets all requirements.

XML data are represented naturally in a tree-like data structure consisting of a set of *nodes*. We distinguish between two node types: element nodes and text nodes that are always children of an element node. An element may have one to many elements and text node children . Attributes are not directly supported in this model; instead they are transformed into equivalent elements. The model is defined as follows:

Definition 1 (Node)
A *node* n is a tuple

$$n = (id, type, label, value, parent, children)$$

with

id	A unique id that identifies this node.
$type$	the type of the node n. This can be `element` or `text`.
$label$	the name of the node n that corresponds to the XML tag if n is an element node. If n is a text node then the label is λ, the empty symbol. The type of label is *string*.
$value$	The string value of the node n if it is a text node. The value of an element node is λ.
$parent$	The parent node of n. Every node except the root node has exactly one parent. The root node has the parent λ
$children$	An arbitrary set of nodes that are children of n if n is an element. For all text nodes $children = \emptyset$.

The set of all nodes is denoted with N. Therefore, it holds that $n,\ parent \in N$ and $\forall m \in children$ it holds that $m \in N$.

Having a node n the function $n.parent$ returns the parent node. Analogously, $n.children$ returns the set of child nodes of n and $n.id$ returns the id that may be used for indexing the node.

The labels of all nodes are taken from the element alphabet Σ. The text values can be arbitrary strings without further restrictions. For a given node n its label can be determined by the use of the function $n.label$. □

Definition 2 (XML data t)
An XML data value t is a tuple (*nodes*, *root*) with *nodes* $\subset N$ the set of nodes and *root* $\in$ *nodes* the root node. It holds that *root*$.parent = \lambda$ and $\forall\ n \in$ *nodes* $\setminus$ {*root*} $n.parent \neq \lambda$. The set of all XML data is denoted with T. Because XML data may have arbitrary many elements T is an infinite set. □

Example 1 The small XML example

```
1 <item>
2   <name>
3     Sinus MP3 Player
4   </name>
5   <location>
6     Luebeck, Germany
7   </location>
8 </item>
```

has the following representation in the model:

$$\begin{aligned}
n_0 &= (0, element, \text{"}item\text{"}, \lambda, \lambda, \{n_1, n_3\}) \\
n_1 &= (1, element, \text{"}name\text{"}, \lambda, n_0, \{n_2\}) \\
n_2 &= (2, text, \lambda, \text{"}SinusMP3Player\text{"}, n_1, \emptyset) \\
n_3 &= (3, element, \text{"}location\text{"}, \lambda, n_0, \{n_4\}) \\
n_4 &= (4, text, \lambda, \text{"}Luebeck, Germany\text{"}, n_3, \emptyset)
\end{aligned}$$

The node n_o is the root node of the tree model; it is the only node with λ as parent node. □

Definition 3 (Function $t.nodes$)
The function $t.node : T \rightarrow \mathcal{P}(N)$ is called on an XML data $t \in T$ and returns all nodes of t. The function is defined as follows:

$$t.nodes ::= \{n \in N | n \text{ is a node of } t\}$$ □

Definition 4 (Function $t.root$)
For an XML data $t \in T$ the function $t.root : T \rightarrow N$ returns the root node of the tree model of t. The function is defined as follows:

$$t.root ::= n \text{ with } n \in t.nodes \text{ is the root node}$$ □

Based on this simple model for XML data we define a model for path expressions as follows:

3.2 A Model for Path Expressions

XPath is an essential fundamental of this thesis that is used at various places, especially for defining indexes and database operations. In order to analyze theoretical properties of XPath and to implement the index system we need an abstract and formal model of XPath that is more precise than the semantic description given in [126].
Because not all features of XPath are required in this thesis, a restricted subset of XPath is defined formally. This is done to keep the model and the implementation simple and manageable without loosing most of XPath's expressiveness.

The restrictions of the XPath fragment are as follows

- The attribute-axis is omitted because attributes are not supported by the underlying XML model. This is also done because attributes do not significantly increase the expressiveness of XML (see 2.1.2).

- We concentrate on the most important axes for navigating in an XML tree structure. These are `self (.)`, `child (/)`, `parent (..)`, and `descendant (//)`.
- Any node functions like `count` or `sum` are not part of the XPath fragment because these functions are not in the focus of the indexing approach.

The restrictions are not chosen by accident but are common in studies that analyze theoretical aspects of XPath. Examples include [81, 82, 90, 117, 106]. In general, it is possible to extend the fragment by attributes and more axes, although the expressiveness of the restricted model is high enough and supports the majority of real-world queries.

In the following we define three fragments of path expressions.

Definition 5 (Linear Path Expression)
A *linear path expression* p_l for a given alphabet Σ of element labels is defined by the following grammar:

$$\begin{aligned}
\mathbf{p_l} &::= \mathbf{p_{l_{rel}}} \mid /\mathbf{p_{l_{rel}}} \mid ../\mathbf{b} \\
\mathbf{b} &::= \mathbf{p_{l_{rel}}} \mid ../\mathbf{b} \\
\mathbf{p_{l_{rel}}} &::= \mathbf{p_{l_{rel}}}/\mathbf{p_{l_{rel}}} \mid \mathbf{p_{l_{rel}}}//\mathbf{p_{l_{rel}}} \mid * \mid n
\end{aligned}$$

with / denoting the child axis,// the descendant axis and .. the parent axis. $*$ is an arbitrary element and n a specific element with the label $n \in \Sigma$.
Linear path expressions may start with a leading / indicating an absolute path expression or arbitrary many leading parent axis (..) followed by a relative linear path expression $p_{l_{rel}}$. Within $p_{l_{rel}}$ the parent axis is not allowed, because it enables structural conditions[1]. The parent axis can only be applied at the beginning of relative linear path expressions.
The set of all linear path expressions is denoted with P_l . □

We call this set of path expressions linear because they are built by a sequence of location steps. Please note that linear path expressions do not have predicates.

Example 2 An example for a linear path expression could be:

$$../../person/name$$

The path expression $/a/../b$ is not a linear path expression because the parent axis is only allowed at the beginning. □

Based on the definition of linear path expressions we define *General path expressions* which may include structural conditions and value comparisons in qualifiers:

[1] e.g. $/a/b/../c \Leftrightarrow /a[b]/c$.

Definition 6 (General Path Expression $XP^{\{[],*,//\}}$)
General Path Expressions p extend P_l and are defined as follows:

$$\begin{aligned} \mathbf{p_g} &::= \mathbf{p_l} \mid \mathbf{p_l}[\mathbf{q}] \\ \mathbf{q} &::= \mathbf{p_l} \mid \mathbf{p_l}\ \mathbf{r}\ \mathbf{l} \mid \mathbf{q}\ \textit{and}\ \mathbf{q} \mid \mathbf{q}\ \textit{or}\ \mathbf{q} \\ \mathbf{r} &::= = \mid \neq \mid < \mid > \mid <= \mid >= \\ \mathbf{l} &::= \textit{string} \mid \textit{int} \mid \textit{float} \end{aligned}$$

□

where p_l is a linear path expression and *string*, *int* and *float* are typed literals. The set of all general path expressions is denoted with $XP^{\{[],*,//\}}$.

Example 3 An example for an expression of $XP^{\{[],*,//\}}$ could be:

$$/site/*/item[description\ \textit{and}\ name =' MP3\ Player']$$

The set $XP^{\{[],*,//\}}$ is close to the fragment used within the work of Miklau and Suciu [81, 82] with the difference that we allow to express conditions upon the content of a node using so-called key comparisons.

A path expression like $/a[b > 0\ \textit{and}\ b < 0]$ is satisfiable because the conditions may be fulfilled by two different b-children of a. The XPath semantic interprets the path expression as $(\exists b)(b > 0)$ *and* $(\exists b)(b < 0)$ and not as $(\exists b)(b > 0\ \textit{and}\ b < 0)$ as one might think first. The latter can be expressed in XPath with $/a[b[. > 0\ \textit{and}\ . < 0]]$ and is not satisfiable because the value of the same b element cannot be larger and smaller than 0 at the same time.

If we allow the *NOT* operator in qualifiers we get a more general class of path expressions.

Definition 7 (General Path Expression $XP^{\{[],*,//,NOT\}}$)
The XPath fragment $XP^{\{[],*,//,NOT\}}$ has an additional *NOT* operator in the rules for q. Therefore, it is possible to express negations inside a qualifier or key comparison. □

Example 4 An example for an expression of $XP^{\{[],*,//,NOT\}}$ could be:

$$/publications/article[NOT(publisher)]$$

selecting all article elements that have no `publisher`. □

It is obvious that: $P_l \subset XP^{\{[],*,//\}} \subset XP^{\{[],*,//,NOT\}}$.

Definition 8 (Absolute/relative path expressions)
A path expression (linear as well as general) is called *absolute* if it starts with a slash (/) that indicates that the document's root is the context node. Path expressions without a leading slash are called *relative* and operate on a previously selected context node.
The subset of relative general path expressions is denoted by $XP_{rel}^{\{[],*,//\}}$, whereas absolute general path expressions are denoted by $XP_{abs}^{\{[],*,//\}}$.
The same denotations are done for linear path expressions; relative respectively absolute linear path expressions are denoted by $P_{l_{rel}}$ respectively $P_{l_{abs}}$. □

Please note that only relative path expressions may start with leading parent axes.

Lemma 1 *Path expressions that use the descendant axis (//) as first location step (e.g.* `//item`*) cannot be expressed directly with the grammar for P_l. But an equivalent absolute path expression can be expressed with the* `self`*-axis (.) and a following descendant axis. The example would look like* `/.//item`*. In the remainder of this thesis we use the abbreviated syntax with the leading descendant axis.* □

Definition 9 (Function linearize)
An absolute general path expression can be linearized by function $linearize : XP_{abs}^{\{[],*,//\}} \rightarrow P_{l_{abs}}$ that transforms an absolute general path expression into a linear one by removing all keys and qualifiers. □

Example 5 An example for this function could be

$$linearize(\ /a[b]//c[d > 1]\ )\ =\ /a//c$$

When evaluating a path expression p on an XML data $t \in T$ a set of nodes is selected in t. This is done by the function $p(t)$.

Definition 10 (Path evaluation $p(t)$ with $t \in T$ an XML data)
$p(t)$:== $\{n \in t.nodes | n$ is selected when evaluating p on $t.root\}$ □

Analgously we are able to evaluate p on a single *node* n of an XML data t.

Definition 11 (Path evaluation $p(n)$ with $n \in t.nodes$ an XML node)
$p(n)$:== $\{n' \in t.nodes | n'$ is selected when evaluating p on n, with $n \in t\}$ □

We refer to section 2.3.3 for details on how a path expression is evaluated.

A path expression p can be applied to any XML data $t \in T$. But only for some XML data p selects one or multiple nodes of t. The set of XML data with p selecting a non-empty set of nodes is called the *model* of p *Mod*(p).

Definition 12 (Model $Mod(p)$)

$$Mod(p) ::= \left\{t \in T \mid p(t) \neq \emptyset\right\}$$

It is obvious that *Mod*$(p) \subseteq T$.

Lemma 2 *Mod*$(p) = \emptyset$ *or Mod*(p) *is infinite.*
If p is not satisfiable (e.g. [1]) Mod$(p) = \emptyset$. On the other hand, if we have $t_1 \in$ Mod(p) we are able to create t_2 by adding an arbitrary node (element) to t_1. Then t_2 will also be $\in$ Mod(p) because p will select the same nodes in t_2 as in t_1. Therefore, the set Mod(p) is infinite (without proof). □

An algorithm that decides whether an XML data t is $\in$ *Mod*(p) is introduced in section 7.2.2.

In order to process path expressions with automaton theory, in section 7.2.2 we need access to the nodes of a path expression.

Definition 13 (Function *nodes*$(p) : P_l \rightarrow N_{P_l}$)
The function *nodes*(p) returns a set consisting of all node tests of the linear path expression p. The set of all path expression nodes is denoted with N_{P_l}. □

Analogously to nodes in the XML model the functions

- $p.root$ returns the root node of a linear path expression p;
- $n.label$ with $n \in N_{P_l}$ returns the label of n;
- $n.children$ returns the one child node of n if it exists or $\emptyset$ otherwise;
- $n.descendant$ returns the one descendant node of n if it exists or $\emptyset$ otherwise;

Example 6 $p = /a//b/*/b//a$ with $p \in P_l$ has the following properties:

$$\begin{aligned} \textit{nodes}(p) &= \{a_1, a_2, b_1, b_2, *_1\} \\ root(p) &= a_1 \\ a_1.label &= a \\ a_2.label &= a \\ b_1.label &= b \\ b_2.label &= b \end{aligned}$$

[1] p=//a[b and not(b)]

$$\begin{aligned} *_1.label &= * \\ a_1.children &= \emptyset \\ a_1.descendant &= b_1 \\ b_1.children &= *_1 \\ b_1.descendant &= \emptyset \\ b_2.children &= \emptyset \\ b_2.descendant &= a_2 \end{aligned}$$

□

Definition 14 (Alphabet $\Sigma(p)$)
The set $\Sigma(p)$ consists of all labels of nodes in a linear path expression. The wildcard symbol is not part of $\Sigma(p)$.

$$\Sigma(p) ::= \{n.label | n \in nodes(p),\ n.label \neq *\}$$

Example 7 For $p = /a//b/*/b//a$ with $p \in P_l$, it holds that $\Sigma(p) = \{a, b\}$. □

$\Sigma(p)$ is determined by a simple extraction function.

3.2.1 Regular Expressions in Formal Languages

Linear path expressions without leading parent axes can be transformed into a *regular expression* of formal languages. A regular expression, often called a pattern, is a string that describes a set of strings without listing them. Instead, it defines certain syntax rules that match the string.

We create a regular expression p_{reg} from a linear path expression $p \in P_l$ by a renaming procedure as follows: each node test with a specific label remains unchanged; a node test with a wildcard (*) is transformed into a disjunction of symbols. Because the wildcard matches any element label the disjunction would contain infinitely many symbols.
We avoid this disjunction by the symbols used in the path expression by an extra symbol called α that is not in Σ. When evaluating the path expression on an XML data t, α will match all elements with a label $\notin \Sigma(p)$.

The descendant axis (//) is treated in a similar way: Basically, // matches any node below a given node without paying attention to its label. Therefore, the descendant axis is replaced analogously by a choice of symbols $\in \Sigma \cup \{\alpha\}$. Additionally, with the Kleene closure it is possible to skip several elements. We denote the Kleene closure by the symbol $*_k$ in order to differentiate it from the wildcard symbol (*).

The set of all regular expressions that can be built with the symbol alphabet Σ and α is denoted by $REG_{\Sigma,\alpha}$

Example 8 The linear path expression $p = /a//b/c/*/d$ with $\Sigma(p) = \{a, b, c, d\}$ leads to the following regular path expression:

$$a\ (a|b|c|d|\alpha)^{*_k}\ b\ c\ (a|b|c|d|\alpha)\ d$$

Definition 15 Any regular expression $r \in REG_{\Sigma,\alpha}$ defines a language L_r consisting of all words that match the pattern of the path expression. □

We use regular path expressions when building finite automata in section 7.2.2.

The reverse transformation from regular expressions to linear path expressions is not always possible: for instance, for the regular expressions $a(b|c)d$ there is no corresponding linear path expression because the choice $(b|c)$ cannot be expressed. With the wildcard (*) a path expression would accept all symbols and not only b and c. With a normal node test the choice is not expressible.

Even in full XPath 1.0 $a(b|c)d$ cannot be expressed in a simple way. A workaround may use a node function called `name` and looks like $/a/*[name(.) ='\ b'$ *or* $name(.) ='\ c']/d$.
XPath 2.0 supports a choice in node tests. The corresponding path expression is $/a/(b|c)/d$.

3.2.2 Tailing Predicates and Normalization

The grammar for general path expressions allows only one predicate that is always the last part of a path expression. This raises the question whether these *tailing predicates* are expressive enough for realistic queries. With a small example we show how a more general path expression with multiple predicates can be normalized to an equivalent path expression with one tailing predicate:

$$e_1 = //a/b[//c =' x']/d[/e =' y']/f$$

The XPath expression e_1 returns all elements with the label f that are child of an element d having another child e that has the value y. In addition, the element d must be the child of an element b that is ancestor of a c with value x. Finally, b is child of an element a that appears somewhere in the XML data. The same semantic can be expressed with path expression e_2.

$$e_2 = //a/b/d/f[../e =' y'\ \textit{and}\ ../..//c =' x']$$

The normalization process combines all predicates by using AND statements. The resulting predicate (with multiple value comparisons) is moved to the end while

adjusting the leading paths. The normalization of XPath expressions relies on the equivalence of XPath expressions; details can be found in [81, 82, 92, 112].
XPath in general offers functions that consider the order of elements (e.g. position(), first(), last()). Expressions containing such a function cannot be normalized as described above because the order information would be lost. Our model for path expressions does not reflect order functions, therefore this restriction can be ignored.

Chapter 4

Introduction to Recent Approaches in XML Indexing

In this section we classify and describe recent approaches indexing XML and semistructured data. Some approaches were published before XML gained the current importance and generally operate on semistructured data. We transferred these approaches to XML.

The basic idea of an index for semistructured data and XML is to accelerate the execution of path expressions, for instance XPath. The more complex XQuery expressions benefit from an index, too, because XQuery relies on the execution of XPath expressions for addressing the nodes of the sequences.

All indexing approaches have in common that they try to avoid the linear inspection of XML nodes when performing node tests or checking predicates. For instance, when evaluating the XPath expression `//item[/name='MP3 Player']` every element is treated as if it has the label `item` or not. Second, for each `item` element all children are checked whether they have the label `name`. Third, for all `name` elements the corresponding text value is compared with the given string. For larger databases this evaluation method leads to unacceptable processing times.

Although all indexing approaches have the same goal, their methodology, the internal data structures, and the query processing vary significantly. For this reason we establish some criteria in order to classify and compare the related work on XML indexing.

Some index approaches index the structure of the XML data without regarding the value of elements or attributes. These approaches are called *structural indexes* or *pure-path indexes*. On the other hand some indexes cover only the value of elements and attributes without reflecting the leading path to these values; these approaches are called *value indexes*. Advanced approaches cover both structure and values leading to an acceleration of more general and realistic path expres-

sions; these approaches are itemized as *hybrid indexes*.

The *selectivity* on an index states whether it always covers the whole XML data or is tunable for specific and user-defined fragments. A non-selective index has to be updated whenever the original data is modified. A selective index consumes less space and can be tuned for the typical usage of the database leading to less update operations. A relational index is selective because it is defined upon a table and a column.

Key-queries may return an element which differs from the key-element(s) that is/are used for the value comparison. For instance, the general path expression $//item[quantity > x_1]$ returns `item` elements whereas the value used for the comparison belongs to a `quantity` element. The majority of index approaches can only return the indexed key-element leading to additional expenses for navigation if the return element is different. For large paths between key and the return value this may add significant costs for the query processor. Some approaches like *KeyX* and the *Refined Path* from the *Index Fabric* are able to directly return the requested element without further navigation in the XML data.

In order to explain and illustrate the different indexing approaches in a quickly understandable manner we use some XML data taken from the XMark project and generate a specific index for each approach to be evaluated. The sample data consists of two items, one located in Asia and two in Europe. The items have different child elements describing the properties of the item. Additionally, the sample data contains two persons with their addresses. The textual representation of the sample data is presented in figure 4.1.

```
1  <site>
2    <regions>
3      <asia>
4        <item id="item1">
5          <location>Singapur</location>
6          <quantity>2</quantity>
7          <name>512 MB USB Stick</name>
8          <payment>Money order</payment>
9          <payment>Cash</payment>
10       </item>
11     </asia>
12     <europe>
13       <item id="item3">
14         <location>Hamburg</location>
15         <quantity>1</quantity>
16         <name>Beuys Sculpture </name>
17       </item>
18       <item id="item4">
19         <location>Paris</location>
20         <quantity>2</quantity>
21         <name>Louvre Tickets</name>
22         <payment>Cash</payment>
23       </item>
24     </europe>
25   </regions>
```

```
26    <people>
27      <person id="person0">
28        <name>Huei Demke</name>
29        <address>
30          <street>95 Grinter St</street>
31           <city>Luebeck</city>
32           <country>Germany</country>
33         </address>
34       </person>
35       <person id="person1">
36         <name>Daishiro Juric</name>
37         <address>
38           <street>5 Pinet St</street>
39           <city>Athens</city>
40           <country>Greece</country>
41         </address>
42       </person>
43     </people>
44   </site>
```

Figure 4.1: XMark fragment to illustrate the indexing approaches

The corresponding DOM tree is presented in figure 4.2 showing each element and attribute as a node with an individual ID. The text values are ommitted for space reasons.

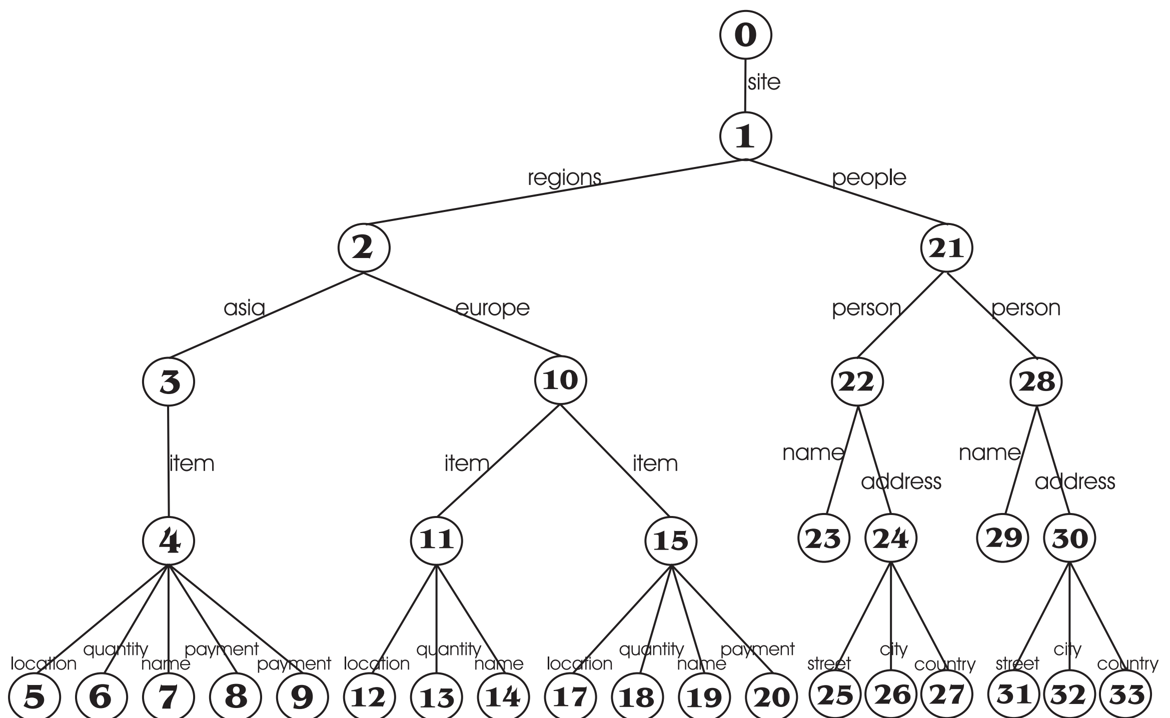


Figure 4.2: XML sample data represented as a DOM-tree

In this section we introduce and compare recent approaches for indexing XML data.

Approaches that use an underlying relational DBMS to store and index XML (e.g. [10]) have in common that they can reuse existing and performant implementations of the relational world. However, XML queries have to be mapped to SQL queries leading to many expensive join operations if a multi-key query is executed. The problems of rewriting XML to SQL are discussed in [67]. Therefore, we concentrate on new indexing approaches for the native storage of XML. The following survey of related work begins with techniques which accelerate structural queries, proceeds with concepts dealing with value queries and ends with hybrid approaches that support queries containing structural and value conditions.

4.1 Structural Indexes

Structural indexes reflect the structure of the XML data with its element labels. The values of the elements are not kept in the index structure. Therefore, only pure path expressions are supported.

4.1.1 The Strong DataGuide

One common approach to index the structure of semistructured data are so-called *Structural summaries* that summarize nodes of the original XML data to extents. One early approach is the *Strong DataGuide* from the *LORE* project [37, 76, 77] providing a general index structure to accelerate structural path expressions starting at the document's root. The DataGuide itself is a tree structure. Elements in the XML data that are reached by the same path expression are summarized in one node of the Strong DataGuide. This node is called *extent*. The parent-child relationship of nodes in the XML data is reflected in the Strong DataGuide.

Formally, the creation of extents relies on a symmetric binary relation which is called *bisimulation*; symbolized by $\approx$. The bisimulation holds for two nodes $u \approx v$ in the XML data if and only if u and v have the same label and secondly, if par_u is the parent node of u and par_v is the parent node of v, then $par_u \approx par_v$. Two nodes are called *bisimilar* if the bisimulation relation holds.
An extent of the Strong DataGuide collects all nodes that are bisimilar. Bisimilarity in that context implies , that the nodes in one extent cannot be distinguished by absolute linear path expressions without the descendant axis (//) and the wildcard node test (*).

Figure 4.3 shows the Strong DataGuide for the given XML sample.
Referring the example, the `name` elements of all persons are stored together in one extent because they are selected by the linear path expression

$$/site/people/person/name.$$

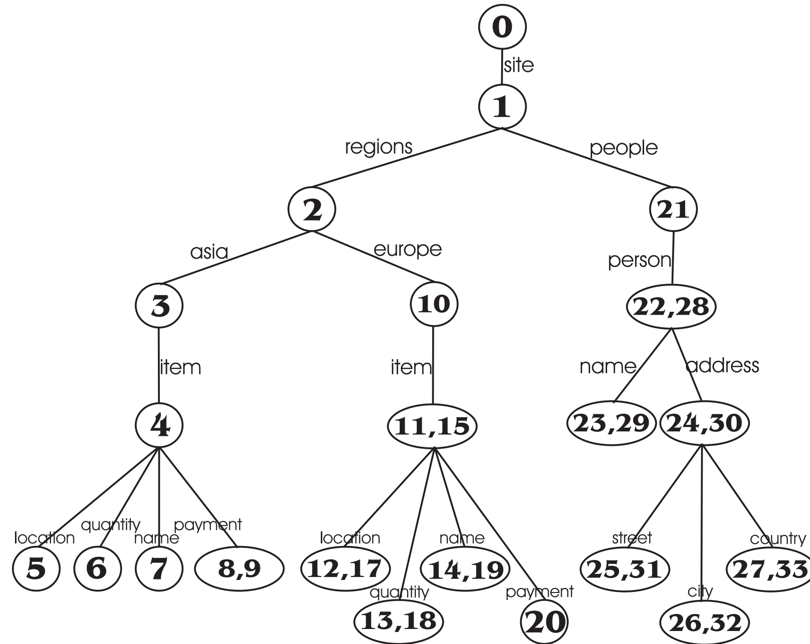


Figure 4.3: The Strong DataGuide

This path expression can now be executed by navigating in the Strong DataGuide to the relevant extent. In contrast to an XML document / DOM-tree where one element may have several children with the same name, an extent in the Strong DataGuide may have only one outgoing edge per label. Therefore, the evaluation of path expressions is done significantly faster, because each child axis leads to an extent containing references to many elements in the XML data.
For path expressions with the child axis only, the time to find the corresponding extent is linear to the size of the query. Path expressions containing the descendant axis or a node test with a wildcard lead to full evaluation of the Strong Dataguide because the results of this query may be distributed over multiple extents. For instance, the expression `//name` requires all extents of the DataGuide to be checked if their label conforms to `name`. Those path expressions cannot be executed efficiently with a Strong DataGuide.

The creation of a DataGuide can be compared to the transformation of a nondeterministic finite state machine to a deterministic one by the fusion of equivalent states.
For XML data with references between nodes the underlying structure can be interpreted as a graph and no more as a tree. For this data the Strong DataGuide can become larger than the original data leading to higher storage costs.

We compare the performance of the Strong DataGuide by measurements for several queries of different types in [42].

4.1.2 1-Index

For tree-like XML data containing no references between nodes the *1-Index*[83] is equivalent to the Strong DataGuide. If references are contained in the XML data or if we face general semistructured data the Strong DataGuide may lead to a redundant storage of nodes that are reachable by several paths due to the existence of references.
This problem is faced by the 1-Index which summarizes these nodes in a common extent. Because the XMark data contains no explicit references we have to illustrate the differences between the Strong DataGuide and the 1-Index by some other data presented in figure 4.4. The figure is taken from [83].

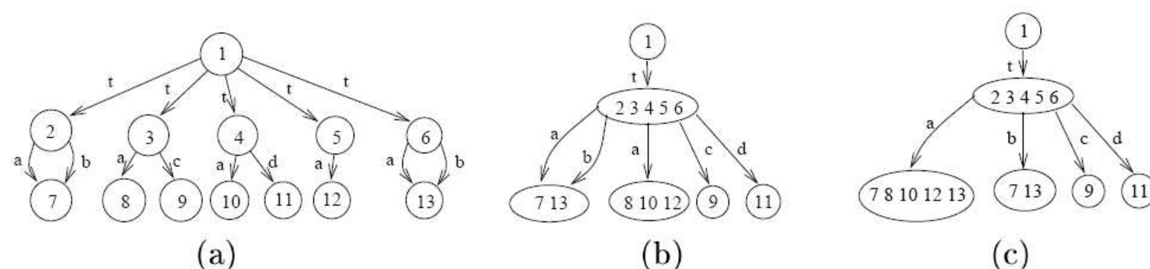


Figure 4.4: The XML data (a), its 1-index (b) and its Strong DataGuide (c)

The sample data does not distinguish between parent-child relationships and references. The node with the id 7 in the XML data is reached by the path expressions `/t/a` and `/t/b` from the root node. Because these two path expressions are different the Strong DataGuide puts the node 7 in two separate extents. In particular, the Strong DataGuide for the whole XML data contains more nodes than the original data.
The 1-Index removes duplicates by combining these different extents into one with multiple ingoing edges. Therefore, a 1-Index is not always a tree-structure, it becomes a graph itself. When processing a path expression on a 1-Index it may happen that multiple extents are selected and must be unified in a further step. An example may be the path expressions `/t/a`.
Both the 1-Index and the Strong DataGuides are designed for absolute path expressions starting at the document's root. Relative path expressions or path expressions starting with the descendant axis (//) lead to an evaluation of the full data structure which eliminates most performance advantages of the index.

4.1.3 2-Index

The *2-Index* [83] supports absolute and relative path expressions between nodes that are connected by parent-child relationships. The 2-Index is also based on extents that contain nodes that are reachable by similar path expressions.
For two nodes u and v we define a set of paths as follows:

$$L_{(u,v)} = \{w | w \text{ is a path from } u \text{ to } v\}$$

Two pairs of nodes are defined as equivalent $(u, v) \equiv (u', v')$ if $L_{(u,v)} = L_{(u',v')}$ meaning that both node pairs share the same connecting path. In contrast to the 1-Index and the Strong DataGuide this path must not begin at the root node. Therefore, the 2-Index can be applied on relative path expressions starting at an arbitrary node in the data. Figure 4.5 shows the 2-Index for the sample XMark data omitting the branch with the items for space reasons.

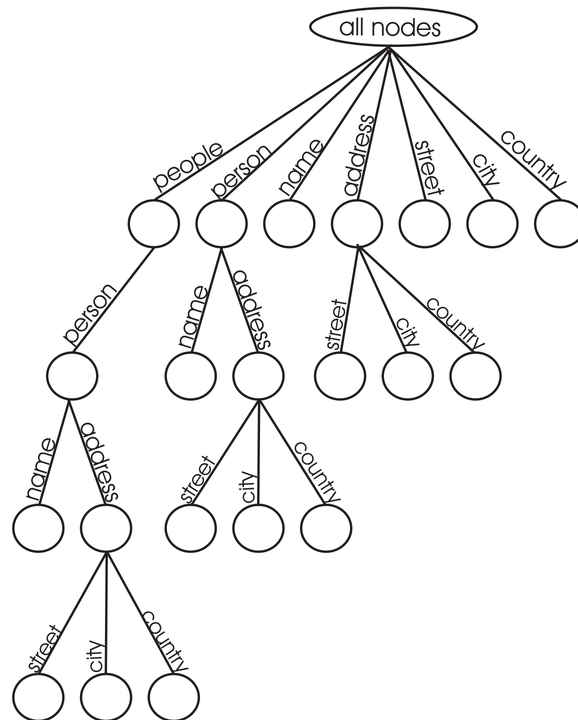


Figure 4.5: The 2-Index for the sample data.

All nodes of the data are in the root extent so that absolute as well as relative path expressions start at the root of the 2-Index. As one can see the 2-Index may become larger than the original data leading to higher storage costs. In the worst case the memory consumption is quadratic in the size of the original data.

The 1-Index, 2-Index, and the Strong DataGuides differ significantly from indexes in RDBMS: Instead of accelerating specific queries very efficiently they try to improve the evaluation of path expressions in general. In contrast to indexes in RDBMS where indexes are selected by the database administrator, these structural summaries are permanently enabled and thus not selective.
For XML data that is often modified (for instance the bidding price of an auction changes several times during the auction's lifetime) a structural summary must be updated whenever a structural modification takes place. This may lead

to significant performance loss limiting the profit of the index. One solution can be selective indexing approaches that index only relevant parts of the XML data and ignore parts that are frequently updated. Of course, such an approach does not cover all queries with the same performance. In the following, we introduce selective structural summaries.

4.1.4 T-Index

The *T-Index* [83] is a generalization and specialization of the 1-Index and the 2-Index. It is a generalization because the 1-Index and the 2-Index are special cases of the T-Index. This means, that every 1-Index and 2-Index is a T-Index, too. On the other hand, the T-Index is a specialization because it is tailored to answer specific queries whereas the 1-Index and the 2-Index can be used to process any path expressions.

The main idea of the T-Index is to establish a structural summary that only covers path expressions fulfilling one specific *template*. The template describes the structure of the path expressions by node tests and placeholders. If the indexing system shall support multiple path expressions with different structures that cannot be summarized with one common template, we need several T-Indexes, one for each template.

A T-Index, in general, supports path expressions with a template

$$T_0x_0T_1x_1...T_kx_k.$$

For each $0 \leq i \leq k$ T_i is either a node test or a linear path expression $\in P_l$ of arbitrary length symbolized by **P**. **P** is equivalent to the descendant axis (//) in the XPath syntax. The variables $x_0...x_k$ represent nodes in the data.
For instance, the template `/site/people` x_0 `person` x_1 selects all `person` nodes that are reached by the path `/site/people/person`. These nodes are bound to the variable x_1.
The template `/site/people/person` x_0 **P** x_1 selects all nodes that are located under the nodes selected by `/site/people/person` and bound to x_0.
Because **P** can be an arbitrary path expression consisting of multiple location steps all descendants are selected. The corresponding path expression in XPath syntax is `/site/people/person//*` with `//` being the descendant axis.

We omit some more technical details from the T-Indexes that are less relevant for understanding how T-Indexes work, such as additional nodes needed to separate the path expressions. These details can be found in [83]. The T-Index shown in figure 4.6 is based on the example template from above. There is a single edge S_o representing the linear path expression `/site/people/person`, so that the node variable x_0 is bound to the nodes with the ids 22 and 28. Below these two

nodes all possible paths are located in a way similar to the 1-Index or 2-Index. Figure 4.3 shows the Strong DataGuide for the given XML sample.

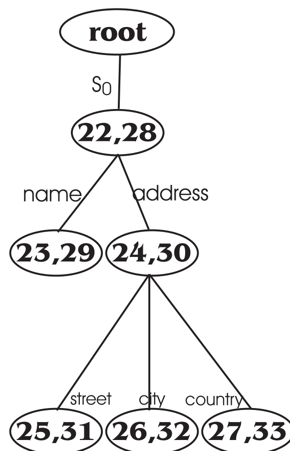


Figure 4.6: A T-Index for the sample data

Like the Strong DataGuide the T-Index is traversed top-down when evaluating the query. Because of the non-deterministic structure backtracking may be required. When evaluating a path expression p it must first be checked if p fulfills the template. If not, the T-Index cannot be used. Having more than one T-Index the appropriate one must be determined. This relies on the query rewriting problem. Intuitively, a query with a prefix matching a template can be rewritten, so that the first part (the prefix) can be evaluated by a T-Index. The remainder must be looked up in the original XML data (or another index if available and suitable).

The work [83] does not discuss the updatability of T-Indexes in detail. Because the T-Index does not pay attention to values in the XML data, only structural changes must be reflected by the T-Index. There seems to be no incremental update possibility but just the approach of marking some parts of the T-Index as out-dated. After a multitude of some changes a complete reconstruction of the index is required in order not to mark the whole index as out-dated.

4.1.5 Apex

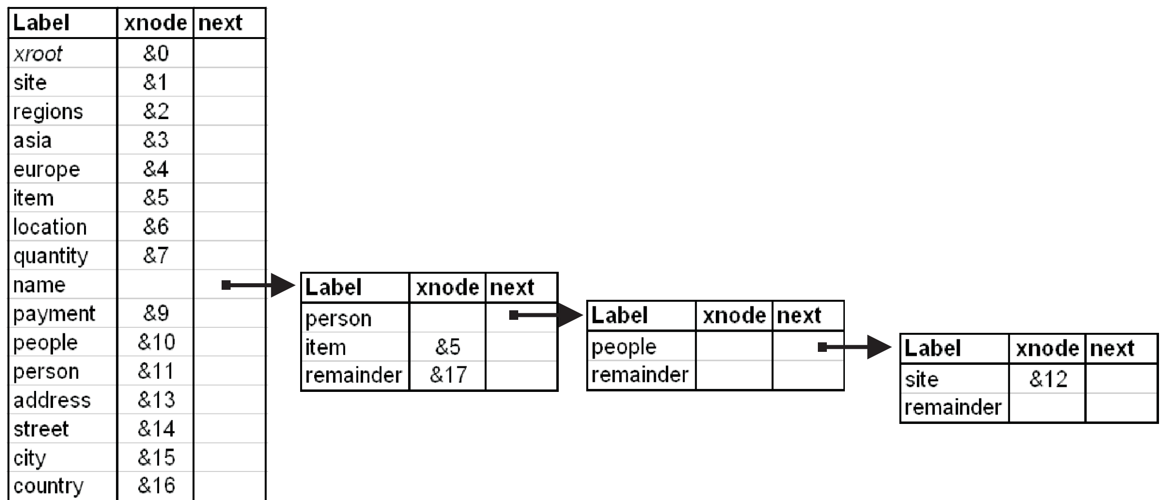
The problem of large structural summaries designed to support all queries motivates the *Adaptive Path Index (APEX)* [19] which has three goals: First and analog to the T-Index, frequently occurring queries should be accelerated more than general queries. Secondly, path expressions starting with the descendant axis (//) that lead to a full evaluation of most structural summaries should be supported in an effective manner. Third and different to the T-Index, APEX can be updated incrementally according to the changes of query workloads.

Apex was introduced by Chung et al. and consists of two structures, a graph G_{APEX} with the structural summary storing references to elements, and a tree of hashtables H_{APEX} representing the incoming path to nodes of G_{APEX}. Each node of G_{APEX} corresponds to one extent of nodes of the XML data. H_{APEX} is a specific index optimized for frequent queries that must be selected before building the index. The selection may be made by a database administrator who knows or estimates the typical usage of the database.
Because it is difficult or even impossible to build an index that is efficient for all possible path expressions, APEX tries to change its structure according to the frequently used path expression. In order to determine frequently used path expressions it is assumed that they are kept by the database system.
The order of the hashtables is inverse to the order of nodes in a DataGuide. This means that a query is evaluated from the tail to the head. This is the reason why APEX can easily support descendant queries. The APEX index for the sample data is illustrated in figure 4.7.

In this picture each extent of G_{APEX} is identified by an id; for instance, &12 references to the extent containing all nodes selected by the path expression `/site/people/person/name`. The hash structure H_{APEX} in this example is optimized for the three path expressions `//name`, `//item/name` and `/site/people /person/name`. Because path expressions are evaluated in reverse order the APEX query processor starts with the last node test of the path expressions. Path expressions of length one with a leading descendant operator are supported very efficiently: For instance, `//payment` is looked up in the first hashtable that references immediately to the extent &9. Path expressions with a length greater than one are also supported efficiently if they are assigned to be frequent and therefore organized in the hashtables' structure. For the frequent path expression `//item/name` the last node test `name` is looked up in the first hashtable that references to a second hashtable, because more than one extent is affected. For the path expression `//item/name` we now have to look for the `item` entry that references to the extent &5. If the frequent path expression `/site/people/person/name` is executed we have to follow the `person` entry that references to a third hashtable. The extent of the frequent path expression `//name` (&17) is stored as *remainder* in the second hashtable because no other element has an outgoing path to the `name` element.

The publication [19] does not explain whether and how APEX supports queries containing node tests with a wildcard (*). In general, a wildcard implies that all children of one extent must be processed leading to performance degradation. For instance, the path expression `/site/regions/*/item` querying all items independently from their continent cannot be supported by the hashtable approach.

We compare the performance of the APEX by measurements for several queries of different types in [42].



Label	xnode	next
xroot	&0	
site	&1	
regions	&2	
asia	&3	
europe	&4	
item	&5	
location	&6	
quantity	&7	
name		→
payment	&9	
people	&10	
person	&11	
address	&13	
street	&14	
city	&15	
country	&16	

Label	xnode	next
person		→
item	&5	
remainder	&17	

Label	xnode	next
people		→
remainder		

Label	xnode	next
site	&12	
remainder		

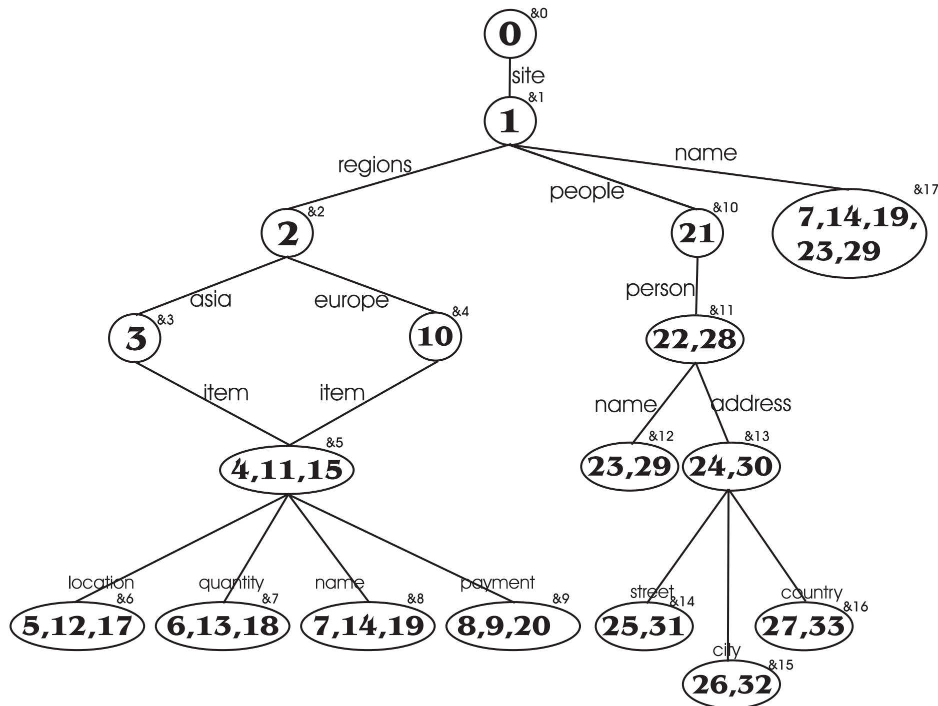


Figure 4.7: The adaptive path index (APEX) for the sample data.

4.1.6 Numbering Schemes and Tree Signatures

Numbering schemes (e.g. [38],[57]) map each element of the XML data to one or more numbers that are mostly determined by post/preorder XML-tree traversing algorithms. The numbers are used for a faster retrieval of relationships between elements. The work [62] proposes a numbering schema that is optimized for updates leading to less number recalculation when the XML data is modified. In general, numbering schemes are not selective and do not cover key-queries. On the other hand, they are comparable to a structural summary because the structure is implicitly expressed in the numbers.

Approaches that use tree signatures to process XML queries (e.g. [135]) can be compared to the numbering schemes with the difference that the function creation signatures from XML data is not bijective. Therefore, signatures may lead to wrong hits that have to be filtered in a further step.

4.1.7 Further Structural Summaries

Structural summaries are the main data structure in further works, including the Forward-and-Backward-Index [2], the D(K)-Index [17], Covering Indexes [58], and HOPI [103], a so-called *connection index* . HOPI is tailored for queries with long paths especially with wildcards and the XPath axes *descendant* and *ancestor* which cannot be supported efficiently by several structural summaries.
All structural summaries have in common that they require navigation with several steps in their internal data structure when evaluating a query. A major disadvantage is that structural summaries ignore the values of elements and attributes, so that path expressions with value conditions cannot be executed efficiently.

4.2 Value Indexes

4.2.1 Inverted Lists

Some XML index approaches only index the value of elements without paying attention to the full leading path. Those indexes can only support very restricted XPath queries like `//*[.='Dan']`. Because any structural conditions are ignored the relevance of these approaches is questionable. Nevertheless we list some works here.

Accelerating key value queries for XML data by information retrieval techniques is one popular proposal that reuses known implementations like Inverted Lists or Tries. An inverted list enumerates all values appearing in the XML data in a search structure and references the position of their appearance. The inverted list for the XMark sample data is shown in figure 4.8.
For the reason of readability only the first 5 references into the original XML data are drawn. In reality, each item of the inverted list has at least one reference.
An inverted list offers logarithmic time to retrieve an element if the element's value is known.

4.2.2 Lore Value Index

The *Value Index* of the *Lore Database Management System* [76] is a selective inverted list for element values. Selective means that the label of an element is reflected and that a set of labels to be indexed can be defined by a database administrator. For instance, the DBA assigns all `name` elements to be indexed by

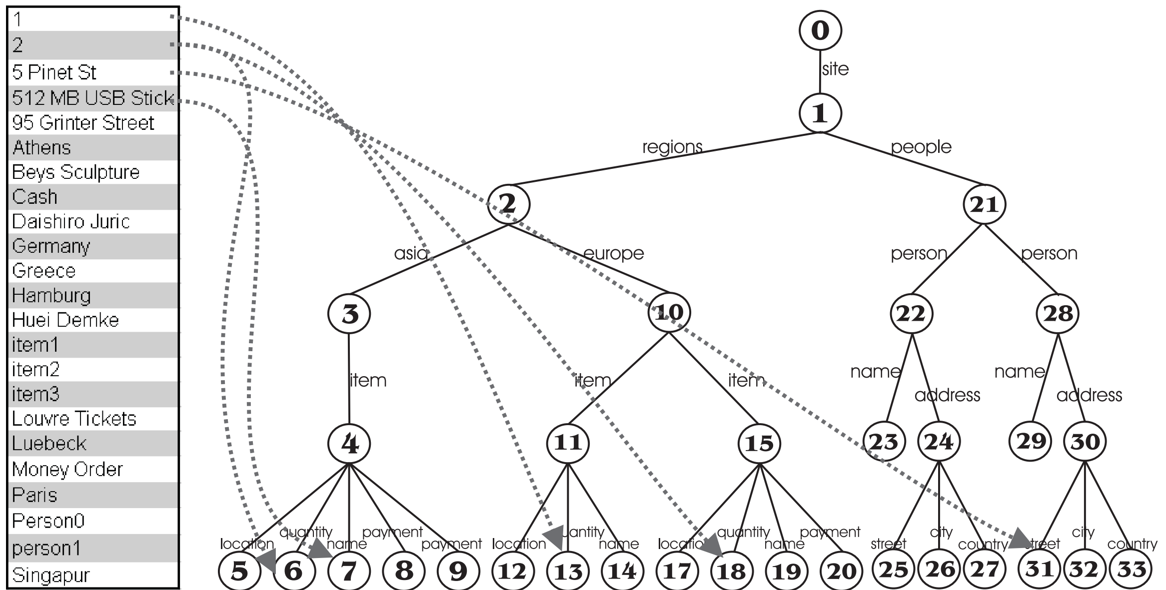


Figure 4.8: The inverted list for the XMark sample data.

their values. But because the path to the elements is ignored the Lore Value Index cannot distinguish between elements with the same label but different paths, e.g. the name of an item and the name of a person.

4.2.3 SEQL

A more sophisticated approach is presented by *SEQL (Search Engine Query Language)* [89], managing an additional inverted list for the element labels. Nevertheless, because SEQL regards XML data as a text document and not as a tree of nodes, it returns text positions instead of nodes, so that further navigation is uncomfortable.

4.3 Hybrid Approaches

We think that XML indexes that support both structural queries and value queries are the most relevant indexes because they cover most XML query capabilities. Even a simple XPath query like `//people/person[name='Jan']` cannot be supported efficiently by the structural indexes and value indexes. Structural summaries may return all `//people/person` nodes more or less efficiently depending on how the descendant axis is dealt with. A value index cannot distinguish between a person's name `Jan` and an item's name `Jan`.

4.3.1 Structural Summary plus Inverted List

An obvious approach to support both structural and value queries is to combine two separate data structures like a structural summary and an inverted list. This

is done in the work of [9, 40, 59]. A query is separated into the structural part processed by the structural summary and the value part that is executed by use of an inverted list . Both data structures return relevant nodes that have to be intersected at runtime to retrieve the final result set. In general, this intersection operation may produce enormous costs for removing wrong hits.

4.3.2 Content-Aware DataGuide

The problem of the expensive intersection operation is faced in the *Content-Aware DataGuide* (CADG) [115, 116] by Weigel et al. where a content/structure join is precomputed and leads up to a 400 times faster execution compared to the conventional DataGuide .
Weigel et al. introduce two approaches: The naive *content - centric* approach where a separate DataGuide is established for each value (i.e. content). A content - centric DataGuide is presented in figure 4.9.

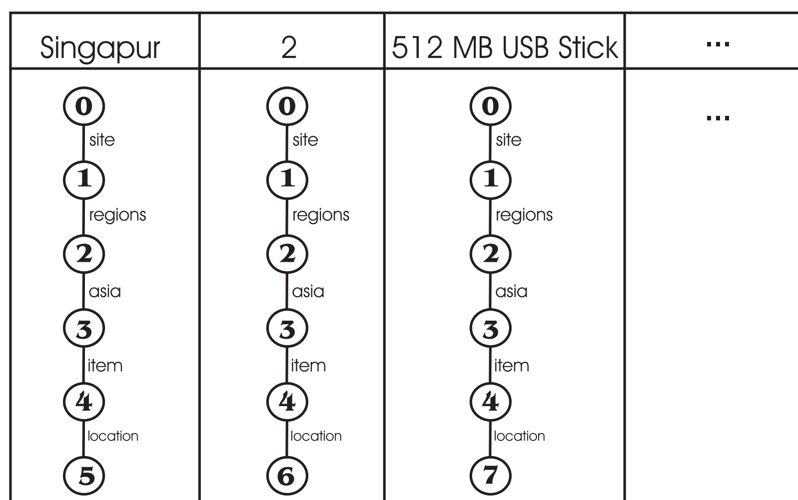


Figure 4.9: A content-centric Content-Aware DataGuide.

For each element value of the sample data the content-centric Content-Aware DataGuide contains one conventional DataGuide that refers to all nodes that contain this value. Figure 4.9 contains the first DataGuides, in total there are more than a dozen - one for each value. When processing a path expression p one has to extract its value, take the corresponding DataGuide (if available) and check if the path of p is evaluated to the leaf node of the DataGuide.
Beside the fact that this approach wastes a lot of space it is only suitable for single key queries. Path expressions with more than one value comparison or a range of values are supported less efficiently and require join-operations.

The authors that are aware of these severe disadvantages propose a second approach that is *structure - centric*. It takes a conventional DataGuide and enriched extents with content information. This information is taken to prune irrelevant

paths when processing a path expression. Two functions may be evaluated for an extent e: The first Boolean function *governs(e, v)* responds if the extent or one of its descendant contains an element with the requested value v. The second Boolean function *governs(e, v)* returns only true, if e itself contains v. In the sample data only the node with the id 5 contains an element with the value `Singapur`, but all nodes with ids 0 to 5 govern the value.
The authors propose two methods to capture the content in the extents: The first approach assigns a unique id to every value in the XML data and puts it into the extents that contain/govern this value. Because an own id for each values increases the size of the DataGuide dramatically the second proposal uses binary signatures of a restricted length. A non-bijective function assigns values to signatures. If an extent governs or contains more than one value the corresponding signatures are unified bitwise to a single signature that represents all values. This process is not lossless, leads to false positives and therefore requires postprocessing when evaluating a path expression.

A major issue of the CADG is its limited capability to deal with updates. When adding/deleting a node or when changing the value of a node the corresponding signatures/ids must be identified and recalculated respectively deleted. In general, an update implies that all extents of the Content-Aware DataGuide must be touched. This is a linear complexity in the size of the database.

4.3.3 ViST

With the *Virtual Suffix Tree (ViST)* [114] Wang et al. introduce an approach that encodes and represents XML data and path expressions as structure-encoded sequences. XML data is represented by the preorder sequence of its tree structure produced by a depth-first traversal of the XML data. The value of elements and attributes and the labels of all elements are combined to one large sequence. Therefore, ViST is comparable to a numbering schema. Since isomorphic trees may produce different preorder sequences an order among sibling nodes is enforced using the lexicographic order of the labels. Multiple siblings with the same label (e.g. the `payment` element in the XMark sample data) are ordered randomly.

In order to motivate the ViST approach we use the following DOM-represented XML fragment in figure 4.10.
ViST transforms an XML data into a sequence of (symbol, prefix) pairs - the so-called structure-encoded sequence D. The XML fragment of figure 4.10 leads to the following structure-encoded sequence D:

```
1  D=
2  (<site>,) ,
3  (<regions>,<site>) ,
4  (<asia>,<site><regions>) ,
5  (<item>,<site><regions><asia>) ,
6  (<location>,<site><regions><asia><item>) ,
```

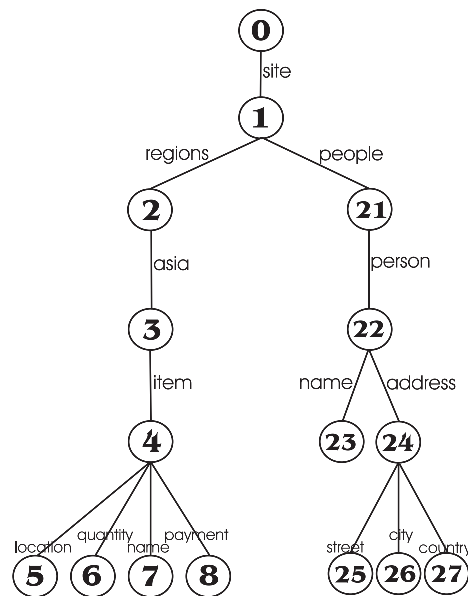


Figure 4.10: A fragment of the XML sample data used for ViST.

```
7  (Singapur,<site><regions><asia><item><location>),
8  (<quantity>,<site><regions><asia><item>),
9  (2,<site><regions><asia><item><quantity>),
10 (<name>,<site><regions><asia><item>),
11 (512 MB USB Stick,<site><regions><asia><item><name>),
12 (<payment>,<site><regions><asia><item>),
13 (Money Order,<site><regions><asia><item><payment>),
14 (<people>,<site>),
15 (<person>,<site><people>)
16 (<name>,<site><people><person>),
17 (Huei Demke,<site><people><person><name>),
18 (<address>,<site><people><person>),
19 (<street>,<site><people><person><address>),
20 (95 Grinter St,<site><people><person><address><street>),
21 (<city>,<site><people><person><address>),
22 (Luebeck,<site><people><person><address><city>),
23 (<country>,<site><people><person><address>)
24 (Germany,<site><people><person><address><country>)
```

Figure 4.11: The structure-encoded sequence of the XML fragment.

A path expression is transformed analogously into a sequence. For instance, the query `/site/people/person[name='Huei Demke']` is tranformed into the following query sequence.

```
1 (<site>,),
2 (<people>,<site>),
3 (<person>,<site><people>),
4 (<name>,<site><people><person>),
5 (Huei Demke,<site><people><person><name>)
```

The purpose of introducing structure-encoded sequences is to model XML queries through sequence matching to the structure-encoded sequence D: Querying XML

becomes equivalent to finding subsequence matches. The wildcard operator * matches any single symbol and the descendant axis // with any portion of the path in D. Through sequence matching, the query is matched as a whole against the XML data without splitting it up into sub-queries of paths that must be joined.

Without any further data structures the subsequence matching requires the whole structure-encoded sequence D to be processed when evaluating a query. This would raise time costs linear to the size of the database. Therefore, ViST provides a unified index on structure and content that captures the ancestor-descendant relationship of XML nodes. The index's data structure is a B^+tree , so that existing implementations of disk-resident search structures may be reused. Range queries on the index structure are performed for the fast retrieval of child nodes. Afterwards the returned nodes are taken for matching the remaining part of the query. In general, the evaluation of a path expression with ViST requires a multitude of range queries performed by the index.

ViST is not selective, meaning that the whole XML data is indexed and consulted for the sequence matching of queries - this may lead to performance degradation if the XML data or parts of it are often modified. For structural modifications the preorder of subsequences in the sequences may rise additional maintenance costs: ViST faces similar problems as the numbering schemes: If the structure of the document changes, for instance by adding a new item, all elements after this new node need to be reorganized as their preorder position may have changed. Queries containing a wildcard(*) or the descendant axis (//) are supported but with less performance, as a lot more sequence matching has to be done. In addition, the sequence matching algorithm has a higher complexity than the logarithmic key retrieval operation of KeyX.

4.3.4 Index Fabric

The *Index Fabric* [22] is an indexing approach that generates keys by the concatenation of the values of elements and attributes and the elements labels of the leading paths. For instance, `'siteregionseuropeitemnameLouvre Ticket'` is the key that references the name element of the third item in the XMark sample data. So-called *designators* are applied to shorten the length of the keys. A designator is a symbol representing a step of the path. The designators' symbols may not be a character of the XML data in order to prevent ambiguities. The above example can be shortened by the use of designators to '$\alpha\beta\gamma\delta\epsilon$`Louvre Ticket`' with α denoting `'site'` and so on.

The keys for all leaf nodes with values of the XML data are stored in a balanced tree structure offering logarithmic key retrieval time. This index structure is called the *Raw Path index*. The data structure is a Patricia Trie[85]. In contrast to search trees, like a red-black-tree or a B-tree, the Patricia Trie does not store the full key

but skips prefixes that are identical. By performance measurements[75] we have shown that for representative XML data the Patricia Trie does not perform better than other search trees like the Java TreeMap, for example. This may be due to the reason that the designators remove already the redundant part of the keys.

This Raw Path is tailored to answer single-key queries starting from the document root and contain one value comparison at the end of the queries' path expression. Structural queries (with no value comparison) or queries containing a wildcard (*) or the descendant axis (//) cannot be supported. The Raw Path index is not selective, this means that it indexes all leaf nodes with a value.
The Raw Path requires additional navigation if the indexed element is not the requested one. For example, if we are interested in the author of a book with a given title, the index returns the `title` element. From this element we have to navigate in the original XML data to the `author` element(s).

The second approach of the Index Fabrics is the so-called `Refined Path index` which is selective to specific queries. The path expressions to be supported by the Refined Path index have to be manually preselected by the database administrator. Each Refined Path is represented by a designator and may include multiple keys .
For instance, it is possible to index a person by its name and city values. This refined path would cover queries of the type

`/site/people/person[name =`x_1 `and address/city=`x_2`].`

A special designator symbol is assigned to this refined path, let's say ξ. The keys for the XMark data would be 'ξHuei DemkeLuebeck' and 'ξDaishiroJuricAthens' referencing the nodes with the id 22, respectively 28. These keys are stored in the same Patricia Trie used for the Raw Path index keys. The Index Fabric with all Raw Paths and the one described Refined Path is illustrated in figure 4.12.
The designator dictionary (a) keeps the abbreviations of all label names. All keys of the Raw Path are stored in a Patricia Trie. We list all keys and their references to nodes of the XML data in b). For the reason of readability only not all references are drawn. The Refined Path leads to the last two entries of b) and references non-leaf nodes of the XML data.

Range queries are supported by the Index Fabric - but only for single key indexes. Before a range query can be executed all strings in the Patricia Trie representing numeric values have to be normalized to the same length. For example '2' is normalized to '002'. This has to be done because the trie is ordered lexicographically ('2' is greater than '10' whereas '002' is smaller than '010'. The Refined Path supports multi-key indexes and avoids the navigation from an indexed key value to the return value. As described in [22] range queries cannot be executed upon multi-key indexes as all keys are concatenated to one atomic artificial key. In the above example it is not possible to separate a person's name from its city. Queries

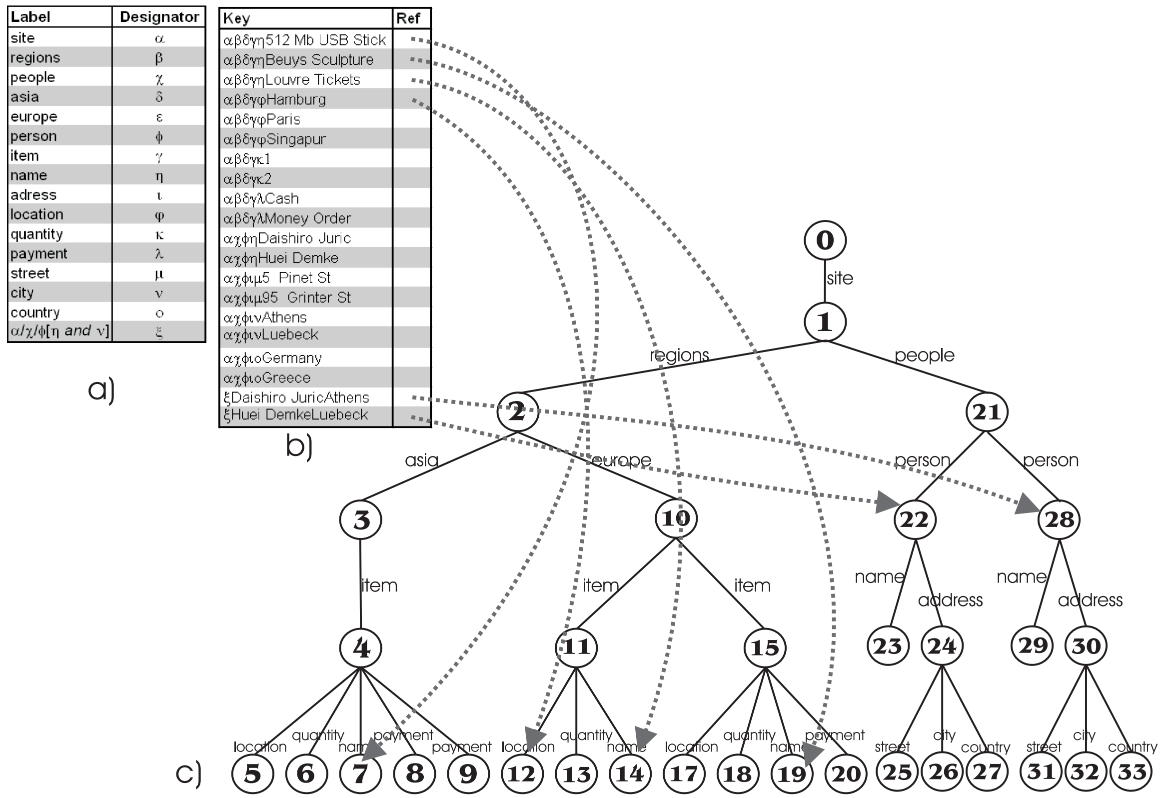


Figure 4.12: The Index Fabrics for the XMark sample data

with a wildcard may be supported by the Index Fabric Refined Path if the wildcard is part of the path expressions that defines it.

We compare the performance of the Index Fabric's Raw Path and Refined Path by measurements for several queries of different types in [42].

4.3.5 System RX Index

The hybrid XML database management system System RX [11] also provides an index for queries containing both structural and value conditions. As in KeyX, the Index Fabric indexes nodes that are selected by an (X)path expression. The index is implemented with two B+Trees. The *path index* maps each distinct reverse path (revPath) to a generated path identifier (pathId). A reverse path (revPath) is a list of node labels from leaf to root - compressed into a vector of label identifiers. To make an analogy with relational systems, the path index is like a dynamic version of the COLUMNS catalog that slowly changes as documents are inserted. The paths are stored from leaf to root for efficient processing of descendant queries such as //name which only bind the tail of the path. This approach is similar to the inverse order of hashtables in APEX.

The value index consists of the keys storing the pathId, the value and a nodeId.

The nodeId identifies a node within the document and can provide quick access to a node. The order of the keys in the value index is again a trade-off. Placing the pathId first allows for quick retrieval of specific path queries. For example, if we create an index on //name, which might match many paths, then a query on /book/author/name still has consecutive index entries. The path index plus placing the pathId first in the value index gives us some structural index support as well. But the tradeoff is that a query like //name='Maggie' will need to examine every location in the index per matching path.
The paper [11] introduces System RX in general, but is not providing enough insight into the indexing aspect. Therefore, the description of System RX's indexing approach in this thesis must be somewhat superficial.

4.4 XML Indexes and Updates

An index is a data structure that reflects parts or the total indexed data. Therefore, an index a is highly dependent on the original indexed data b. Whenever b is modified the modification has to be performed on a, too in order to guarantee consistency. If there is a difference between a and b the results of queries performed on the index and on the original data are not the same. If an index is created once an and the indexed data is never modified in lifetime of the database this problem is not raised - but this is an unrealistic assumption.
In the context of XML databases we face two problems: first, for indexes that are selective and cover only parts of the indexed data (e.g. Index Fabric, T-Index) it has to be checked whether the index is affected by a modification or not. Chapter 7 is dedicated to this topic and presents an algorithm that decides this problem for given path expressions. Non-selective indexes like the structural summaries are affected by every modification. Therefore, the problem of checking the affection is not relevant for these approaches. Non-selective approaches raise huge maintenance costs if they are used in a database application that has many updates. For instance, in an auction scenario it is likely that the price of an item changes very often. A selective indexing approach can be tuned not to index the price elements whereas the non-selective structural summaries raises many updates.

Whenever an index is affected its underlying data structure must be updated. This is a technical problem that is highly dependent on the data structure of the index. For structural summaries, like the Strong DataGuide and the 1/2-Index it is required to find the affected extents and their content. For updates with a pure navigational path expression (e.g. delete all books) it is relatively easy to find the corresponding extent and to process it as a whole (e.g. delete it). For updates that contain a value condition (e.g. delete all books written before 1999) the full content of the affected extents have to be checked. Because the structural summaries usually store no value information this may raise exhaustive costs, if all references in the extents have to be dereferenced to XML elements, retrieved from

the database and checked if they fulfill the condition. A path expression with a wildcard or a descendant axis makes the problem even worse because multiple extents may be affected. All in one, indexing approaches that are not selective and that have no value information seem to be unsuitable for applications that have frequent value-centric updates. Unfortunately, the mentioned publications do not address this problem in particular.

4.5 Conclusion

The multitude of approaches that we introduced in this section raises the question why we need another XML index structure like KeyX. First, we think that only approaches that support both structure and content are relevant for real-world applications (e.g. find a customer by its customer number). Second and analogously to RDBMS, a broad support of query types including queries with multiple keys and ranges is not less important (e.g. find all customers with a ZIP between 23000 and 23999 who ordered items for more than €100). All introduced approaches lack more or less in one or more characteristics.
We collect the characteristics of the surveyed index approaches in table 1.

	Sel	PPQ	SKQ	RQ	DQ	WQ	MKQ	K≠V	Nav
Strong Data Guide (Lore)	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
1-Index	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
2-Index	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
T-Index	⊕	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
APEX	⊕	⊕	⊖	⊖	⊕	⊕	⊖	⊖	$O(n)$
Numbering Schemes	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
Tree Signature	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊖	$O(n)$
Value Index (Lore)	[2]	⊖	[1]	[1]	[1]	⊕	⊖	⊖	–
SEQL	⊖	⊖	⊕	⊕	⊕	⊖	⊖	⊖	–
CADG	⊖	⊕	⊕	⊖	⊕	⊕	⊖	⊖	–
ViST	⊖	⊕	⊕	⊕	⊕	⊕	?	⊖	–
Raw Paths (Index Fabric)	⊖	⊕	⊕	⊖	⊖	⊖	⊖	⊖	–
Refined Paths (Index Fabric)	⊕	⊕	⊕	[3]	⊕	⊖	⊕	⊕	–
System RX	⊕	?	⊖	?	⊕	?	?	?	–
KeyX	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	–

⊕: feature supported,
⊖: feature not or not efficiently supported,
?: not clarified in publication
Sel: index is selective and optimizes frequent queries
PPQ: pure path query (e.g. `/dblp/inproceedings/author`)
SKQ: single-key query (e.g. `/dblp/book/author[.='Suciu']`)
MKQ: multi-key query (e.g. `/dblp/book[author='Suciu' AND year=2004]`)
RQ: range query (e.g. `/dblp/article/year[.<2004]`)
DQ: descendant query (e.g. `//title`)
WQ: wildcard query (e.g. `/dblp/*/title`)
K≠V: return value does not have to be the key value (reduces navigation)
Nav: navigation complexity in index structure (n is length of query, m the number of indexed nodes)

Table 4.1: Comparison of different XML index approaches

[1]Provided if the path has exactly one step.
[2]Provided, but elements with same name but different paths are not distinguishable.
[3]Range queries are supported as single-key queries, but not as multi-key query.

Chapter 5

The Key-Oriented XML Index KeyX

In this chapter we introduce a new approach for indexing XML data formally and by examples. Our approach - called KeyX - is motivated by the selective index structures used within the relational world. Relational indexes are defined upon a specific table and one (or multiple) columns. Only queries that operate on these columns can be accelerated with this index. Therefore, a relational index is *selective* to specific queries.
Like relational indexes, KeyX is based on *keys* - the values of elements and attributes which are accessed by a specific path expression. The path expression can be part of an XQuery or XUpdate operation.
For a set of frequent queries[1] the relevant keys are extracted from the original XML data and stored in a search structure optimized for efficient key retrieval. Those search structures include hashtables, tries, binary search trees, B^+Trees for disk resident indexes, or any other data structure that is capable of storing and retrieving keys.

An index is defined by the 'shape' of the path expression to be optimized. After materializing the index, further queries with a matching shape are processed by the index - with logarithmic instead of linear complexity. For real databases with a size of several megabytes a set of suitable indexes implies an acceleration factor of many magnitudes.
KeyX can also be used to accelerate specific navigational queries. In contrast to structural summaries like Strong DataGuides and APEX our indexing approach is defined for a set of frequent navigational queries[1]. A selective structure index consumes less space and can be tuned for update issues.

In the following we introduce KeyX formally and by examples. We prove the quality of this approach by performance measurements.

[1]Frequent queries can be defined by a database administrator or by tools that analyze the workload of the database.

5.1 KeyX Formally

An index in KeyX consists of a declaration describing the shape of the covered path expressions and the index's content which depends on the XML data to be indexed. An index declaration is used to create an index and is derived from a general path expression $p \in XP^{\{[],*,//\}}$ to be optimized.

5.1.1 Index Declaration

Definition 16 Formally, an *Index Declaration* d for a general path expression $p \in XP^{\{[],*,//\}}$ is defined as a triple $i = (K, Q, v)$ where K is a list of n linear path expressions referring to the n *keys* of p:

$$K = (k_1, k_2, ..., k_n), \textit{with } k_1, ..., k_n \in P_l.$$

Analogously, Q is a list of m linear path expressions referring to the m *qualifiers* of p:

$$Q = (q_1, q_2, ..., q_m), \textit{with } q_1, ..., q_m \in P_l.$$

Finally, $v \in P_l$ is a linear path expression referring to the return value of p.

We can regard an index declaration as a mathematical function $I : (k, q) \rightarrow v$ with (k, q) the domain and v the range. The set of all index declarations is denoted by D. □

Informally, one can say that an index declaration consists of the information about the paths to its keys, qualifiers, and return value. The qualifiers are a structural property that must be fulfilled by all indexed nodes.
In contrast to the content of an index, its declaration d is independent of the concrete XML data.

Example 9 The general path expression $p_1 \in XP^{\{[],*,//\}}$ with

$$p_1 = /publications/book[isbn][author = x_1 \textit{ and } year = x_2]$$

returns all `book` elements that have an author child and a year child with given values and a child `isbn`. Because `isbn` is a qualifier its value is not relevant.

Based on p_1 we determine

$$\begin{aligned} K &= \{/publication/book/author, /publication/book/year\} \\ Q &= \{/publication/book/isbn\} \\ v &= /publication/book \end{aligned}$$

by the use of path extraction functions defined in the following. □

5.1.2 Path Extraction Functions

The linear path expressions to the keys, the predicates, and the return value are extracted from a path expression $p \in XP^{\{[],*,//\}}$ that defines the index declaration d. We use three functions to extract the key paths, the qualifier paths and the value path.

Definition 17 (Value Node Path)
The function *value* : $XP_{rel}^{\{[],*,//\}} \rightarrow P_l$ extracts a linear path expression to the value nodes of a general path expression and is defined as follows:

$$\begin{aligned} \textit{value}(p_1\ \alpha\ p_2) &= \textit{value}(p_1)\ \alpha\ \textit{value}(p_2) && (5.1)\\ \textit{value}(p[q]) &= \textit{value}(p) && (5.2)\\ \textit{value}(s) &= s && (5.3) \end{aligned}$$

with $p, p_1, p_2 \in XP_{rel}^{\{[],*,//\}}$, $s \in P_l$ and $\alpha \in \{/, //, ..\}$ denoting the axis. □

Line 5.1 separates the location steps of the path expression. Line 5.2 removes all predicates. The remaining path expressions are linearized so that possible parent axes are removed.

The extension of $XP_{rel}^{\{[],*,//\}}$ to absolute path expressions $XP_{abs}^{\{[],*,//\}}$ is given by:

$$\textit{value}(/p) = /\textit{value}(p)$$

Example 10 The value node path of

$$p_1 = /publications/book[isbn][author = x_1 \text{ and } year = x_2]$$

is extracted as follows:

$$\begin{aligned} \textit{value}(p_1) &= \textit{value}(/publication/book[isbn][author = x_1 \text{ and } year = x_2])\\ &= /\textit{value}(publication/book[isbn][author = x_1 \text{ and } year = x_2])\\ &= /\textit{value}(publication)/\textit{value}(book[isbn][author = x_1 \text{ and } year = x_2])\\ &= /publication/\textit{value}(book)\\ &= /publication/book \end{aligned}$$

□

In a similar manner a function can extract the key node paths of a given path expression. Because we can have multiple key values given in the path expression, the operation returns a set of linear path expressions.

Definition 18 (Key Node Path)
The function *key* : $XP_{rel}^{\{[],*,//\}} \rightarrow \mathcal{P}(P_l)$[1] extracts a set of absolute linear path expressions to the key nodes of a path expression and is defined as follows:

$$\begin{aligned}
key(p_1\ \alpha\ p_2) &= key(p_1) \cup \{value(p_1)\ \alpha\ x\ |x\ \in key(p_2)\} && (5.4)\\
key(p[q]) &= key(p) \cup \{value(p)/x \mid x \in key(q)\} && (5.5)\\
key(q_1\ and\ q_2) &= key(q_1) \cup key(q_2) && (5.6)\\
key(q_1\ or\ q_2) &= key(q_1) \cup key(q_2) && (5.7)\\
key(not\ q) &= key(q) && (5.8)\\
key(s\ r\ l) &= value(s) && (5.9)\\
key(s) &= \emptyset && (5.10)
\end{aligned}$$

with $p, p_1, p_2 \in XP_{rel}^{\{[],*,//\}}$, $s \in P_l$ and $\alpha \in \{/, //, ..\}$ denoting the axis; r is the operator used to compare the value of an element with $r \in \{=, \neq, <, >, \leq, \geq\}$; l is the literal used for the comparison. See also section 6. □

The extension of $XP_{rel}^{\{[],*,//\}}$ to absolute path expressions $XP_{abs}^{\{[],*,//\}}$ is given by:

$$key(/p) = \{/x|x \in key(p)\}$$

Example 11 The key node path of $/publications/book[isbn][author = x_1$ and $year = x_2]$ is extracted as follows:

$$\begin{aligned}
key(p) &= key(/publication/book[isbn][author = x_1 \text{ and } year = x_2])\\
&= /key(publication/book[isbn][author = x_1 \text{ and } year = x_2])\\
&= /key(publication) \cup\\
&\quad \{\ value(publication\ /x \mid x \in key(book[isbn][author = x_1 \text{ and } year = x_2])\ \}\\
&= /\Big(\emptyset \cup \{publication/x|\ x \in key(book[isbn][author = x_1 \text{ and } year = x_2])\ \}\Big)\\
&= /\Big(publication/x \mid x \in \Big\{key(book[isbn])\ \cup\\
&\quad \{value(book[isbn]/x \mid x \in key(author = x_1 \text{ and } year = x_2))\}\Big\}\Big)\\
&= /\Big(publication/x \mid x \in \Big\{key(book) \cup \{value(book)/x|x \in key(isbn)\}\ \cup\\
&\quad \{value(book[isbn]/x \mid x \in key(author = x_1 \text{ and } year = x_2))\}\Big\}\Big)
\end{aligned}$$

[1] $\mathcal{P}$ denotes the power set

$$
\begin{aligned}
&= /\Big(publication/x \mid x \in \Big\{\emptyset \cup \{book/x|x \in \emptyset\} \;\cup \\
&\quad \{\textit{value}(book[isbn]/x \mid x \in \{\textit{key}(author = x_1) \;\cup\; \textit{key}(year = x_2)\}\}\Big\}\Big) \\
&= /\Big(publication/x \mid x \in \Big\{\emptyset \cup \\
&\quad \big\{\textit{value}(book[isbn]/x \mid x \in \{\textit{value}(author) \;\cup\; \textit{value}(year)\}\}\Big\}\Big) \\
&= /\Big(publication/x \mid x \in \{\textit{value}(book[isbn]/x \mid x \in \{author, year\}\Big) \\
&= /\Big(publication/x \mid x \in \{\textit{value}(book[isbn]/author), \textit{value}(book[isbn]/year)\}\Big) \\
&= /\Big(publication/x \mid x \in \{book/author,\ book/year\}\Big) \\
&= /publication/book/author,\ /publication/book/year \qquad \square
\end{aligned}
$$

The function to extract the qualifier set Q is almost the same. The only difference is in the rules 5.9 and 5.10 discriminating the linear path expressions of qualifiers from the value comparisons of the keys.

Definition 19 (Qualifier Node Path)
The function *qualifier* : $XP_{rel}^{\{[],*,//\}} \rightarrow \mathcal{P}(P_l)$ extracts a set of absolute linear path expressions to the qualifier nodes of a path expression and is defined as follows:

$$
\begin{aligned}
\textit{qualifier}(p_1\ \alpha\ p_2) &= \textit{qualifier}(p_1) \cup \{\textit{value}(p_1)\ \alpha\ x\ |x\ \in \textit{qualifier}(p_2)\} && (5.11)\\
\textit{qualifier}(p[q]) &= \textit{qualifier}(p) \cup \{\textit{value}(p)/x \mid x \in \textit{qualifier}(q)\} && (5.12)\\
\textit{qualifier}(q_1\ \textit{and}\ q_2) &= \textit{qualifier}(q_1) \cup \textit{qualifier}(q_2) && (5.13)\\
\textit{qualifier}(q_1\ \textit{or}\ q_2) &= \textit{qualifier}(q_1) \cup \textit{qualifier}(q_2) && (5.14)\\
\textit{qualifier}(\textit{not}\ q) &= \textit{qualifier}(q) && (5.15)\\
\textit{qualifier}(s\ r\ l) &= \emptyset && (5.16)\\
\textit{qualifier}(s) &= \textit{linearize}(\textit{value}(s)). && (5.17)
\end{aligned}
$$

with $p, p_1, p_2 \in XP_{rel}^{\{[],*,//\}}$, $s \in P_l$ and $\alpha \in \{/, //, ..\}$ denoting the axis; r is the operator used to compare the value of an element with $r \in \{=, \neq, <, >, \leq, \geq\}$; l is the literal used for the comparison. □

The qualifiers are determined in a similar manner as the keys. The qualifier node paths of $/publications/book[isbn][author = x_1$ *and* $year = x_2]$ are

$$\{/publication/book/isbn\}$$

5.1.3 KeyX Search Structure

KeyX stores key-value tuples in a data structure s that is optimized for fast key retrieval. The values are references to nodes (elements) in the XML data. In general, any data structure that offers efficient key retrieval (e.g. B-Trees, B^+Trees, binary search trees, or tries and hashtables) can be applied.

If a KeyX index contains multiple keys (e.g. it indexes books by an author and year) the search structure is *nested* like in relational indexes. The reader is referred to [13] for an overview of query optimization in relational systems. Nesting means that keys are arranged in different layers. Keys of one layer (e.g. the authors) point to relevant keys in the subjacent layer (e.g. the years). This approach is adopted from search trees within the relational world and does not combine multiple keys and paths to one artificial key like the Index Fabric does. That is the reason why KeyX supports range queries easily.

For memory-resident indexes we use a *Red-Black tree* guaranteeing an average $log(n)$ time for key retrieval. Disk-resident indexes may rely on search structures that have proven their efficiency within the relational world. Examples include B-Trees or B*-Trees loading multiple tuples at once in order to limit the number of hard disk accesses.
Because KeyX abstracts from the underlying search structure it is possible to use the one that performs optimal for a given application. This includes the very special search trees like the Patricia Trie, for instance. As one can see, KeyX is not a new data structure but an index approach that may reuse existing implementations. By performance measurements [42, 75] we have shown that standard data structures like the binary search tree (implemented in the Java TreeMap) is not outperformed by more special data structures like the Patricia Trie when applied in the indexing context.

For the performance measurements in this paper we used memory-resident indexes in order to minimize the influence of the hard disk and the operating system.

5.2 Index Creation Algorithm

The creation procedure of a *single-key index* is based on a given path expression $p \in XP_{abs}^{\{[],*,//\}}$ and a given XML data t. p can be an XPath or XQuery based query operation as well as the selecting part of an XUpdate operation.
The algorithm that creates a KeyX index works as follows:

- First, the one given key path $p_k \in P_l$ is evaluated on the XML data returning the key elements. The text values of these elements are interpreted as keys.

p has exactly one key path because it is a single-key query (see section 2.3.2).

- Second, for each key node we navigate to the corresponding return value leading to a set of *(key, return value)* tuples. For each of these tuples we test whether the conditions expressed by the qualifiers are fulfilled. If we have no qualifiers this step can be omitted.
- The last step stores the remaining tuples in a new search tree dedicated to executing queries of the shape of p.

The creation procedure of a `multi-key index` is a little more complex because the keys of the different key paths are arranged in levels. Therefore, we use a recursive algorithm that nests search trees for the key paths in different levels. The *(key, return value)* tuples are stored in the last level. We present a Java derived pseudocode of the creation algorithm for single-key and multi-key indexes in Listing 5.1:

```
1  void createIndex(XMLNode context,
2                   SetOfKeyPaths Kp,
3                   SetOfQualifierPaths Qp,
4                   ReturnValuePath rp,
5                   SearchTree tree){
6    if ( |Kp| == 1){           //Kp has only 1 key path
7      keyP = Kp[1];            //get path to this keys
8      relKeyP = keyP-context; //relative path to the keys from context node
9      relValP = rp-keyP;      //relative path to the return value

11     KEYS = eval(context, relKeyP);  //retrieve the key elements
12     forall (k in KEYS){
13       val = eval(k, relValPath);    //retrieve the return value
14       if (testQualifier (val, Qp)){ //test qualifier
15         int id = val.getId();       //get id of return node
16         tree.add(k.getText(), id);  //add key-value tuple
17       }
18     }
19   }
20   else{                             //build a multikey index
21     keyP = Kp[1];                   //take first KeyPath
22     Kp = Kp \ Kp[1];                //and remove it from Kp
23     relKeyP = keyP-context;         //relative path to keys from context

25     KEYS = eval(context, relKeyP);  //retrieve the key elements
26     forall (k in KEYS)
27       SearchTree tree2 = new SearchTree(); // create empty search tree

29       createIndex(k, Kp, Qp, rp, tree);    // recursive call of method

31       tree.add(k.getText(), tree2);        //nest the second tree
32     }
33   }
34 }
```

Figure 5.1: Pseudocode to create a single-key or multi-key index

The algorithm assumes that the path expressions to the keys, qualifiers, and the return value have been extracted from a given XPath query. These path expressions are extracted by the use of the functions *key*, *qualifier*, and *value* that are

defined in the previous section. The algorithm starts with an empty search tree by `createIndex(/, Kp, Qp, rp, new Searchtree())` with `/` the root node of the XML data.

In line 6 the algorithm tests if there is only one key path. Until now all path expressions are *absolute*, meaning that they refer to the root node. For the further processing we need *relative* path expressions to the key and the return value. This computation takes place in lines 7 to 9. In line 14 the key path is evaluated on the context node by the database management system and returns the key nodes (XML elements) in the XML data. For each of these key nodes we evaluate the relative path to the return value in order to get the corresponding return value (line 13). Afterwards, we have to check if all qualifiers are fulfilled for the return value; this is done in line 14 by the method *testQualifer*. The code of this method can be found in the Appendix 10.2. If there are no qualifiers the method returns always true. The remaining return values and the values of the keys are stored in the search structure in line 16. Of course, we cannot store XML nodes directly; instead we use their id as a reference. An example for a single-key index is presented in figure 5.4.

When creating a multi-key index the algorithm extracts the first key path from the set of all key paths in lines 21 and 22. Afterwards the relative path to the keys from the context node i is determined in line 23. By evaluating this path expression in line 34 we get all key nodes (XML elements) from the XML data. A new search tree is created for each of these key nodes (line 27) and a recursive call with the key node as context and the new tree as search structure takes place in line 29. In order to construct the nested search structure the newly created trees are added to the initial one recursively in line 31. An example for a multi-key index is presented in figure 5.5.

For some applications the scenario of one huge XML document representing the database (like the DBLP or XMark) is unrealistic - instead they propagate an XML database to be a collection of smaller XML fragments. The KeyX approach can easily be extended to support collections by adding a document id to the node id. Instead of storing two ids it is possible to unify both into one common id that can be divided again if required. In this thesis we concentrate on indexing one document/data.

5.2.1 Structural Indexes in KeyX

Pure path queries are structural queries $\in P_l$ with a path expression without any value comparison. Therefore, a pure path query operates on the structure of the XML data ignoring the text values of elements. An example could be

$$p_3 = /site/regions/*/item$$

selecting all `item` nodes. Pure path queries are supported by KeyX by interpreting the whole path as one unique key and storing it in one index dedicated to pure path queries only. The creation algorithm for structural indexes as shown in listing 5.2 is very simple.

```
void createIndex(LinearPathExpression p){
  if (ppIndex == null) ppIndex = new SearchTree();
  String key = p.toString ;
  NodeSet VALUES = eval(p);
  forall (v in VALUES)
      ppIndex.add(key, v.getId());
}
```

Figure 5.2: Pseudocode to create a structural index

All structural indexes are stored in one search tree called i_{pp}. The linear path expression p to be covered by the index is interpreted as key and evaluated afterwards. The ids of the returned nodes are added to the key and stored in i_{pp}. When evaluating a linear path expression with the help of i_{pp} the corresponding key is looked up and the attached ids can be dereferenced instantly. Therefore, exhaustive navigation to all relevant nodes in the XML data is avoided. Additionally, KeyX needs no navigation in the index structure like APEX and Strong DataGuide. We want to emphasize that the KeyX pure path index is not a structural summary because it does not reflect all elements and possible paths of the XML data. Instead it can be applied selectively for specific queries leading to less space consumption and less update expenses. Figure 5.6 shows a pure path index.

5.3 KeyX by Examples

In this section we introduce the KeyX indexes by examples. For the examples in this section and later on at the performance measurements we refer to the well-established computer science bibliography *DBLP* [70] that is available as one huge XML document consisting of approximately 600,000 publications, mainly articles, inproceedings and books. The different publication types are organized under one common root node called `dblp`. In contrast to the XMark data the DBLP data is not artificial but comes from a real database. It is less structured than the XMark data and contains less element types - this makes the DBLP data more clear in our examples.

The sample data consists of two inproceedings and one book:

```
<dblp>
  <inproceedings>
    <author>Mary F. Fernandez</author>
    <author>Dan Suciu</author>
    <title>A Query Language for XML.</title>
    <year>1998</year>
```

```
7       <ee>http://www.w3.org/TandS/QL/QL98/pp/att-position-paper.html</ee>
8     </inproceedings>
9     <inproceedings>
10      <author>Dan Suciu</author>
11      <title>Semistructured Data and XML.</title>
12      <pages>0-</pages>
13      <year>1998</year>
14      <url>db/conf/fodo/fodo98.html#Suciu98</url>
15    </inproceedings>
16    <book>
17      <author>Serge Abiteboul</author>
18      <author>Peter Buneman</author>
19      <author>Dan Suciu</author>
20      <title>Data on the Web</title>
21      <year>1999</year>
22      <isbn>1-55860-622-X</isbn>
23    </book>
24  </dblp>
```

the `Book` and `inproceeding` elements share some elements (e.g. `author` and `title`) and differ in others (e.g `isbn`).

The corresponding DOM-tree [118] representing the XML data as a tree is displayed in figure 5.3. The nodes are numerated and labeled with the element name. The text nodes of the elements are omitted to keep the DOM-tree readable. We use the DOM-tree to illustrate our index structure.

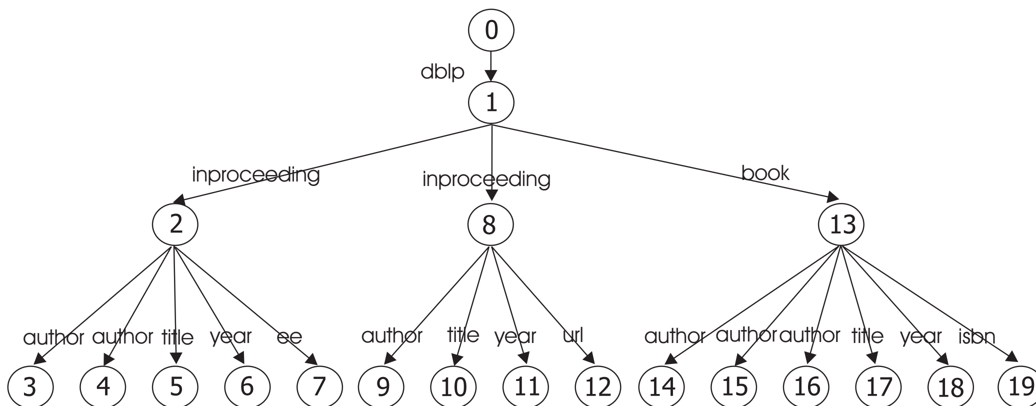


Figure 5.3: XML sample represented as a DOM-Tree

5.3.1 Single-Key Indexes

The path expression $p_2 = /dblp/inproceedings[author =' x']$ compares all author names with the given value *'x'*. For this reason the values of all `author` elements are regarded as keys which are extracted from the XML data by evaluating the key paths. Each key references one or more corresponding nodes (elements or attributes) in the XML data. For instance, the key *'Dan Suciu'* in the index structure for p_2 references all inproceedings which are written by this author. The key(s) k

and the value v of an index may be different. For instance, in p_2 the keys are the values of the element *author* whereas the values are the corresponding inproceedings. Of course, it is possible that the indexed key k is the returned value v of a query. This concept avoids costly navigation from a key to the return value in the XML data. This navigation is only performed once when creating the index or adding new keys. The index structure for p_2 is shown in figure 5.4.

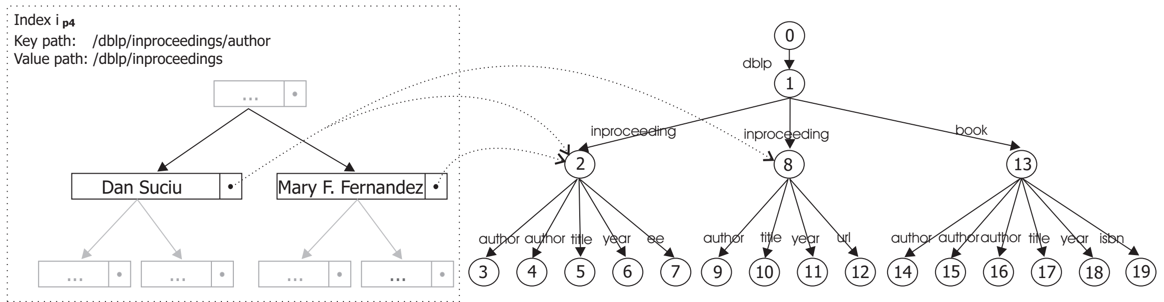


Figure 5.4: Established index i_{p_2}

Please note that this index is specific for queries p_2 indexing all authors of inproceedings. The key *'Dan Suciu'* in the index structure references the inproceedings of this author but not his book.

The path expressions of queries may also contain the *self-or-descendant* (//) and *wildcards* (*) axis shown in the next example. Therefore, it is possible to keep both the authors of books and the authors of articles in one index structure.

5.3.2 Multi-Key Indexes

If a query involves more than one key we need a multi-key index. For example, the query $p_7 = /dblp/*/title[../author.=' x' \text{ and } ../year > y]$ looks for titles of any publication (*) written by a given author and a year. This path expression contains the two keys *author* and *year*.
A multi-key index in KeyX is constructed like in RDBMS: The index's tree structure is ordered by a first key (e.g. *author*), the values of this key are indexes which are built upon the next key (*year*) and so on. Our approach does not combine multiple keys and paths to one artificial key like the Index Fabric's approach does (see section 4.3.4). That is the reason why KeyX supports range queries easily. As our tree implementation differentiates between texts and numbers it does not have to normalize numeric values. Figure 5.5 shows the index for p_7; it is a tree consisting of trees; the second tree's values reference the XML data. Due to the wildcard operator in this query both inproceedings and books are indexed.

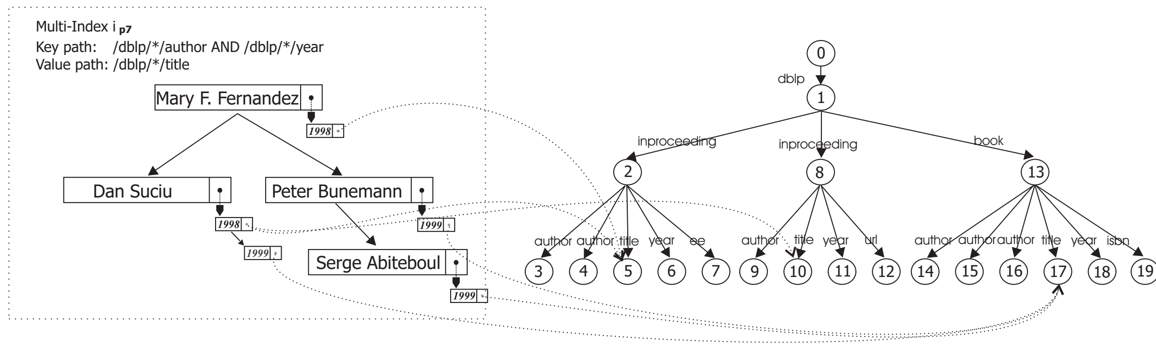


Figure 5.5: Established index i_{p_7}

5.3.3 Selective Structural Indexes

Pure path queries or navigational queries are queries with a path expression but without any predicate. An example is $p_1 = /dblp/inproceedings$ selecting all `inproceedings` nodes. Pure path queries can easily be supported by KeyX if we interpret the whole path as a key and store it in an index dedicated to pure path queries only. In p_1 the path expression is one key and returns all inproceedings instantly. Therefore, exhaustive navigation to all relevant nodes in the XML data is avoided. Additionally, KeyX needs no navigation in the index structure like APEX and Strong DataGuide. This is because the whole path expression consisting of several steps is regarded as one unique key.
We want to emphasize that the KeyX pure path index is not a structural summary because it does not reflect all elements and possible paths of the XML data. Instead it can be applied selectively for specific queries leading to less space consumption and less update operations in the index structure. Figure 5.6 shows the pure path index for four specific queries.

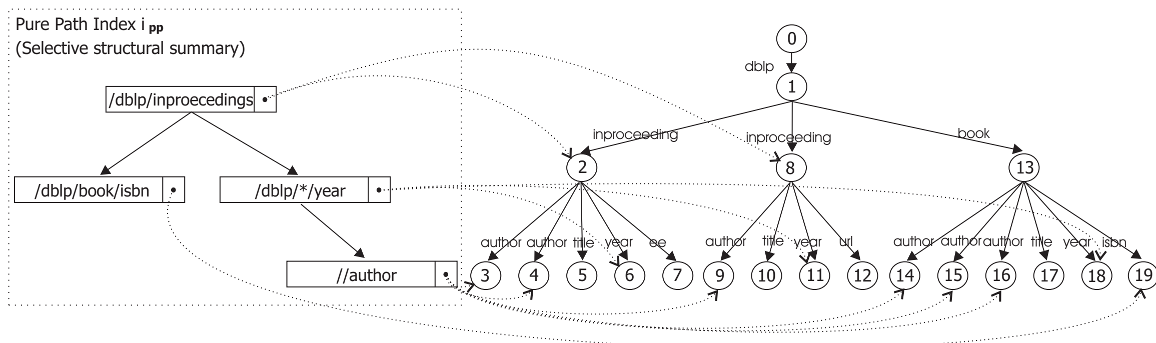


Figure 5.6: Established index i_{pp}

5.4 Query Processing

Executing a query without an index implies a sequential search in the full XML data leading to linear complexity in the size of the XML data. For larger data, e.g. a list of available books in a bookstore, this implies a prohibitive query execution time.

5.4.1 Query Execution with matching KeyX Indexes

The query optimizer extracts the keys of a given query q and looks for an appropriate index. In the following an index j is defined by its declaration consisting of the linear path expressions to the keys, qualifiers and the return value. Formally, j is the tuple (K_j, Q_j, v_j) see section 5.1.1 for details). An XPath based query q consists of analogous path expressions; $q = (K_q, Q_q, v_q)$.
If the path expressions of the query q and an index j are exactly the same then the index matches best and can instantly be used. The key(s) are searched in the tree structure of the index; like in relational indexes this is done in logarithmic time. If the key is found the attached value is either the reference to the corresponding return value in the XML data (single-key index) or a search tree of a lower level (multi-key index) so that further (recursive) key retrieval is performed. The references to the return values in the XML data are returned as the result of the query.

5.4.2 Index Usage with Deviating Return Values

If the path expression of the keys and the qualifiers of the index j and the query q are the same ($K_q = K_j, Q_q = Q_j$) but the path expressions to the return value differ ($v_q \neq v_j$) then the index might still be used in some cases with additional postprocessing:
If the elements requested by v_q are reachable from v_j by a linear relative path expression $p_\triangle = v_q - v_j$ that contains no wildcard or descendant operator we can evaluate $p_\triangle$ on each node that is referenced by the index j through v_j.

Example 12 Lets say we have an index j that indexes `item` elements by their `name` value. The corresponding XPath expression is `//item[name='x']` with $K_j = \{//item/name\}$, $Q_j = \emptyset$ and $v_j = //item$.

The query $q = //item/location[../name =' Sinus\ MP3\ Player']$ has the same key as the index j but the return value differs: $//item \neq //item/location$. The relative linear path expression $p_\triangle$ describes the path from the elements that are referenced by the index to the elements that are requested by the query. In this case $p_\triangle = //item/location - //item = /location$ navigates to the `location` children of each `item` element. □

The postprocessing raises additional costs but if the set of elements referenced by v_j is small the total costs of the query evaluation with the index will still outperform the exhaustive evaluation over the whole document.

If there is a wildcard operator in the path expressions of the query the index may not be suitable anymore because it does not cover all requested elements. For instance, the index defined by `/site/regions/asia/item[name='x']` indexes all items located in asia. The query `/site/regions/*/item[name='Sinus MP3 Player']` does not have a regional restriction. We could calculate the path expression $p_\triangle = ../*$ that navigates from the `item` elements in asia to all children of its parent but this would not lead to success because the index does not cover the values of `name` elements that do not belong to asia. The decision whether we can use an index or not relies on the subset relationship (containment) of the corresponding keys.

5.4.3 Containment Problem

In general, a selective index covers all queries with a result set being a *subset* of the query that defined the index. For instance, an index that is designed to accelerate queries of the form $/dblp/*[author='x']$ is also capable of evaluating queries like $/dblp/book[author='x']$ or $/dblp/article[author='x']$ because the selected keys are a subset of the keys of the index.
When using an index that covers a superset of the elements that are selected by the query an additional postprocessing step has to filter wrong hits: A simple node test checks if the selected nodes are of the requested element type (e.g. an element selected by $*$ is checked if it has the label `book`). Like in the previous case the postprocessing requires linear complexity in the size of the elements that are returned by the index.

If an index is defined with a non-empty set of qualifiers (e.g. only books with an `isbn` child) it cannot be used to process a query that ignores the qualifier because the index does not cover all requested elements. In contrast, a query that poses more qualifiers than the index can be processed by the index with additional postprocessing because the query's result is a subset of the elements that are indexed.

The decision whether the selected nodes of one XPath expression p are a subset of the result set of a second expression p' ($p \subseteq p'$) can be solved using the containment algorithm presented by Miklau and Suciu [82]. This algorithm constructs tree patterns for the path expressions p and p' and creates two (alternating) tree automata A and A' accepting XML data that can be queried by p and p'. Containment holds ($p \subseteq p'$) if $lang(A) \subseteq lang(A')$. A third automaton A'' accepting the complement of $lang(A')$ ($\overline{lang(A')}$) is built on the base of A' by exchanging all accepting states with the non-accepting states. If the product automaton $B = A\ x\ A''$ has no reachable accepting state it holds, that $lang(A) \cap \overline{lang(A')} = \emptyset$. This is equiv-

alent to $lang(A) \subseteq lang(A')$.
The containment algorithm is significantly more complex because A and A' must be transformed into deterministic automata, the tree automata must be ranked and several optimization steps are performed. We implemented the Miklau and Suciu approach in [39].

5.4.4 Rating of Indexes for the Query Execution Plan

If we have more than one index that may be useful for the execution of a query q the question is raised which one is the best for q.

Example 13 Given are the indexes i_1 and i_2 defined by the following path expressions:

$$i_1 := /dblp/book[title =' x']$$
$$i_2 := /dblp/book[year = y]$$

The query $q = /dblp/book[title =' XML'$ *and* $year = 2005]$ asks for books with given values for the `title` and `year` element. The indexes i_1 and i_2 are not matching the query exactly because one key is missing for both indexes. Anyhow, the indexes can be used if additional postprocessing (filtering the second key) takes place. When using i_1 one has to filter the resulting elements by their `year` value. Analogously, when using i_2 one has to filter the resulting elements by their `title` value. □

When the query optimizer receives q the containment algorithm indicates that both i_1 and i_2 can be used but it does not answer the question which of the indexes is better. The best index leads to the least cost for post processing. For the reader it may be obvious that i_1 performs considerably better because one can assume that there are less books with the title XML than books written in 2005. But from the syntactical and structural point of view i_1 and i_2 are identical with a minor variation in one label. The size of the results returned by the indexes that is necessary to estimate the postprocessing expenses cannot be derived from the indexes' definition. Therefore, the decision which index to use cannot be made without further information of the indexed data.

Statistic DataGuide

In order to solve this problem we extract statistical information of the frequency of elements and the distribution of their values. This information is used by the query optimizer to determine the number r of elements that are selected by a path expression. The *selectivity* sl is the rate of different elements that share the same value. For elements with unique values sl is $\frac{1}{r}$. For elements that have no unique values (e.g. `year`) we assume an equal distribution leading to $sl = \frac{1}{d}$ with d the

number of different values. This is a first approximation that may lead to inaccuracies if the distribution is unbalanced. A better but more space consuming approach would use histograms to model the distribution of elements values.
The product $sl \cdot r$ calculates the expected value of the number of elements that are returned by a key-query. For an element with a unique value it holds that $sl \cdot r = \frac{1}{r} \cdot r = 1$.
With different cost models the query optimizer is able to estimate the query execution time with an index (logarithmic) and without an index (linear) and choose a plan with minimal total costs.

The basic data structure that we use to store the required statistic information r and sl is close to a Strong DataGuide, i.e. it builds extents of elements that are reachable by the same linear path expression. In contrast to the Strong DataGuide we do not store elements or references to elements in the extents but a tuple $< r, sl >$ consisting of the average number of this element per parent and its selectivity. The selectivity is nil if the element has no values.
Figure 5.7 shows the statistic DataGuide for a randomly selected 10MB fragment of the DBLP data.

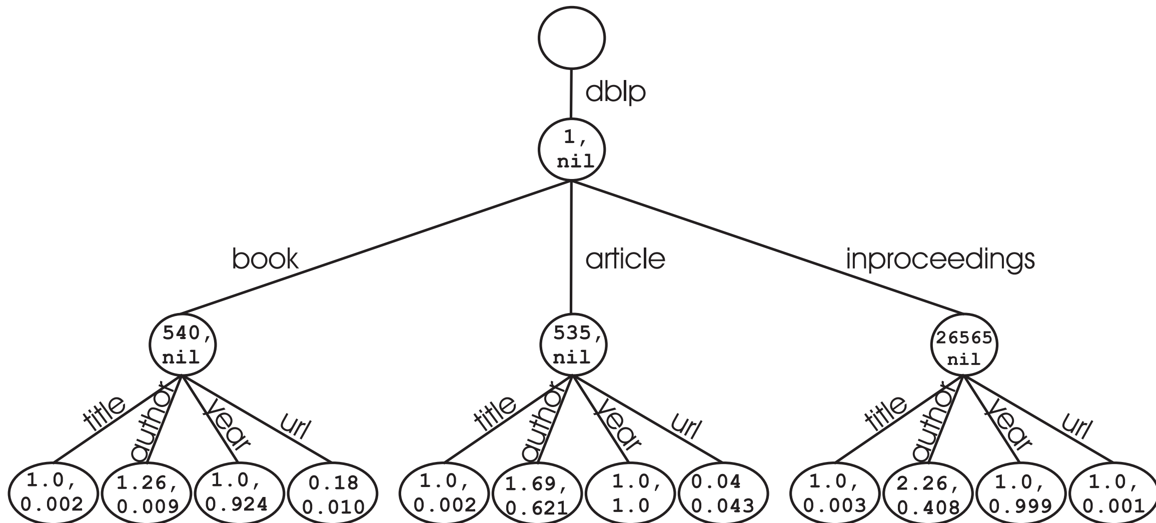


Figure 5.7: A statistic DataGuide for a DBLP fragment.

The $/dblp$ extent indicates that its root (the document root) has exactly one child with the label `dblp`. Because the `dblp` element has no values its selectivity is nil. The `dblp` element has 540 children with the label `book`. This is reflected in the extent for the path expression $/dblp/book$. Each of these `book` element has one `title` child, in the average 1.26 `author` children, one `year` child and in the average 0.18 children with the label `url`. These values can be taken from the corresponding extents. The selectivity of the `title` values is very low (0.002) indicating that most books have different titles. In contrast, the selectivity of the year values is relatively high (0.924) indicating that the 540 books share most values for years. In

particular there are only 41 different values for the `year` element of the books. The selectivity of 1.0 for the $/dblp/article/year$ path expression indicates that all values are the same. The selectivity for the other elements that have values in this DBLP fragment can be seen in the figure.

The values r and sl are used when estimating the number of elements that correspond to a path expression to be evaluated. This number is the result of a multiplication of all r values on the path to the selected element. If the path expression contains a key comparison the value for its selectivity is multiplied additionally.

Example 14 The path expression $/dblp/article/author$ will lead to $1 \cdot 535 \cdot 1.69 = 904$ selected elements.
The same path expressions with a value comparison ($/dblp/article/author[. =' x']$) leads to $904 \cdot 0.621 = 561$ selected elements. This number is relatively high because the 535 articles in the selected DBLP fragment are written by only 342 different authors.
The similar path expression $/dblp/article/title[. =' x']$ leads to only $1 \cdot 535 \cdot 1 \cdot 0.002 = 1.07$ hits. Therefore, querying an article by its title is more than 300 times faster than querying it by the authors.

If the query contains a wildcard (*) or the descendant axis (//) multiple extents must be regarded. The values for r can be summarized in order to get the final result. With a key in the path expression the r values of the different affected extents have to be weighted by the selectivity sl before summarizing them.

Example 15 The path expression $/dblp/*/url$ affects three `url` extents of books, articles and inproceedings. We calculate their numbers independently and summarize them afterwards. Therefore, the result of this path expression has the expected cardinality $1 \cdot 540 \cdot 0.18 + 1 \cdot 535 \cdot 0.04 + 1 \cdot 26565 \cdot 1.0 = 26684$. □

The statistic DataGuide is a relatively simple but in most cases sufficient and efficient approach to estimate the cardinality of selected elements of a path expression. The particular value can be used in cost models for indexes and conventional XPath evaluation.

The approach assumes statistical independence between elements in the XML data. If we have elements that are statistically dependent, for instance they are mutually exclusive the statistic DataGuide will lead to reduced precision: For instance, an element X has two children a and b. Half of the X elements have exactly one a child and the other half has exactly one b child. Therefore, no X element has both an a and b child. The statistic DataGuide would assign $r = 0.5$ for the a and b extent. The path expression $//X[a \textit{ and } b]$ would lead to an estimated cardinality of $|X| \cdot 0.5 \cdot 0.5 = \frac{|x|}{4}$ indicating that a quarter of the X elements have both an a and b value.

For this reason the statistic DataGuide is an early approach that needs further refinement. An important question is the updatability of this approach when the underlying XML data is modified. Anyhow, the problem of rating different indexes with additional statistical information occurs only if we have multiple indexes that are able to execute the query. Therefore, a huge amount of database application will probably perform well even without any statistical ranking of indexes.

5.4.5 Algorithm for the Query Execution

The process of the query execution is organized in three phases: the selection of indexes, the key retrieval and the optional postprocessing. Figure 5.8 illustrates these phases. In the following we describe the phases with pseudo code.

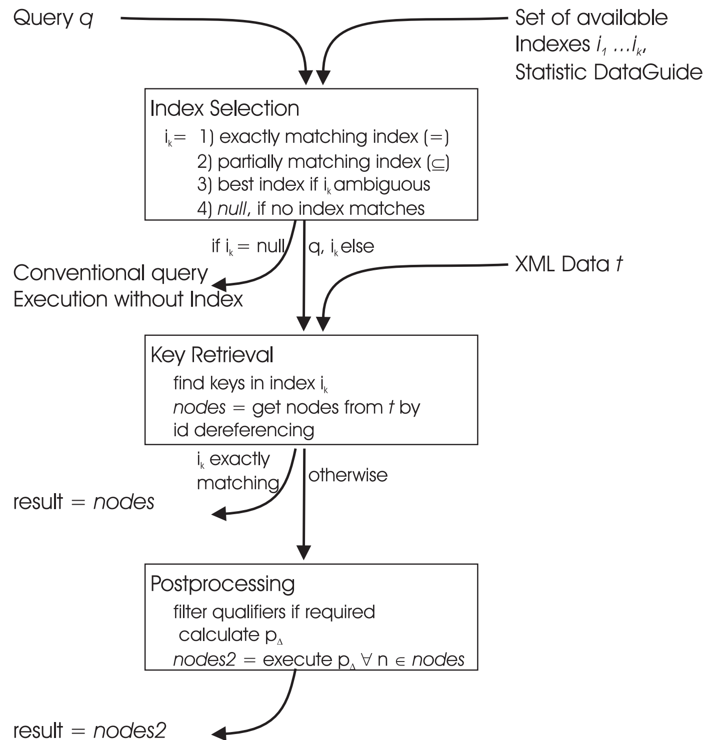


Figure 5.8: The three phases of the KeyX query execution

1. Phase 1: Index Selection
   The first phase is responsible for determining an index j that matches with the query q. First, the algorithm extracts the path expressions for the keys, qualifiers and return values of the index declaration and the query. If all path expressions are equal an optimal index is found and returned. If no such index exists the algorithm tries to find indexes that can be used due to the containment relationship. If only one subset index is found it is returned instantly. If multiple indexes are found the algorithm requests an advise from the statistic DataGuide.

```
1  Index getIndex(IndexCollection indexes, Xpath q, StatDataGuide sdg){
2    subsetIndexes = new List();

4    Kq = q.getKeyPaths;
5    Qq = q.getQualifierPaths;
6    vq = q.getReturnValuePath;

8    forall index j indexes{
9      indexdecl = index.getDeclaration();
10     Kj = indexdecl.getKeyPaths;
11     Qj = indexdecl.getQualifierPaths;
12     vj = indexdecl.getReturnValuePath;

14     if (Kq==Kj AND Qq==Qj AND vq==vj)
15       return j;                    // Exactly matching index found

17     else if (Kj subset Kq AND Qj subset QQ)
18       subsetIndexes.add(Kj);
19   }

21   if (subsetIndexes.size ==1)
22     return subsetIndexes[1];
23   else
24     return sdg.findBestIndex(subsetIndexes,q);
25 }
```

Figure 5.9: Phase 1: Index Selection

2. Phase 2: Key Retrieval
   In the second phase the key(s) are searched in the tree structure of the index; like in relational indexes this is done in logarithmic time. If the key is found the attached value is either the reference to the corresponding node in the XML data (single-key index) or a search tree of a lower level (multi-key index) so that further (recursive) key retrieval is performed. The references are dereferenced to XML nodes and returned as the result of the query.

```
1  public XMLNOdeList evaluateQuery(Index i, XPath q, XMLData t){

3    tree = i.getSearchTree();
4    keys = q.getKeyValues();
5    int ids = retrieval(tree, keys);

7    nodes = new List();
8    forall id in ids{
9      node = t.getXMLNodeByID(id);
10     nodes.add(node);
11   }
12   return nodes;
13 }



17 private IDList retrieval(SearchTree tree, keys k){

19   if (isEmpty(k)){
20     if (tree is single node) return tree.id;
21     else return getAllIds(tree);
22   }
```

```
24     else{
25       it2 = tree.findKey(k[1]);
26       return retrieval(it2, k-k[1]);
27     }
28 }
```

Figure 5.10: Phase 2: Key Retrieval

3. Phase 3: Postprocessing
   Phase three receives XML nodes that may require postprocessing if the index j is not fully matching the query q. First, if the query contains more keys than the index, the corresponding values are fetched and filtered from the XML data. The same happens if the query contains additional qualifiers that must be fulfilled by the XML nodes. Finally, if the return value of q has a different path expression than the nodes that are returned by the index, the difference path expression p_δ is computed and evaluated on each node.

```
1  public XMLNodeList PostProcessing (XMLNodeList input,
2                                     XPath q,
3                                     XMLData t,
4                                     Kj, Qj, vj) {
5    XMLNodesList out = new List();
6    forall node in input{

8      if (Kj subset Kq){ // Check for additional keys
9        remainder = Kq-Kj;
10       if (!node.fulfill(remainder) input.remove(node);
11     }

13     if (Qj subset Qq){  // Check for additional qualifers
14       remainder = Qq-Qj;
15       if (!node.fulfill(remainder) input.remove(node);
16     }

18     if (vq==vj)
19       out.add (node);
20     else{
21       pdelta = vq-vj;
22       out.add(node.evaluate(pdelta) on t);
23     }
24   }
25   return out;
26 }
```

Figure 5.11: Phase 3: Post Processing

5.5 Performance Measurements

In this section we compare KeyX with implementations of APEX, the Strong DataGuide, Index Fabrics Raw Path, and Refined Path by performance measurements. All index implementations run on top of the native XML database management system *Infonyte DB* [55] storing the XML data persistently. Basically, any XDBMS offering an XPath query engine and a unique id for each XML node can be used. Examples include Xindice and Natix.
All implementations are realized in Java; the measurements were performed on an Intel P4 with 2,66 MHz and 1GB RAM. All index implementations provide an XPath interface that processes the queries.
Instead of testing one random database application we measured the impact of the indexes on the execution of different query types separately. For each type (single-key, multi-key, range, pure-path, descendant) we executed several representative queries on a fragment of the DBLP data. With the separate results for each query type it is possible to infer the impact of an index approach on a full application if the distribution over the query types is known. In order to obtain precise values we executed all queries up to 2.5 million times measuring the total time which is afterwards divided to get an average value for each query. The evaluated queries are presented in table 1.

	Type	XPath Expression	#Results
Q_1	Pure-path	/dblp/inproceedings	26565
Q_2	Single-key	/dblp/inproceedings/author[.='X']	1
Q_3	Multi-key	/dblp/inproceedings[author='X' AND year > y]	1
Q_4	Range	/dblp/inproceedings/title[../year > y]	4508
Q_5	Descendant	//isbn	488
Q_6	Partially matching	/dblp/inproceedings/title[../author='X']	10
Q_6		/dblp/inproceedings/title[../author='X']	approx. 10

Table 5.1: Path expressions used for the performance measurements

In the following scenario we assume that the selective indexes Refined Path and KeyX created optimal indexes for the queries Q_1 - Q_5. With query Q_6 we want to measure the performance of KeyX with partially matching queries. Therefore, there is no specific index for q_6; instead we use the indexes for q_2 and q_3 to execute q_6. With query Q_6 we want to measure the performance of KeyX with partially matching queries. Therefore, there is no specific index for q_6; instead we use the indexes for q_2 and q_3 to execute q_6. The non-selective approaches Raw Path, Apex and Strong DataGuide created their structural summaries. For every query that has a key we selected a multitude of values in order to get realistic measurements.

In the following figures we present and discuss the measurements of each query: The pure path query Q_1 has a result set of 26565 nodes. They have to be fetched from the database by their ids. As this database access is the most expensive process the query execution times are similar for all approaches. Because Q_1 has

no predicate it cannot be executed by a Raw Path. For a query with a huge result set (e.g. all publications written after 1975) it makes no significant difference whether we use an index or not.

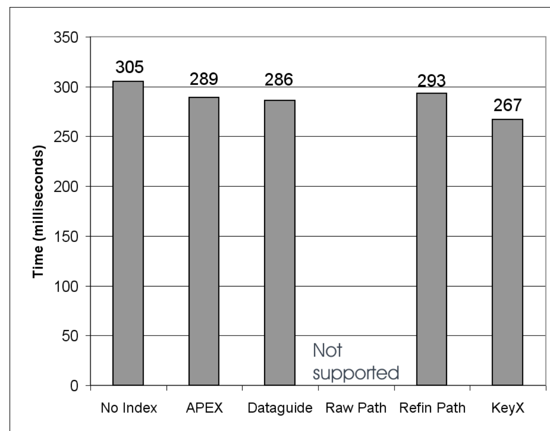


Figure 5.12: Performance measurements of query Q_1

The query Q_2 returns only one node that has a defined unique key. The structural summaries cannot use the key directly and have to check all `author` nodes whether they contain the required value. This linear complexity leads to an execution time which may exceed the execution time without index. The key-based approaches retrieve the key from their internal data structure and can therefore execute the query significantly faster. Please note the logarithmic scale in some of the following figures. The two Index Fabrics approaches are a little bit slower than KeyX because they need to look up the designator of the path and concatenate it with the predicate. Additionally, the resulting key has to be encoded to binary format for the lookup operation in the Patricia Trie.

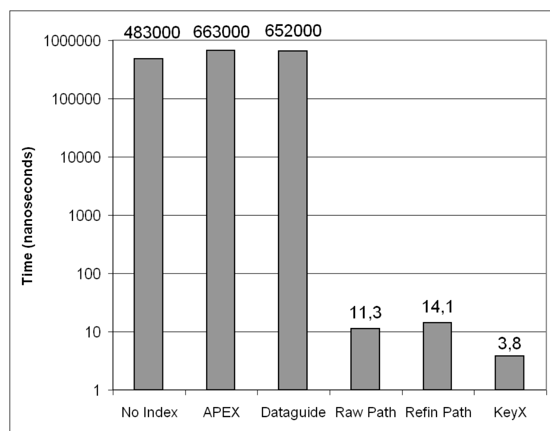


Figure 5.13: Performance measurements of query Q_2

Query Q_3 is a multi-key query with one key comparison and a range query upon the second key. The structural summaries have the same disadvantage as in query Q_2 leading to poor results. Both Index Fabric approaches cannot support a multi-key query with a value range directly; our Index Fabrics implementation executes such queries universally by splitting a multi-key query in multiple single-key queries. The results have to be intersected afterwards to find the relevant nodes. KeyX can naturally support Q_3 by its internal data structure that is based on nested trees.

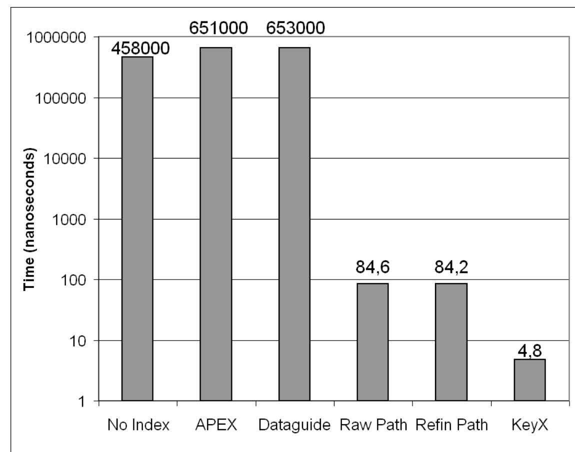


Figure 5.14: Performance measurements of query Q_3

A range query upon a single-key is performed by Q_4. The Refined Path is close to KeyX but like in Q_1 the high cardinality of the result set leads to a significant deceleration for both approaches. The Raw Path is significantly slower because it has to navigate to the sibling node `title` for all of the 4508 `year` elements of the result set.

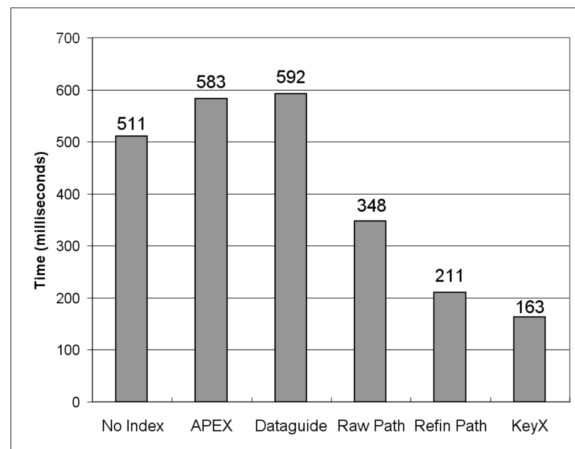


Figure 5.15: Performance measurements of query Q_4

The descendant query Q_5 selects all `isbn` nodes without regarding their leading path. Because Q_5 has no key it cannot be executed by a Raw Path. The other approaches provide similar results. Due to the medium size of the result set the process of finding relevant nodes is more expensive than fetching them from the database. On the other hand, the navigation inside the structural summaries is performed so fast that there is no significant difference between APEX and the Strong DataGuide. As the DBLP data is relatively high structured it contains only few element types (e.g. inproceedings, title, isbn). Thus, the extent of one element type can quickly be retrieved.

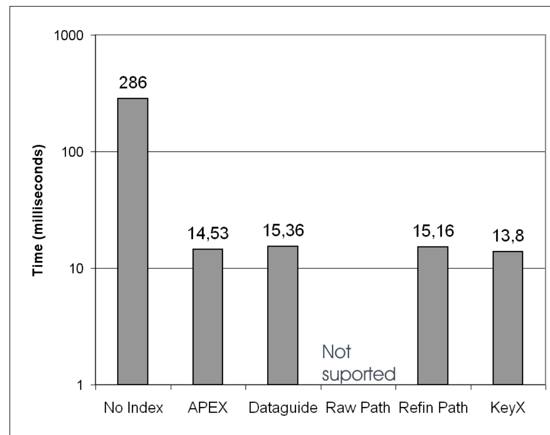


Figure 5.16: Performance measurements of query Q_5

Partially matching queries are evaluated by the query q_6 that is executed upon the indexes i_{q2} for q_2 and index i_{q3} of q_3. In both cases the execution time lasts a bit longer than by using an optimal index. When executing q_6 upon i_{q2} the relative navigation step $../title$ is executed on the elements which are referenced by the result of i_{q2}.

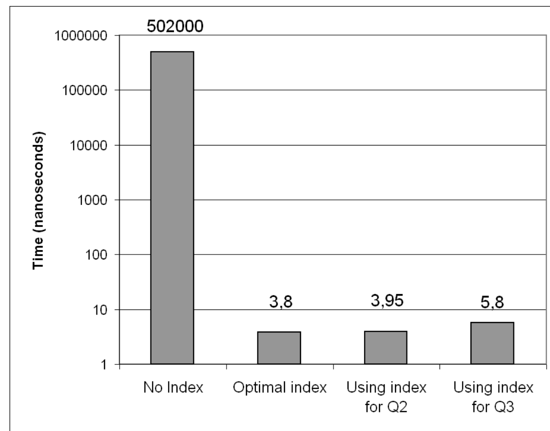


Figure 5.17: Performance measurements of query Q_6

Executing q_6 upon i_{q3} means that the subtree of `year`-references below the `author` key have to be unified. Because i_{q3} returns references to `inproceedings`- elements the additional navigational step $/title$ has to be performed on the database. This explains why i_{q3} performs poorer than i_{q2} for q_6. But compared to the execution without any index it still makes sense to use existing indexes for partially matching queries.
Queries containing the wildcard operator are supported by KeyX like descendant queries resulting in equivalent query execution times. As our implementations of Index Fabric, Apex and the Strong DataGuide do not support wildcard queries, currently we cannot present measurements at this place. Theoretically, if the wildcard query contains no predicate the structural summaries will perform well as they can find the relevant extents quickly. For queries with a predicate they will have to compare all nodes with a value leading to poor times. The Raw Path cannot support wildcard queries as the total path is encoded to one string. The Refined Path idea allows the definition of a specific index for this query with an expected profit comparable to Q_5.

Figure 5.18 shows the KeyX query execution time of Q_2 in dependency of the document size. These measurements were performed upon randomly selected inproceedings of the original DBLP data. A query over 30,000 inproceedings takes roughly 0.5 seconds without an index. This response time is unacceptable for huge databases like the DBLP containing 500,000 publications. The execution time of KeyX grows logarithmically as the underlying data structure (a Java TreeMap) provides logarithmic key retrieval time.

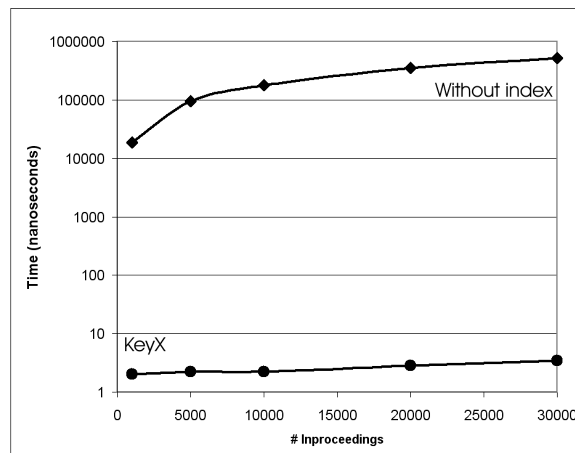


Figure 5.18: KeyX complexity measurements

Figure 5.19: Picture taken from [72].

Chapter 6

The Index Selection Problem

Today's larger business applications with an underlying database cannot exist without indexes covering the most frequent queries. Recently, relational database management systems (RDBMS) are dominant although XML database management systems (XDBMS) are becoming more and more relevant for real world applications.
Defining suitable indexes is a major task when optimizing the database. Usually, a human database administrator (DBA) defines a set of indexes in the design phase of the database. This can be done manually or with the help of so-called *index wizards* or *index adviser* tools that analyze predefined or collected database operations. In both cases the typical usage of the database has to be known in advance.

But, even having an optimal set of indexes when setting up a database there is no guarantee that these indexes will suit future demands. Rather, it is realistic that the typical usage of the database will change after a while because new queries appear, for instance. In consequence, the existing indexes are suboptimal.

The typical approach to face this problem is that a database administrator maintains the database permanently: she logs the workload, analyzes the performance of the database and the existing indexes, and redefines indexes when necessary. These tasks are time-consuming and require a skilled expert.

In XML database management systems (XDBMS) this problem becomes even worse. In relational databases (in first normal form 1NF) only the atomic values of specified columns are reflected in an index; in contrast, XML indexes have to cover both the structure of the data and the values of elements and attributes; queries and modifying operations may contain structural and content properties. Therefore, the number of possible queries and indexes is significantly higher than in relational databases.
Additionally, for XML data without a schema like a DTD or XML Schema, queries and indexes cannot be defined finally in advance because elements may appear

and disappear at any time. Both facts tend to result in higher maintenance costs for XML indexes compared to relational indexes. For large databases with a complex structure where a multitude of queries is expressible (e.g. DNA databases) it may be impossible to determine the best indexes manually.

Beside these more technical issues there are more good reasons for self optimizing and self configuring DBMS. Studies have shown that the cost to manage storage is typically twice the cost of the actual storage system [33, 98, 97]. In the field of databases 81% of the costs are people costs [1]. Figure 6.1 shows the distribution of the total costs of ownership.

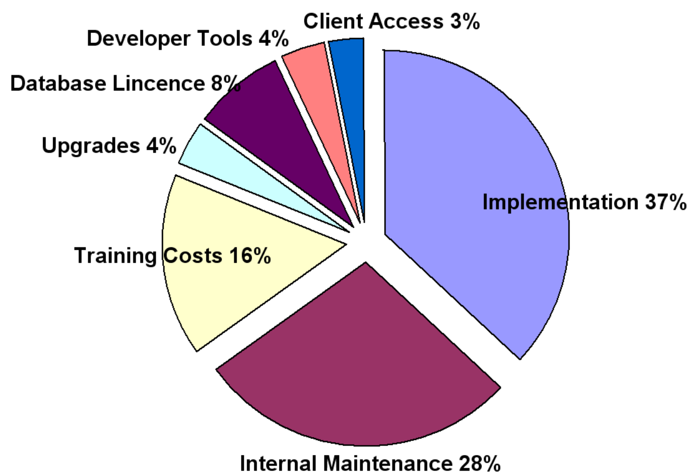


Figure 6.1: Costs of running a DBMS

With the help of a self-optimizing and self-configuring DBMS these costs can be cut to a great extent.

In this chapter we show how a native XDBMS enriched by KeyX indexes can find good indexes for a workload of database operations. Additionally, we present how KeyX can be used to turn a DBMS into an autonomous and adaptive database management system that analyzes its workload periodically and creates/drops XML indexes automatically. This approach guarantees a high performance over the total life-time of a database. Although we present our index system *KeyX*, the idea and the main results in this chapter are transferable to other XML indexing approaches.

6.1 Introduction to the Index Selection Problem

In this section we introduce the well-known *Index Selection Problem* (ISP). The selection of optimal indexes is an important task in physical database design to improve the performance of a database system. Indexes reflect the values of

specified attributes (e.g. columns in a RDBMS) in a data structure, like B-trees, optimized for fast key retrieval.
In a relational database, the indexed attributes are columns, whereas in a semistructured or XML database attributes are marked tree nodes or XML elements.

The optimal index configuration is the one with minimal query processing time for the whole workload. Defining an index upon all columns leads to fast retrievals but increases the execution time of modifying database operations like an insert. Additionally, such an index configuration would lead to exhaustive storage costs.
By workload, we mean a set of operations accessing and modifying the database, such as SQL statements, for instance. Each built index requires a maintenance time if the database is modified, while each query is accelerated if performed upon an index. An index can only improve data retrievals of the specific attributes that the index is built upon.

The main goal is to minimize the overall time needed to process the whole workload. An important constraint of the ISP is that the consumption of memory of all realized indexes must not exceed a predefined limit. In this vein the ISP is comparable to the *Knapsack problem*.

The ISP is an inherently computationally difficult problem belonging to the class of NP-complete problems which was proven in [20]. The ISP is a generalization of the *Uncapacitated Facility Location Problem* (UFLP), which is known as NP-hard. The reader is referred to [3] for details of NP-complete problems. Simplified, one can say that there is no deterministic algorithm to solve this problem in less than exponential time (under the assumption NP $\neq$ P).

In a database consisting of n columns there are 2^n possible index configurations for single column indexes as every column may be indexed or not. If multi column indexes are possible there are a lot more configurations. This shows that the computation of an exact solution is mostly too expensive or impossible for larger n. Approximative algorithms like heuristics or other approaches are required to find a (suboptimal) approximation of the exact solution.

6.2 Index Selection in Relational DBMS

In this section we describe how commercial database products have integrated tools or features to optimize the query execution time by suggesting index configurations. This is mostly done by implementing a heuristic solution for an ISP because an exact algorithm cannot be computed in appropriate time because of the inherent complexity of the problem. The fact that all important database manufacturers have implemented index tuning tools shows that the ISP is well

understood for relational databases, although different approaches have been realized in the implementations.

The *Microsoft Index Tuning Wizard* (a.k.a *Autoindex selection tool)* [14, 15, 16] is a tool for the SQL Server which automates the task of selecting suitable indexes for a database and a given workload. The workload is provided by a database administrator who can either collect real database operations or create an artificial workload which represents typical database operations. Based on this workload the Tuning Wizard calculates an ISP solution and suggests a set of indexes to be created. Expected changes of the database performance and storage consequences are reported as well. The tool evaluates the estimated costs of an index using so-called what-if indexes. What-if indexes are indexes that are not yet created but are already known to the query optimizer of the SQL Server. As the tool is used in productive use, the authors have invested a lot of efforts in efficient heuristics to solve the ISP and to determine the costs of an index configuration. To minimize the set of index candidates in the configuration, in a first step the Tuning Wizard determines the best indexes for each query independently. Other indexes are abandoned as they are unlikely part of the optimal index configuration for the whole workload. Additionally, the heuristic starts with single column indexes (having only one key attribute) and increases the number of key attributes step by step.

IBMs *DB2 advisor* [111] for the DB2 Universal Database utilizes a component in the optimizer that recommends indexes for a given workload. The ISP is interpreted as a 0-1 integer linear programming problem with the indexes by 0-1-flags indicating if an index is selected (1) or not (0). The solution of the ISP is a sequence of 0-1-flags determining the index configuration. As long as the user's time budget is not exhausted DB2 creates new random configurations and memorizes the one with minimal costs calculated yet. This approach allows the user to define a time restriction to the ISP solving algorithm.

Oracle's *Tuning Pack* [94] has a component called *Index Tuning Wizard* which has comparable functionalities. Based on an SQL workload, changes on existing indexes are proposed. Unfortunately, we could not find details about the implementation and the used heuristics.

In the last decades a multitude of heuristics for the ISP where presented in academia: In [12] two solutions are presented to determine an optimal index configuration for relational databases. The first approach generates an exact solution and is based on a branch and bound tree algorithm reducing the set of index configurations considered to be part of the optimal solution. The exact solution of the ISP is a sequence of 0-1-flags indicating which index is selected (1) or not (0). The ISP is interpreted as a 0-1 integer linear programming problem and

a solution is computed in polynomial time using Khachian's algorithm [63]. The solution is a sequence of values between 0 and 1. If some values in this sequence are exactly 0 or 1 they are transfered to the final optimal solution. Fractional values different from 0 and 1 cannot be part of the final solution as they do not indicate whether the affected index is selected or not. These values have to be fixed to 0 or 1 by evaluating both possibilities. This leads to a decision tree which grows exponentially in the number of fractional values. This approach performs well if the linear programming algorithm generates a solution with only few values different from 0 and 1.

The second approach discussed in the same paper uses a suitable knapsack heuristic to find a suboptimal solution. Suboptimal means that the approach does not find the best index configuration but one which comes very close to it. The initial point is the solution of the 0-1 integer linear programming problem described above. A value differing from 0 and 1 is interpreted as the likelihood of an index to be part of the best index configuration. These indexes are sorted according to decreasing values of the likelihood. The indexes which are most likely to be part of the best index configuration are selected until the required space exceeds the available space. This approach is faster than the decision-tree algorithm as it can be computed in polynomial time.

A more or less exotic approach to solve the ISP using a polynomial genetic algorithm is introduced in [66]. Genetic algorithms are stochastic search methods that are motivated by the biological evolution. A suitable index configuration for each query of the workload is interpreted as an individual with its time gain as fitness factor. The fittest individuals are randomly recombined (to simulate a biological crossover) in order to find better index configurations. A random change of a selected index in an index configuration represents a mutation and leads to a modified individual which may have a better fitness. As only the best index configurations survive there will be good solutions of the ISP after some generations.

6.3 Index Selection Problem Applied to KeyX Indexes

In this section we describe the Index Selection Problem applied to KeyX indexes. Given a workload of path expression based database operations we extract possible indexes and define an optimization problem to find the best index configuration for the whole workload.

A database operation is an operation for querying, updating, deleting, and inserting data stored in a database.

Definition 20 (Database Operation)
A *database operation* o is a tuple (p,t,i) consisting of a path expression $p \in XP^{\{[],*,//\}}$ a type t and additional information i if it is a modifying operation[1]. The set of all database operations is denoted by O. □

The path expression p of a database operation o can be extracted using the function $path : O \rightarrow P$. Analogously, the type t of an operation o is determined by the function $type : O \rightarrow \{\textit{query}, \textit{insert}, \textit{change}, \textit{delete}\}$.

All database operations which shall be relevant for the Index Selection Problem are collected in a bag called workload.

Definition 21 (Workload)
The *workload* is a finite bag of database operations. In our work we assume that there are n operations in the workload W.

$$W = \{o_1, o_2, \ldots, o_n \mid o_j \in O \wedge 1 \leq j \leq n\}$$

A database administrator may create a workload that represents the typical usage of the database application. Another way is to collect all database operations that occur in a given time period.

6.3.1 Index Candidates

The following function enumerates every possible index declaration $d \in D$ in a set that can be built upon the key and value nodes (index declarations are defined in section 5.1.1).
The key, qualifiers, and value paths of a database operation $o_j \in W$ are determined by calling the path extraction functions introduced in section 5.1.2.

[1]For instance, if t is an insert operation, i contains the nodes to be inserted.

Definition 22 (Index Candidates)
The *index candidates* are defined as a function $ican : P \rightarrow \mathcal{P}(D)$ returning a set containing all possible index declarations for a given path expression p. The following definition combines and permutes the key nodes:

$$ican(p) = \{([k_1, k_2, \ldots, k_m], \textit{value}(p)) \quad | \quad k_j \in \textit{key}(p) \ \wedge \\ 1 \leq j \leq m \wedge 1 \leq m \leq |\textit{key}(p)|\} \qquad \square$$

In total we have to consider $\sum_{n=0}^{m} \frac{m!}{(m-n)!} - 1$ different possible indexes. As most of these indexes are dropped during ISP calculation, we call them *index candidates* of the path expression p. Index candidates are virtual and not materialized in the database.

Example 16 A query o with the path expression

$$p_5 = /dblp/article[author = "X" \text{ and } title = "Y"]$$

has the following key and value nodes:

$$\begin{aligned} key(p_5) &= \{/dlbp/article/author, /dlpb/article/title\} \\ value(p_5) &= /dlbp/article \end{aligned}$$

All index candidates are listed below. As the order of key nodes matters, the first two index candidates of $ican(p_5)$ are not equivalent.

$$\begin{aligned} ican(p_5) &= \{i^1_{p_5}, i^2_{p_5}, i^3_{p_5}, i^4_{p_5}\} \text{ with} \\ i^1_{p_5} &= ([/dblp/article/author, /dblp/article/title], /dblp/article) \\ i^2_{p_5} &= ([/dblp/article/title, /dblp/article/author], /dblp/article) \\ i^3_{p_5} &= ([/dblp/article/author], /dblp/article) \\ i^4_{p_5} &= ([/dblp/article/title], /dblp/article) \end{aligned} \qquad \square$$

The two multi-key indexes $i^1_{p_5}$ and $i^2_{p_5}$ constitute the best suitable indexes for p_5 as they reflect both key nodes. In contrast the two indexes $i^3_{p_5}$ and $i^4_{p_5}$ require additional processing of the referenced nodes, but are still more efficient than an evaluation of the plain path expression without an index. Please notice that the number of index candidates grows exponentially with the number of key nodes of a path expression and increases the costs of solving the ISP dramatically. A path expression with 4 key nodes will lead to 64 index candidates! Heuristics to decrease the computational expense have to start at this point by reducing the number of index candidates.

To consider the whole workload we need to regard the index candidates of all database operations $o \in W$. This is done by unifying the index candidates of all operations to the total index candidate set.

Definition 23 (Total Index Candidates)
The index candidates of all database operations of the workload W are unified to the *total index candidates* set TIC_W:

$$TIC_W = \bigcup_{o \in W} ican(\textit{path}(o))$$

The set TIC consists of all (l) possible index declarations $d_1, \ldots, d_l \in D$ which are relevant for the workload W.

$$TIC = \{d_1, d_2, \ldots, d_l\} \text{ with } d_j \in D \text{ and } 1 \leq j \leq l$$

The set TIC is constant for a given workload and stays unchanged while exploring the ISP. The number of index candidates in TIC may be less than the sum of all index candidates in all sets of $ican(o)$ as they may contain duplicates. Nonetheless the number of indexes in TIC grows exponentially.

$$|TIC_W| \leq \sum_{o_j \in W} |ican(\textit{path}(o_j))|$$

6.3.2 Index Configuration

A lot of the index candidates of TIC are dropped when calculating the set of indexes of TIC which is optimal for the workload. A set of indexes is called index configuration.

Definition 24 (Index Configuration)
An *index configuration* c points out which index declarations of TIC are materialized and available when executing the workload. A configuration c is a vector of flags identifying which index candidates from TIC are selected or not.

$$c = (f_1, f_2, \ldots, f_l) \text{ with } f_j \in \{0, 1\} \text{ and } 1 \leq j \leq l$$

The j-th flag f_j identifies if the j-th index i_j from TIC is selected ($f_j = 1$) or not ($f_j = 0$). The set of all possible index configurations is denoted by $CONF_{TIC_W}$. □

Example 17 The configuration $c_1 = (1, 0, 0, 1, 1)$ indicates that the first and the last two indexes are materialized while the second and third are inactive. □

Because every combination of indexes is a different index configuration it is obvious that

$$|CONF_{TIC}| = 2^{|TIC|}$$

6.3.3 Cost Functions

In order to find the best index configuration for the workload W we have to compare the costs and profits of all index configurations. Materializing each index

declaration and executing the whole workload for each index configuration will lead to enormous space consumption and prohibitively high computational expenses. Therefore, the costs of an index configuration have to be estimated.

When evaluating a path expression p with the help of an index i, we first have to find the requested keys. In a data structure like a Red-Black-Tree search tree or a B*Tree this time depends logarithmically on the number n of stored keys. The corresponding function is

$$cost_{t_{key}}(i) = \alpha_l \cdot \log |i_k| + \beta_l$$

with $|i_k|$ denoting the number of keys in i and α_l, β_l two constants that depend on the underlying system. The number of keys can be extracted from the XML data or it can be computed with the *statistic DataGuide*, a statistical summary of the XML data introduced in section 5.4.4. The constants are determined by test runs and do not change significantly until the underlying hardware is modified (e.g. a faster processor).

Having an index i and the cost function $cost_{t_{key}}(i)$ it is possible to estimate the time to find a key in i. This key contains one or multiple references to nodes in the original XML data stored in the database. Because the result of a path expression contains these nodes we have to retrieve them from the database. In general, this includes read operations from the harddisk leading to higher expenses. The corresponding function is

$$cost_{t_{deref}}(i) = \alpha_a \cdot \frac{|i_n|}{|i_k|} + \beta_a$$

with $|i_k|$ denoting the number of keys in i, $|i_n|$ denoting the number of referenced nodes in i. Again, α_a and β_a are system-dependent constants that are determined once by initial test runs. In this function we assume that each key has an average of $\frac{|i_n|}{|i_k|}$ references. Again, the estimated number of references per key can be extracted from the statistic DataGuide.

The total costs of an index are raised by retrieving the key and dereferencing the key's content. Therefore, it holds that

$$cost_{time}(i) = cost_{t_{key}}(i) + cost_{t_{deref}}(i)$$

When evaluating a path expression without an index the execution time can be measured and does not need to be estimated. Anyhow, the statistic DataGuide that summarizes the frequency of elements can be applied to calculate the estimated number of elements selected by a path expression. Without an index we have to use a linear cost model instead of a logarithmic one.

The storage costs of an index is linear to the number of stored keys and their content. Therefore, we estimate the storage costs of an index by the function

$$cost_{space}(i) = \alpha_d \cdot n + \beta_d \cdot k$$

with α_d the required space to store a reference and β_d the space to store a key. Again, α_d and β_d are system-dependent constants. When installing the XDBMS, indexes of different sizes are created, queried, and stored to determine the constants. Particular values for the system-dependent constants can be found in [32].

6.3.4 Costs of a configuration

The evaluation of costs for a given configuration is performed by the query optimizer. The costs of a configuration relies on costs of some selected indexes in the configuration. The query optimizer calculates them with function $cost_{time} : P \times I \rightarrow \mathbb{R}$ defined on a path expression p and a selected index i. Note that a selected index does not mean that the index is established and filled with values. Therefore, the query optimizer needs the ability to 'simulate' the presence of indexes in the database.

In addition the query optimizer is responsible for the execution of a database operation. A query execution plan is prepared for each query including the choice of the best suitable index. This decision is made by a function named *best* : $P \times CONF \rightarrow I$, determining the best suitable index for a path expression and a given index configuration. A second function called *aff* : $P \times CONF \rightarrow \mathcal{P}(I)$ identifies the set of all affected indexes for the path expression of a given modifying operation. Both functions depend on the containment relation for simple path expressions.

Using these functions the query optimizer can determine the costs of a configuration.

Definition 25 (Costs of a Configuration)
The evaluation time cost for a given database operation o and a given configuration c is defined as follows:

$$cost_{time}(o, c) = \begin{cases} cost_{time}(\mathit{path}(o), \mathit{best}(\mathit{path}(o), c)) & \text{if } \mathit{type}(o) = query \\ \sum_{i \in \mathit{aff}(\mathit{path}(o),c)} cost_{time}(\mathit{path}(o), i_i) & \text{if } \mathit{type}(o) \in \{insert, \\ & \quad update, delete\} \end{cases}$$

with $o \in O$, $c \in CONF$ □

A query operation o is served by the best selected index *best*$(\mathit{path}(o), c))$ that is available in the current index configuration. Other selected indexes which may accelerate the query less than the best index can be ignored as it suffices to serve the query by one index. On the other hand, operations that modify the

database (insert, update, and delete) lead to an update of all affected indexes in the configuration.

Example 18 Assume we have an index configuration with 3 selected indexes i_1, i_2, i_3 with:

$$\begin{aligned} i_1 &= ([/dblp/article/author], val), \\ i_2 &= ([/dblp/article/title], val) \text{ and} \\ i_3 &= ([/dblp/article/author, /dblp/article/title], val) \end{aligned}$$

where $val \in P$ is an arbitrary path to a value node.
The path expression $p = /dblp/article/autor$ has i_1 as best index and $\{i_1, i_3\}$ as set of affected indexes if changes occur to a node with path p. □

6.3.5 Index Selection Problem

The *Index Selection Problem* looks for the best index configuration $c_{solution} \in CONF$. The workload is executed for each index configuration while the one with minimal costs is memorized. Configurations which exceed the space limit $maxspace$ are dropped.

Definition 26 (Index Selection Problem)
The configuration with minimal costs of time that fits the space restrictions $c_{solution}$ is defined by:

$$\begin{aligned} c_{solution} &= \min_{c \in CONF} \left[\sum_{o \in W} cost_{time}(o, c) \right], \\ cost_{space}(c_{solution}) &\leq maxspace. \end{aligned}$$

Given k indexes in the candidate set TIC we have 2^k different possible index configurations which have to be regarded to find the exact optimum. As we have at least $\sum_{n=0}^{m} \frac{m!}{(m-n)!} - 1$ index candidates in TIC a naive algorithm solving the ISP has to evaluate $2^{\sum_{n=0}^{m} \frac{m!}{(m-n)!} - 1}$ configurations to find the exact solution. □

6.3.6 Exact Algorithm and Heuristics

The equation of the Index Selection Problem can easily be transformed into a naive algorithm which finds the exact solution in exponential time.
The algorithm, shown in figure 6.2 is split into three parts. The first loop generates the total index set TIC by extracting the key and value nodes from the database operation of the workload W. These nodes are used to generate all possible index candidates which are unified to the set TIC. In the following step all configurations are enumerated in $CONF$ by permuting the indexes in TIC. The last step walks through all configurations in $CONF$ and compares the costs and

```
1.  TIC = {}, W ≠ {}
    for all o ∈ W {
      p = path(o);
      k = keys(p);
      v = val(p);
      TIC = TIC ⋃ ican(p);
    }
2.  Create CONF by permuting all indexes ∈ TIC
3.  Set c_solution = nil; cost_solution = ∞
    for all c ∈ CONF {
      c_t = cost_time(conf)
      c_s = cost_space(conf)
      if (c_t < cost_solution and c_s ≤ Maxspace) update c_solution and cost_solution
    }
```

Figure 6.2: Naive ISP Algorithm finding the exact solution.

the space restriction. The algorithm terminates with the best index configuration $c_{solution}$ for which the costs are minimal for the whole workload. Notice that the execution of all three steps costs exponential time; therefore, this algorithm performs poor for larger workloads with various operations.

We have implemented this naive algorithm to determine the exact solution for an ISP. In order to get a quick suboptimal solution of the ISP we have implemented the following simple greedy algorithm:
Our heuristics are similar to the approach introduced in [14]. We determine the best index i_j for each database operation o_j of the workload independently. Although this is exponential in the number of key nodes of each operation o_j this can be done quickly as most operations use less than a few keys. In a second step we calculate a profit $profit(i_j)$ for each index i_j by subtracting the execution time without indexes from the execution time performed upon i_j. The profit states what we gain in time if the index is realized. The profits of queries are positive while modifying operations have a negative profit as they lead to time-consuming index maintenance. For each index i_j having a positive profit $profit(i_j) > 0$ we add all the negative profits of database operations affecting the same index i_j. This way we calculate the total profit of an index by considering the modifying operations. If after this, the profit is still positive this index accelerates the workload execution time and may be part of the final index configuration $c_{solution}$. Indexes with negative total profit can be dropped as their maintenance costs exceed the query acceleration.

To include the storage restrictions we divide the positive profits of each index i_j by its storage costs $cost_{space}(i_j)$. These ratios are ordered in a descending list. The final index configuration $c_{solution}$ is created by taking indexes from the top of the list until the space limit is reached. This means that indexes which have a good profit with only small space requirements are selected first.

6.3.7 Evaluation and Experiments

In order to evaluate and judge our auto index approach we set up a testing environment with the general architecture illustrated in figure 6.3.

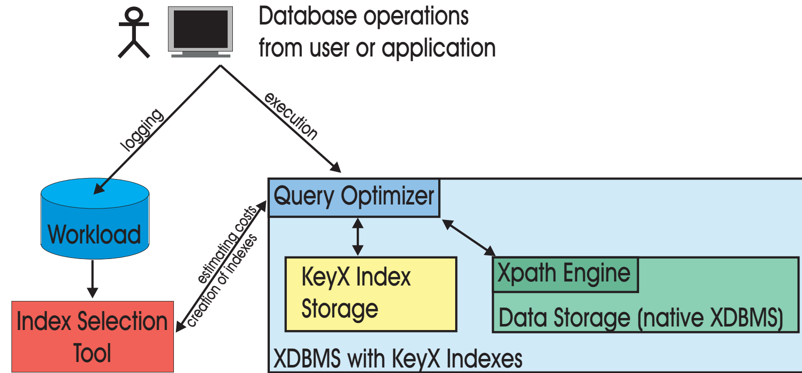


Figure 6.3: Architecture of prototypical system

The system consists of two main components: the XML database management system extended by KeyX indexes and the index selection tool. Our KeyX index enabled XDBMS [32] contains the three subcomponents *data storage*, *index storage*, and *query optimizer*. The data storage, which stores the XML data persistently, is realized by the native XDBMS Infonyte DB [55] providing an embedded XPath query engine. The query optimizer examines the path expression to be evaluated and checks if a suitable index exists in the index storage.

The index selection tool finds a good index configuration for a given workload by using ranking heuristics. For the heuristic H_{rank} we calculate the profit of each index candidate and choose the most profitable ones until the space restriction is reached. The heuristic H_{div} ranks the indexes by the quotient $\frac{profit}{spacecosts}$.

The ISP tool communicates with the query optimizer to determine the evaluation costs of a given path expression and an assumed index configuration. After calculating an optimal index configuration the index selection tool triggers the creation of this index configuration passed to the index storage.

In order to determine the quality of our index selection tool we set up a scenario with 24 different database operations. The ratio of querying and modifying operations differs in two workloads: W_1 contains only queries whereas W_2 has 33% modifying operations requiring the index to be updated. The indexes and their profits for the workloads W_1 and W_2 are shown in table 6.1. The optimal solution would suggest eight indexes with a space consumption of $28.3MB$. A space restriction of 10 MB prevents all indexes from being materialized.
We calculated an exact solution of the ISP with exponential complexity by analyzing all configurations. Because this scenario consists of only eight index candi-

dates and small workloads this can still be done in acceptable time.

	n	k	d	p_1	p_2	p_1/d	p_2/d
I_1	1,088	1,337	60.2	248.3	233.4	4.125	3.877
I_2	11,057	50,035	975.8	481.8	456.4	0.494	0.468
I_3	1,004	1,004	53.2	1,016.1	1,001.2	19.100	18.820
I_4	122,305	305,806	8,317.2	411.3	390.6	0.049	0.047
I_5	41	1,337	15.1	548.6	454.1	36.331	30.073
I_6	147,668	151,268	7,862.4	567.9	552.2	0.072	0.070
I_7	51	30,806	3,060.3	1,136.3	824.5	0.371	0.269
I_8	150,264	150,264	7,964.0	1,546.4	1,530.6	0.194	0.192
Σ			28,307.6				

Table 6.1: Characteristics of the index candidates (n: number of keys, k : number of return values, d: space consumption in kilobytes, p_i : profit of workload W_i)

A comparison between the gained profits of the exact solution and both heuristics H_{rank} and H_{div} is presented in table 6.2.

	Exact		
	selected Indices	p_Σ	d_Σ
W_1	I_1, I_2, I_3, I_5, I_8	3,841.2	9,068.3
W_2	I_1, I_2, I_3, I_5, I_8	3,675.7	9,068.3

	H_{rank}		
	selected Indices	p_Σ	d_Σ
W_1	I_8, I_3, I_5, I_2, I_1	3,841.2	9,068.3
W_2	I_8, I_3, I_2, I_5, I_1	3,675.7	9,068.3

	H_{div}		
	selected Indices	p_Σ	d_Σ
W_1	I_5, I_3, I_1, I_2, I_7	3,431.1	4,164.6
W_2	I_5, I_3, I_1, I_2, I_7	2,969.6	4,164.6

Table 6.2: Comparison of exact solution and heuristics H_{rank} and H_{div} (p_Σ: total profit, d_Σ: total space consumption)

6.4 Autonomous XML Indexing

Most of the introduced relational systems for solving the ISP like IBM's DB2 Advisor and the Microsoft Index Tuning Wizard have in common that they do not operate 'online'. This means that they have an operation phase and a design phase. The workload might be collected in the operation phase but all 'wizards' and 'advisors' have to be activated by a database administrator which decides if the index suggestions are realized.
The approach of [102] extends the functionality of the relational DBMS DB2 by automatically creating indexes without the interaction of a DBA. The system is query driven, meaning that all occurring database operations are stored in the workload which is periodically analyzed. Indexes are built, dropped, and changed during query processing so that the underlying database does not have to be stopped or switched into design mode. Changes of the query patterns are followed by changes in the index configuration of the database.

This section introduces our prototypical implementation of an adaptive XML database management system with KeyX indexes. We present scenarios and performance measurements in order to evaluate the characteristics and the ability to optimize itself. Because the indexes are created and dropped autonomously and at runtime the XDBMS becomes adaptive and optimizes itself in the background.
The basis for this is a heuristic that analyzes the current workload of XML database operations and finds a good approximation for the Index Selection Problem. This approach guarantees a high performance over the total life time of a database without the interaction of a human database administrator.

6.4.1 Architecture and Implementation

The architecture of our testing environment is basically the same as in the previous experiments as illustrated in figure 6.3. The index selection tool is now triggered periodically during the life-time of a database and tries to find a good index configuration for a changing workload of collected database operations. Therefore, we log the database queries passed to the query optimizer in a workload file. This file serves as input to the index selection tool.

In order to evaluate the introduced approach we have set up different scenarios with sample XML data that is stored persistently in the native XML database management system *Infonyte DB V3*. The measurements were performed on an Intel P4, 2.66 GHz with 256 Megabyte RAM, a 40 Gigabyte hard disk and Windows XP as operating system. All indexes were kept in main memory to exclude the time to access the hard disk. Of course, the indexes may reside on the hard disk if their size exceeds the main memory. All measurements include the time to parse the XPath based database operations, their execution and the creation of the result set consisting of corresponding nodes from the Infonyte database.

For the first two measurements we set up artificial workloads based on XMark data [104]. XMark produces scalable and highly structured data so that a multitude of different and reasonable queries are expressible. The test document has a size of 11 MB.

Test Scenario 1

For the first set we created two different classes of querying database operations: The first class A contained person-based queries while the second class B consisted of queries that operate on the items to be sold at the auction. Afterwards, we constructed several workloads with different distributions of the operations from A and B. The first workload only had operations from class A while the ongoing workloads have a growing percentage of operations from B. The last workload consists analogously of operations from B only. This scenario simulates a change in the typical usage of the database. All workloads have 100 operations in total.

In figure 6.4 we present the time measurements for execution of the workloads without an index and an index that is optimized for the first workload (only operations from A). The figure shows that the index suits well for the first time but becomes more and more useless because it cannot accelerate the operations from class B.

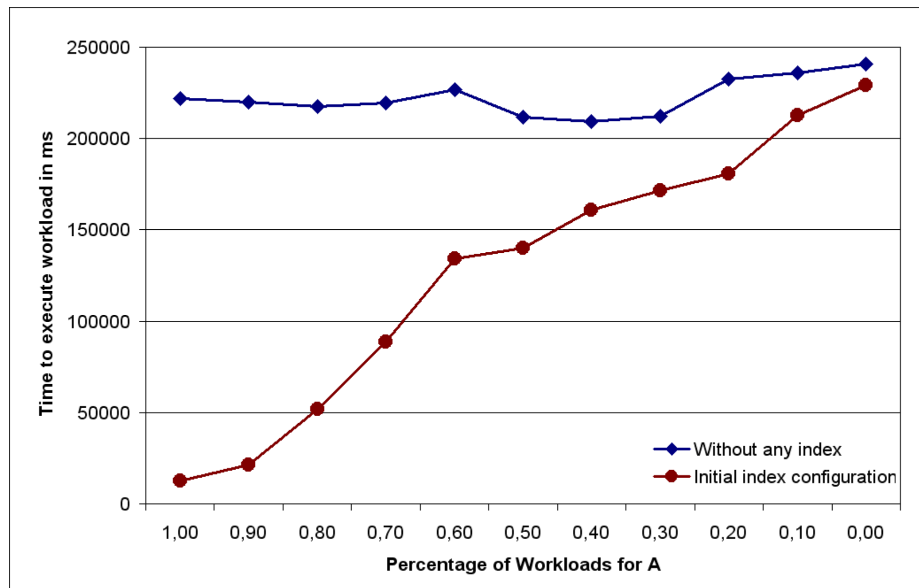


Figure 6.4: Test 1: Measurements without an index and with one initial set of indexes.

In test 1 we have applied the Index Selection Tool of KeyX to the workloads and triggered the Index System to create suitable indexes for each workload. In figure 6.5 we show the results of this test:

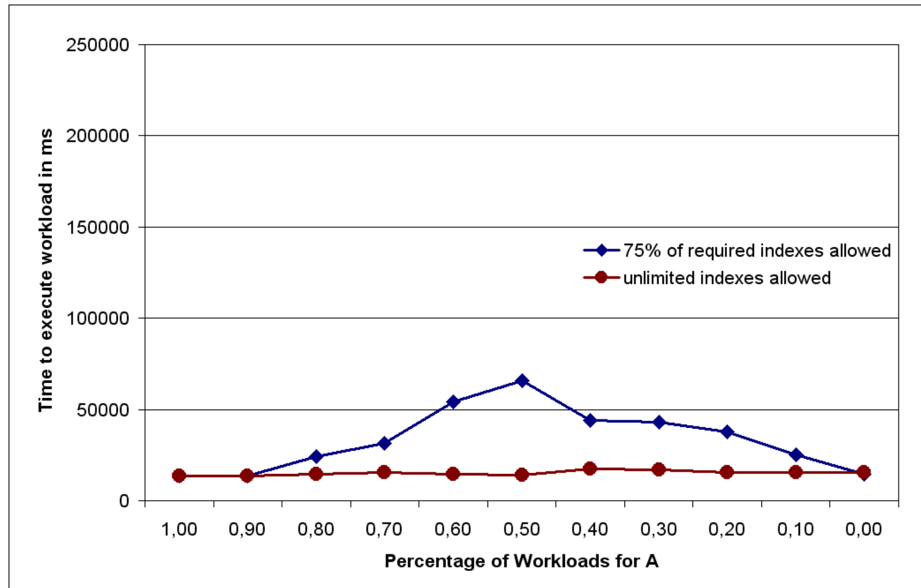


Figure 6.5: Test 1: Measurements with adaptive indexes having different space restrictions.

The first curve of figure 6.5 shows the execution time of the workloads without restricting the number of established indexes. In this case, each query from A and B has a covering index leading to a very good overall performance of the indexes for all workloads. Having no restrictions to the number of indexes is quite an unrealistic assumption. For this reason we started another run with a space constraint: Only 75% of the required indexes are allowed to be created. The results are shown in the second curve of figure 6.5. The index performs excellently for the workloads where A or B are dominant. For workloads with operations from A and B the index performs poorer but the execution time of the workloads is still significantly less than having no indexes.

Test Scenario 2

In the previous scenario we tested how our KeyX index system adapts itself when changes in the typical usage appear. In the second scenario we evaluated the consequences of a change in the ratio of querying and modifying operations of a constant workload.

An inevitable side affect of any index is that changes in the original data must be followed by the index's data structure in order to keep them consistent. This produces additional maintenance costs. Therefore, there is no profit in indexing data that is modified often.

For the test we have set up one workload of 100 database operations with 10 different path expressions. The first version of this workload consists only of querying operations; afterwards we increased the percentage of modifying operations. For this test it makes no difference whether the modification is an *insert*, *update*, or *delete* because the affected index has to be updated in each case. Figure 6.6 shows the results of this experiment:

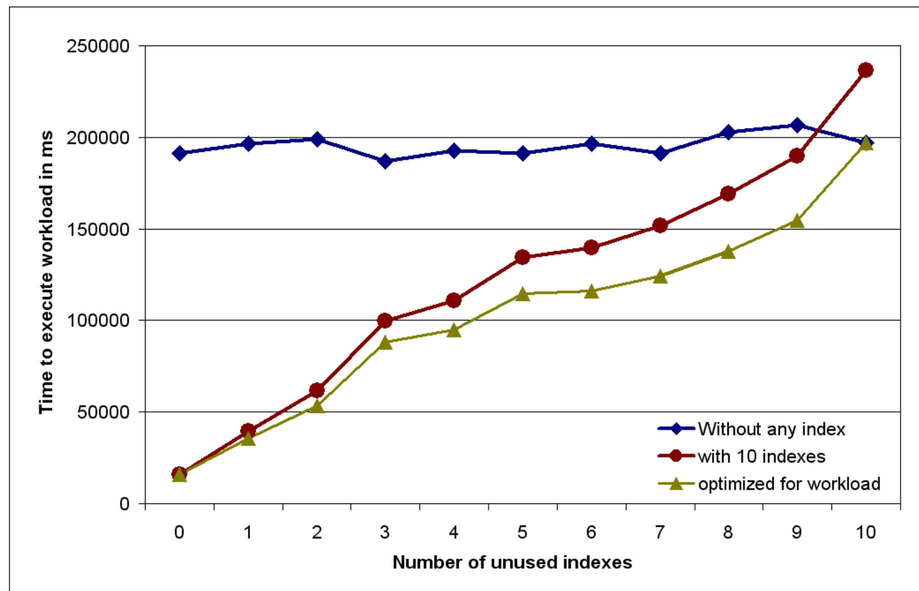


Figure 6.6: Test 2: Change of the query/update ratio of the workload's operations

Like in the previous scenario the first curve shows the execution time of the workload without any index. The initial workload consists only of queries so that the initial index configuration established all 10 indexes.

The second curve shows the performance of theses 10 active indexes if the percentage of modifying operations grows. The execution time of the workload increases because modifying operations operate on both the original data and the index. Finally, the query execution time with the 10 indexes exceeds the execution time without any index because all indexes have to be maintained.

The third curve shows that an adaptive system with less indexes performs better. The Index Selection Tool of KeyX switches more and more indexes off, if their query/update ratio is too small to gain a profit from an index. The difference between both graphs depends on the costs of updating disturbing indexes. In our test only few keys were affected by the modifying operations. For other data and operations the difference may become larger. The graphs do not grow linearly because the different indexes have a varying influence on the workload's execution time.

Test Scenario 3

The previous two scenarios were constructed to evaluate isolated characteristics of the KeyX auto index system and operated on artificial data. In order to determine the overall performance of KeyX we set up a more realistic test using real XML data from the *DBLP* project [70] - the well-known computer science bibliography. The full DBLP data consists of approximately 500,000 publications, mainly articles, inproceedings, and books.

Our concrete test data is an extract of the full DBLP of roughly 26 Megabyte and consists of 586546 element nodes, attribute nodes and text nodes, more precisely 534 articles, 57000 inproceedings and 1024 proceedings.

For the test we set up 27 different XPath based queries. Each operation o has one index candidate of class $ican_1(o)$ which supports the query to the best. We created an initial workload by randomly selecting 25 of these database operations. In general, some operations are selected multiple times and others are not part of the workload. Additionally, the operations in the workload are assigned as querying or modifying at random using a predefined ratio.

Further workloads are created by a delta algorithm that exchanges one operation from the workload with a new one that is selected randomly from the set of 27 operations. The total size of the workload stays unchanged.

The delta algorithm guarantees small and random changes in the workload - both in the contained path expressions and the ratio of querying and modifying operations. This should simulate a real database application that changes over30 time. Due to the slowly changing workload the ISP Tool is able to adapt the KeyX index system: The index selection tool is called periodically (every 30 runs) and finds a new index configuration that suits better for the changed workloads. Of course, each run of this non-deterministic algorithm generates different results. The costs to drop and create new indexes are not taken into account because in realistic scenarios with less fast changing workloads the index selection tool would be called less frequently and index updated can be done in times when the CPU is less used. We present the measurements of a representative test run in figure 6.7.

The first four workloads are executed without any index. Then, the index selection tool is called and creates indexes that accelerate ongoing workloads. The delta algorithm changes the workload more and more so that the established indexes are performing poorer. Each 30th there is seen an edge in the curve that indicates that the index selection tool has updated the index configuration. The sawtooth pattern is typical for this test.

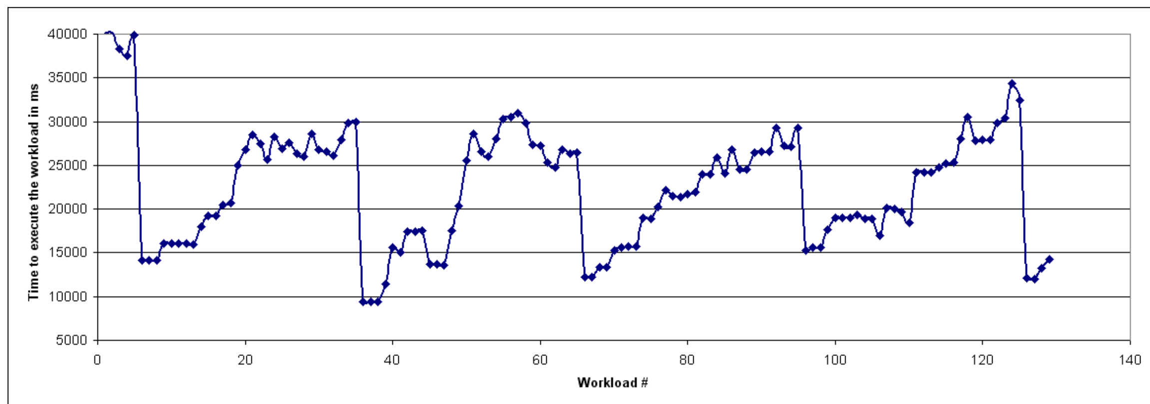


Figure 6.7: Test 3: Long-running evaluation of the ISP tool that optimizes the database to a permanently changing workload.

The pseudo code for this test scenario including the delta algorithm is presented below.

```
1  /**
2    * Pseudo code for scenario 3:
3    * This method creates, modifies and
4    * executes a workload continuously.
5    * The ISP Tool is called to optimize
6    * the workload's execution time.
7    */

9  int counter = 0;
10 int ispFrequency = 30;
11 double queryModifyRatio = 0.75;
12 int workloadSize = 25;
13 Workload w =
14   createRandomWorkload(workloadSize)

16 while (true){
17   // create new database Operation
18   Operation op = new Operation();
19   op.path = createRandomPathExpr();
20   op.type = getType(queryModifyRatio);

22   // swap op with random op from w
23   int index =
24     (int)random.nextDouble()*workloadSize;
25   w[index] = op;

27   // execute and log the workload
28   w.log();
29   w.execute();

31   // call ISP tool each ispFreq'th run
32   if (counter % ispFrequency == 0)
33     ispTool.createSuitableIndexes(w);

35   counter++;
36 }
```

Chapter 7

The XML Index Update Problem

In contrast to relational DBMS, where indexes and index structures are well-known since decades, indexes in XDBMS are still an active field of research with no standards established yet. A lot of approaches have been introduced in recent years dealing with indexes for querying XML data. The problem of updating an index is a minor focus of most publications. If at all, the authors describe how their data structure can be updated from a technical point of view. To the best of our knowledge, the problem whether an XML index i is affected by a modifying operation o has never been faced before. We call this problem the *XML Index Update Problem (XIUP)*.

In this chapter we give an algorithm that is based on finite automata theory and decides whether an XPath-based database operation affects an index that is defined universally upon *keys*, *qualifiers*, and *a return value* of an XPath expression. Although we focus on the KeyX indexing approach the general idea and the algorithms are transferable to other selective indexing approaches for semistructured data. If an approach is non-selective (it covers all data) it is affected by any modification.

We present an efficient intersection algorithm for the XPath fragment $XP^{\{[],*,//\}}$ containing path expressions without the NOT operator. The algorithm is based on finite automata. For the XPath fragment $XP^{\{[],*,//,NOT\}}$ containing path expressions with the NOT operator the intersection problem becomes NP-complete leading to exponential computations in general. With an average case simulation we show that the NP-completeness is no significant limitation for most real-world database operations.
In addition, we provide algorithms for updating our KeyX indexes efficiently if they are affected by a modification. The *Index Update Problem* is relevant for all applications that use a secondary XML data representation (e.g. indexes, caches, XML replication/synchronization services) where updates must be identified and realized.

7.1 Introduction

We assume that an index is *selective*; i.e. it is defined to accelerate a specific query and not all queries in general. Index approaches that are not selective (e.g. structural summaries like the Strong DataGuides) reflect the whole XML data and are affected by every modifying operation leading to an update of the index structure. Therefore, the Index Update Problem for non-selective indexes is trivial. More about the characteristics of selective and non-selective index approaches can be found in chapter 4.

We motivate the XML Index Update Problem (XIUP) by two examples that operate on data from the DBLP project :

Example 19 The index i_1 is defined to accelerate XPath expressions of the shape of query q_1:

```
1  q1 = /dblp/book[author='x']
```

Index i_1 indexes all `book` elements by the value of their `author` child which is interpreted as a *key* for this query. In our index approach *KeyX* all keys are stored in a search tree offering logarithmic retrieval time. Thus, if an author's name is given, we find the corresponding books efficiently.
The XUpdate operation o_1 deletes all books that are written by the author Kempa.

```
1  o1 = <xupdate:remove
2    select=/dblp/book[author='Kempa'] >
3       </xupdate:remove>
```

Obviously one can see that the index i_1 is affected by o_1 because after executing o_1 there is no book author 'Kempa' anymore in the data. The key 'Kempa' has to be removed from the index to keep it consistent. □

At first glance, it seems easy to determine the affection by comparing the contained XPath expressions which are equal in this example. But because XPath expressions may contain more complex navigational steps the decision can become more difficult; this is shown in the following example.

Example 20 Index i_2 indexes all child elements of the `dblp` element which have a `title` child that is used as a key.

```
1  q2 = /dblp/*[title='x']
```

The modifying operation o_2 deletes all children of all `article` elements.

```
1  o2 = <xupdate:remove
2           select=/dblp/article/*/>
3       </xupdate:remove>
```

First, one can remark that the contained XPath expressions are not equal. Second, without any schema information like a DTD or XML Schema we do not know if the `dblp` element is allowed to have an `article` element and that the `article` element may have a child named `title`. Due to the wildcard operator (*) it is not sufficient to perform a string comparison of both XPath fragments. With one or more descendant axis (`//`) this problem becomes even more complex.

7.2 Intersection of Two Path Expressions

In this section we introduce the *XML Index Update Problem* formally and reduce it to the *Intersection Problem* of two XPath expressions.
An index i *covers* a query q if the nodes returned by the index structure are the same as the nodes returned by the database itself. Because the index is defined upon a return value, a set of keys, and a set of qualifiers, the index may be outdated if one of these nodes has changed. The key of an index is a structural and a content property. Therefore, we have to update the index if a key appears, disappears, or its value changes. Qualifiers and the return values are only structural properties - their values are not reflected by the index. Therefore, the modification of these values does not affect the index. For instance, if we index the title of books by their autors the index would consist of the values of authors and references to title elements. The current value of a title does not influence the index and can therefore be ignored.

As described in chapter 3, T denotes the set of all XML data; any well-formed XML document is in T. The evaluation of a path expression p upon an XML document $t \in T$ is denoted by $p(t)$ and returns the nodes of t that are selected by p. We denote *Mod*$(p) \subset T$ the set of all XML data where p returns a non-empty result set.
Basically, the index that is defined upon p is affected by a modifying operation p' if the return sets of the evaluation of p and p' share at least one mutual node in an XML data t. Formally, this means that the intersection of p and p' is not empty:

$$p(t) \cap p'(t) \neq \emptyset$$

We call this problem the *XPath Intersection Problem* (XIP). It is relevant for all applications that use a secondary XML data representation (e.g. indexes, caches, XML replication/synchronization services) where updates must be identified and realized.

The work of *Miklau* and *Suciu* [81, 82] provide an algorithm for the *containment* (subset) relation of XPath expressions detecting if

$$p(t) \subseteq p'(t).$$

If the containment holds for both directions then p and p' are semantically equivalent ($p \equiv p'$). In theory, the algorithm for containment could be applied to calculate the existence of an intersection because it holds that

$$p(t) \cap p'(t) = \emptyset \Leftrightarrow p(t) \subseteq p'(t)^c.$$

with $p'(t)^c$ denoting the complement of $p'(t)$. The complement $p'(t)^c$ is a path expression that selects all nodes that are not selected by $p'(t)$ for all $t \in T$.

For three reasons the algorithm from Miklau and Suciu cannot be applied:

- First, the complement of an XPath expression cannot be expressed in general using only one path expression. One possibility is to enumerate all path expressions that build the complement. But because the alphabet Σ of XML element names is infinite in the XPath fragment of the algorithms this enumeration has no end.
- Second, for a multitude of path expressions the path expressions that build the complement may contain the `NOT` operator that is not supported by the XPath fragment of the containment algorithm.
- Third, the containment problem is co-NP complete leading to an exponential runtime of the deterministic implementation of the algorithm. The intersection problem for the same XPath fragment has polynomial complexity ($O(n^2)$) and our algorithm may even have nearly linear complexity for typical path expressions.

7.2.1 Formalization

Having a concrete XML data t the intersection problem can be computed easily by evaluating both p and p' to the sets $p(t)$ and $p'(t)$. Afterwards, we can calculate the intersection by comparing all nodes of both sets. This can be done in $O(n \cdot \log(n))$ complexity [1]. In the context of databases n is the number of elements that are selected by the path expressions. For a real application n is usually too big, leading to prohibitive long computations: for a database storing thousands of books (e.g. amazon.com) we cannot check whether each book is affected by a modifying operation or not.

Second, in some cases we do not have a concrete XML data t so that the evaluation of the path expressions is not possible. For both reasons the decision whether the intersection is empty or not has to be made exclusively upon the path expressions.

Our approach works without any schema like DTD or XML Schema. This is an important demand for XDBMS. In schemaless XML data element types may appear or disappear during the lifetime of the database.

The *XPath Intersection Problem* for schemaless XML data is defined as follows:

[1] We need $2 \cdot O(n \cdot \log(n))$ steps to order the nodes of both result sets into ordered lists. With a linear search ($O(n)$) we can determine the elements that are contained in both lists.

Definition 27 (XPath Intersection Problem)
Is there an XML data where p and p' select at least one mutual node? Or formally, is the following logic formula satisfiable:

$$\exists\, t \in T : p(t) \cap p'(t) \neq \emptyset$$

with $p, p' \in P_{l_{rel}} \cup XP^{\{[],*,//\}}_{rel} \cup XP^{\{[],*,//,NOT\}}_{rel}$[1]. □

We abbreviate this formula by $p \cap p' \neq \emptyset$.

Next, we want to show that the keys and qualifiers of an expression in $XP^{\{[],*,//\}}$ do not influence the intersection. We use the function $linearize : XP^{\{[],*,//\}}_{abs} \rightarrow P_{l_{abs}}$ as defined in definition 9.

Theorem 1 *For $p, p' \in XP^{\{[],*,//\}}$ it holds that:*

$$p \cap p' \neq \emptyset \;\Leftrightarrow\; linearize(p) \cap linearize(p') \neq \emptyset.$$

The theorem states that two path expressions from $XP^{\{[],*,//\}}$ have a non-empty intersection if and only if their linearized path expressions have a non-empty intersection.

PROOF (by contradiction):
1) Let us assume that $linearize(p(t)) \cap linearize(p'(t)) \neq \emptyset$ and $p(t) \cap p'(t) = \emptyset$.
We illustrate the idea of this proof by a simple example in figure 7.1. Because the linearized expressions have an intersection there is at least one node $\tilde{n}$ that is selected in an XML data $\tilde{t}$. From the root node of $\tilde{t}$ there is a path to $\tilde{n}$ consisting of an arbitrary sequence of nodes $root, n_1 ... n_r, \tilde{n}$. The linearized path expressions do not request any conditions to these nodes while p and p' do.

Because p and p' contain no negations (NOT-operator not allowed in this fragment) there are two XML data $t \in Mod(p)$ and $t' \in Mod(p')$ with p selecting at least one node n in t. The node n' of t' is selected analogously by p'. The nodes n and n' have the same path from the root because the linearized path expression select both $\tilde{n}$.
In t and t' all nodes of the paths $root, n_1 ... n_r, n$ and $root, n_1 ... n_r, n'$ fulfill the conditions that are expressed in p respectively p'

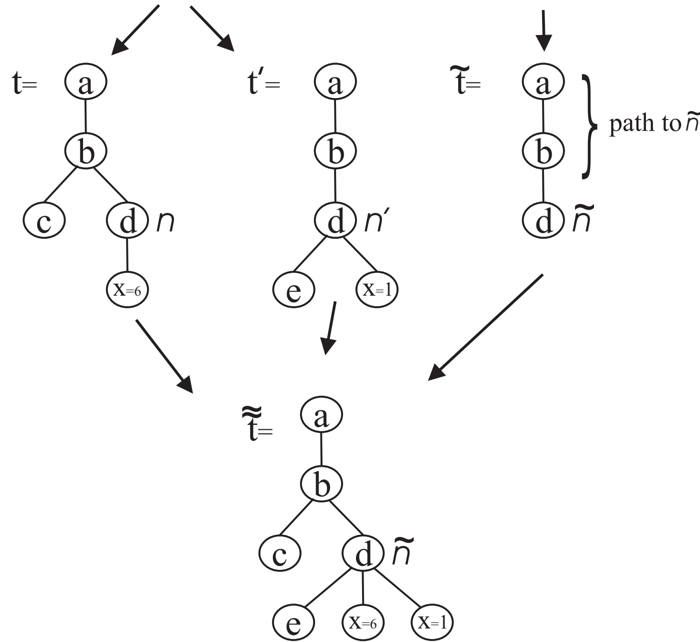
Now we construct a data $\tilde{\tilde{t}}$ that extends $\tilde{t}$ by unifying all nodes of t, t' and $\tilde{t}$. This way,$\tilde{\tilde{t}}$ fulfills the qualifiers and key conditions of both p and p'.
But then $p(t) \cap p'(t)$ cannot be empty because both path expressions select $\tilde{n}$; this is a contradiction to the assumption.

[1]The union of the three fragments $P_l, XP^{\{[],*,//\}}$ and $XP^{\{[],*,//,NOT\}}$ is $XP^{\{[],*,//,NOT\}}$. We list the three fragments separately to show that p and p' can be of any of them.

Example 21 :

$$p = /a/b[c]//d[x > 5] \quad \rightarrow \; linearize(p) = /a/b//d$$
$$p' = /a/b/d[e \text{ and } x < 4] \; \rightarrow \; linearize(p') = /a/b/d$$

□



■

Figure 7.1: Example to illustrate the idea of the proof

2) Conversely, let us assume that $linearize(p(t)) \cap linearize(p'(t)) = \emptyset$ and $p(t) \cap p'(t) \neq \emptyset$. Because qualifiers and key conditions act both as filters cutting down the result sets of p and p' they can never generate an intersection if the unfiltered sets have no intersection. ↯

The theorem holds because we can unify the nodes that fulfill the qualifiers. For instance, let $p = /a/b[x < 2]$ and $p' = /a/b[x > 2]$. If the selected b node has two x nodes with the values 1 and 3 then both p and p' select b.
This is not possible if key comparisons touch the selected node; for instance p=$/a[. > 5]$ and $p' = /a[. < 5]$ have no intersection because the value of a cannot be larger and smaller than 5 simultaneously. First, by the definition of $XP^{\{[],*,//\}}$ those path expressions cannot be expressed and second, they are still decidable if we check their satisfiability as described in 7.3. The same holds for attributes which are not part of the XPath fragments of this work.

As we have shown the XIP for $XP^{\{[],*,//\}}$ is equivalent to P_l. Therefore, it suffices to define an algorithm deciding the emptiness of the intersection of two expressions from P_l.

7.2.2 Automaton for $Mod(p)$

The general idea of the intersection algorithm is to build two finite automata A and A' with A accepting $Mod(p)$ and A' accepting $Mod(p')$ with $p, p' \in P_{l_{abs}}$ two absolute linear path expressions.
Having A and A' we build the product automaton B . The emptiness of the intersection of p and p' is a property of B.

Unfortunately, finite automata are defined on the basis of a finite alphabet. In contrast, path expressions operate on XML data with an infinite alphabet because the node labels are not limited.

In section 3.2.1 we showed that a linear path expression $p \in P_{l_{abs}}$ can be transformed into a regular expression $r \in REG_{\Sigma,\alpha}$ with $\Sigma = \Sigma(p)$ and $\alpha \notin \Sigma$ an arbitrary new symbol.

Lemma 3 *For any regular expression $r \in REG_{\Sigma,\alpha}$, respectively its language L_r, one can construct a finite automaton A that decides whether an input string is a word of L_r or not.* □

The proof of the lemma is omitted here because it is basic knowledge in theory of finite automata. The proof can be found, for instance, in [52].

When reading an XML data as input for a finite automaton we have the following problem: The automaton expects a string of several symbols in a defined order. In tree-like XML data a node may have several children so that the next symbol (element label) is not defined unambigiously.

Therefore, we define a function $path_{leaf} : T \rightarrow \mathcal{P}(string)$ that extracts all paths (sequences of nodes) from the root node to each leaf element node in the XML data. Text nodes are ignored as they are not affected by linear path expressions. The paths are returned as strings built from the labels of the contained nodes. The function is defined as follows:

Definition 28 (Function *path*$_{leaf}$)

$$path_{leaf}(t) = path(t.root)$$

$$path(n) = \begin{cases} n.label & : n.children = \emptyset \\ n.label + \text{";"} + \{path(c) | c \in n.children\} & : \text{otherwise} \end{cases}$$

with $t \in T$ and $n \in N$; + denotes the concatenation of strings with $a + \text{";"}\{b, c, d\} = \{a; b,\ a; c,\ a; d\}$. The semicolon is a delimiter used to distinguish different element label (e.g. $a; b \neq ab$). □

Example 22 The XML data t_1 with $t_1 =$

```
1  <a>
2    <b>
3      text
4    </b>
5    <c>
6      <d/>
7    <c>
8  </a>
```

has the following paths to leaf element nodes: $path_{leaf}(t_1) = \{a;b,\ a;c;d\}$. □

Lemma 4 *$n.label$ with $n \in t.nodes$ appears at least once in a path $\in$ path$_{leaf}()$.* □

PROOF (by contradiction)
We assume that n is a node of t not appearing in any path in t. Because $path_{leaf}(t)$ contains the paths to leaf nodes, n cannot be a leaf node. Therefore, n must have at least one child node c. c or one descendant of c is a leaf node because t is finite and a tree without circles. This leaf is called l. Because each node in t has exactly one parent, n is in the path from l to $t.root$ and reverse from $t.root$ to l. ↯

Now we are able to extract all paths from the root node to leaf nodes as strings of symbols $\in \Sigma_t = \{n.label | n = t.nodes\}$. But because Σ_t may contain symbols that are not in $\Sigma(p)$ the automaton that is defined by the regular expression $REG_{\Sigma,\alpha}$ has no transitions for symbols $s \in \Sigma_t \backslash \Sigma(p)$.

With a further function *rename* : $\mathcal{P}(string) \rightarrow \mathcal{P}(string)$ we change all symbols $s \in \Sigma_t \backslash \Sigma(p)$ of the strings in $path_{leaf}$ to α.

Definition 29 (Renaming function)

$$
\begin{aligned}
rename(S, \Sigma, \alpha) &= \{rename(s) | s \in S\} \\
rename(s) &= rename_s(s_1) + rename_s(s_2) + ... + rename_s(s_n) \\
rename_s(s) &= \begin{cases} s & : \ s \in \Sigma \\ \alpha & : \ otherwise \end{cases}
\end{aligned}
$$
□

with S a set of strings, s one string consisting of the sequence of n symbols $s_1, s_2 ... s_n$.

Technically, the function *rename* is a homomorphism that substitutes a particular string for each symbol.

The resulting strings have an alphabet restricted to $\Sigma(p) \cup \{\alpha\}$ and can be processed by a finite automaton. Next, we show that an XML data t is in $Mod(p)$ if and only if at least one path of t is a word of L_r. Formally this means:

Theorem 2

$$t \in Mod(p) \;\Leftrightarrow\; \exists\, t_p \in \textit{rename}(\textit{path}_{\textit{leaf}}(t),\ \Sigma(p),\ \alpha)\ \textit{with}\ t_p \in L_r$$

PROOF If there is a $t \in$ *Mod*(p) then there is a node $n \in t.nodes$ with $n \in p(t)$. Because of lemma 4 there is a path $t_p \in$ *path*$_{leaf}(t)$ containing n. As a selected node, n fulfills the conditions expressed by the path expression p; the equivalent regular expression will therefore match t_p. Conversely, if $t \notin$ *Mod*(p) there is no node in $t.nodes$ that is selected by p. Therefore, the regular path expression defined by p matches no String in *path*$_{leaf}(t)$. ■

We have shown that the question whether a linear path expression p selects a node in an XML data t is solvable with the help of formal languages and automata theory. In principle, this approach can be used to evaluate path expressions (see also section 2.3.3). But because the numbers of paths *path*$_{leaf}(t)$ grows linearly with the number of nodes this approach is not very efficient. Surprisingly, we will never have to process any XML input string to decide the intersection of two path expressions!
In the next part we show how a finite automaton that accepts L_r is built:

A finite automaton is defined by its *states* and *transitions*. The *transitive closure* of a state s is the set consisting of all states that are reachable from s by using the transitions recursively. See [52] for details about finite automata theory.

Definition 30 (Automaton accepting *Mod*(p))
We build a finite automaton A accepting *Mod*(p) as follows:
A is a tuple $(Q,\Sigma,\ \sigma,\ q_0,\ F)$ with

- Q = NODES(p) a set of *states*,
- $\Sigma = \Sigma(p) \cup \Sigma(p')$ a finite alphabet consisting of all node labels of p and p'.
- σ is a function $Q\ \times \Sigma\ \rightarrow\ Q$ defining the set of *transitions*.
- q_0 is the *initial state* whereas
- F is the set of *final states*.

For each node $n \in$ *nodes*(p) we create a corresponding state q_n. A further state q_0 is used as starting point for A.

1. $\forall x \in$ *nodes*(p) : $x.children = y\ \wedge\ x.label \neq *$, A has a transition (q_y; $x.label$) $\rightarrow q_x$
2. Otherwise, $\forall x \in$ *nodes*(p) : $x.children = y\ \wedge\ x.label = *$, A has $|\Sigma(p) \cup \alpha|$ transitions (q_y; s) $\rightarrow q_x$ for all $s \in\ \Sigma(p) \cup \alpha$.

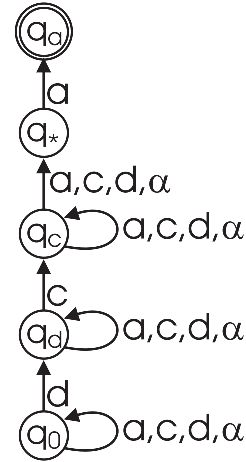
3. $\forall x \in$ *nodes*$(p) : x.descendant = y$, A has $|\Sigma(p) \cup \alpha|$ transitions $(q_y;\ s) \rightarrow\ q_y$ for all $s \in\ \Sigma(p) \cup \alpha$. These transitions build loops on q_y

4. A has one transition $(q_0,\ e.label) \rightarrow\ q_e$ with q_0 the bottommost state that has no incoming transition.

These transitions lead to a nondeterministic finite automaton. The terminal state is $q_{p.root}$. The initial state is q_0. □

Example 23 The linear path expression $p = /a/*//c//d$ leads to an automaton A with $Q = \{q_a, q_*, q_c, q_d, q_0\}$ and $\sigma = \{t_0 \ldots\ t_{14}\ \}$. The alphabet is $\Sigma(p) = \{a, c, d\}$ The following transitions are built:

$$
\begin{aligned}
t_0 &= (q_0; "d") \rightarrow q_d, \\
t_1 &= (q_d; "a") \rightarrow q_d, \\
t_2 &= (q_d; "c") \rightarrow q_d, \\
t_3 &= (q_d; "d") \rightarrow q_d, \\
t_4 &= (q_d; "\alpha") \rightarrow q_d, \\
t_5 &= (q_d; "c") \rightarrow q_c, \\
t_6 &= (q_c; "a") \rightarrow q_c, \\
t_7 &= (q_c; "c") \rightarrow q_c, \\
t_8 &= (q_c; "d") \rightarrow q_c, \\
t_9 &= (q_c; "\alpha") \rightarrow q_c, \\
t_{10} &= (q_c; "a") \rightarrow q_*, \\
t_{11} &= (q_c; "c") \rightarrow q_*, \\
t_{12} &= (q_c; "d") \rightarrow q_*, \\
t_{13} &= (q_c; "\alpha") \rightarrow q_*, \\
t_{14} &= (q_*; "a") \rightarrow q_a, \\
t_{15} &= (q_0; "a") \rightarrow q_0, \\
t_{16} &= (q_0; "c") \rightarrow q_0, \\
t_{17} &= (q_0; "d") \rightarrow q_0, \\
t_{18} &= (q_0; "\alpha") \rightarrow q_0.
\end{aligned}
$$



□

Figure 7.2: Automaton accepting *Mod*(p) with $p = /a/*//c//d$

For an XML data t A processes all strings of the set $rename($*path*$_{leaf}(t))$ bottom-up, i.e. from a leaf node to the root. A visual representation of A is on the right side of figure 7.2. The initial state is q_0; q_a is the one final state. Please note that the transitions are not deterministic (e.g. t_2, t_5). Analogously to tree-automata the finite automaton processes the XML data bottom-up. Basically, it is possible to build an equivalent automaton that processes the input top-down.

Now we are able to build two automata A and A' that accept $Mod(p)$ and $Mod(p')$ independently. A standard product automaton B of A and A' accepts if and only if both A and A' accept. The reader is referred to [52] on the definition of a product automaton. Informally, one can say that B simulates the simultaneous execution of A and A'. Based on the product automaton the following algorithm checks the emptiness of the intersection of p and p':

1. $p = linearize(p)$;
2. $p' = linearize(p')$;
3. create automaton A accepting **Mod**(p);
4. create automaton A' accepting **Mod**(p');
5. create product autom. B accepting **Mod**$(p) \cap$ **Mod**(p');
6. $< q_{0_A} \times q_{0_{A'}} >$ = initial state of B;
7. $< q_{F_A} \times q_{F_{A'}} >$ = final state of B;
8. $CLOSURE$ = transitive closure of $< q_{0_A} \times q_{0_{A'}} >$
9. if ($< q_{F_A} \times q_{F_{A'}} > \in CLOSURE$) return $true$
   else return $false$;

Figure 7.3: Pseudo code of the algorithm that detects the intersection

In the first two steps, we linearize the path expressions because eventually existing qualifiers do not affect an intersection. The algorithm creates an automaton A accepting all XML data where p returns non-empty result sets in step 3. Analogously, step 4 creates an automaton A' for path p'. In step 5 the product automaton B of A and A' is created. B accepts all XML data where both path expressions p and p' evaluate to a non-empty result set. In the last steps of the algorithm it is checked if there is a path from the initial state to a final state of B by calculating the transitive closure of the initial state of B. If this path exists there is an XML data t that is both accepted by A and A'; this implies that the intersection is not empty for this t.
Please note that the emptiness of the intersection is a property of the product automaton. It is determined without processing any concrete XML input.

When we build a product automaton the complexity of the algorithm becomes quadratic ($O(|STATES_A| \cdot |STATES_B|)$). This is because the states of B are built by combining all states of A and B. An improved algorithm avoids the construction of the full product automaton by a lazy evaluation of reachable states in A and A'. This optimized algorithm can be found in [52] and performs better in general (see section 7.5) but has a quadratic worst case complexity, too.

7.3 NP-Completeness for Path Expressions with NOT

The XPath Intersection Problem for the XPath fragment $XP^{\{[],*,//,NOT\}}$ including the `NOT`-operator is more complex. We call this special case of the problem *XIP*^{NOT}. With the `NOT`-operator it is possible to express contradictory XPath expressions

like

$$p_3 = /a[b]; \quad p_4 = /a[\mathit{NOT}(b)].$$

Expression p_3 selects all a nodes that have a b child while p_4 selects a nodes that have no b child. Although both expressions select a nodes they are mutually exclusive. This implies that p_3 and p_4 can never share one same node.

Theorem 3 *The XPath Intersection Problem for the fragment XIP^{NOT} is NP-complete.* □

PROOF First, we show that *XIP^{NOT}* is NP-hard, i.e. every NP-complete problem can be reduced on *XIP^{NOT}* in polynomial time. We show the NP-hardness by reducing 3-SATISFIABILITY (3SAT) [3, 21]. It was proven in 1971, by Cook, that 3SAT is NP-complete.

Definition 31 3SAT is defined as follows: Given a Boolean formula $F = F_1 \wedge ... \wedge F_k$, $F_j = z_{j,1} \vee z_{j,2} \vee z_{j,3}$ and the literals $z_{i,j} \in \{x_1, \overline{x_1}, x_2, \overline{x_2}, ..., x_r, \overline{x_r},\}$.

Question: Is there an assignment of $true/false$ values to the r variables $x_1, ..., x_r$ so that F can be satisfied? □

F is in 3-conjunctive normal form (3-CNF) because each clause has exactly three distinct literals. The basic idea of the reduction is to create two formulas F_A and F_B that are always satisfiable independently. By the construction of F_A and F_B we guarantee that they are only satisfiable simultaneously if and only if F is satisfiable. Having the two formulas we create two expressions $p, p' \in XP^{\{[],*,//,NOT\}}$ that have an intersection if and only if F is satisfiable. The reduction is done in polynomial time. We build F_A and F_B as follows:

1) We take r new variables $a_1, a_2, ...a_r$. F_A is structured like F with the difference that all negated x_is are replaced by a corresponding negated a_i variable. Formally, this means that $F_A = \tilde{F_1} \wedge ... \wedge \tilde{F_k}$ with $\tilde{F_j} = \tilde{z_{j,1}} \vee \tilde{z_{j,2}} \vee \tilde{z_{j,3}}$ and

$$\tilde{z_{i,j}} = \begin{cases} x_l & \text{if} \quad z_{i,j} = x_l \text{ or} \\ \overline{a_l} & \text{if} \quad z_{i,j} = \overline{x_l} \end{cases}$$

This way, F_A is trivially satisfiable by assigning $true$ to all remaining (non-negated) x-variables and $false$ to all (negated) a-variables.

2) The second formula F_B is created as follows:

$$F_B = \Big((x_1 \wedge a_1) \vee (\overline{x_1} \wedge \overline{a_1})\Big) \wedge ... \wedge \Big((x_r \wedge a_r) \vee (\overline{x_r} \wedge \overline{a_r})\Big)$$

F_B is satisfiable if and only if every $x_i = a_i$ independent of whether they are both $true$ or both $false$.

Now we build two path expressions p and p' corresponding to F_A and F_B as follows: Both p and p' have the same prefix, for instance $/a$. The formulas are attached as a qualifier to the a node. The logical operators $\wedge$ and $\vee$ are translated to AND and OR operators from XPath. Each positive variable of F_A respectively F_B is interpreted as the existence of a matching child element. Each negative variable is covered by a NOT operator.

Because F_A and F_B are always satisfiable individually *Mod*(p) and *Mod*(p') are not empty.
If $p(t) \cap p'(t) \neq \emptyset$ it means that there is an a node that fulfills the two qualifiers of p and p'. But this means that F_A and F_B are satisfied simultaneously. Hereby, the reduction $3SAT \leq_{pol} XIP$ is finished; the conversion from a Boolean formula to a path expression can be done in polynomial (even linear) time.

Finally, *XIP*^{NOT} is in NP because a nondeterministic Turing-machine may guess a satisfying assignment if it exists. Because the *XIP*^{NOT} is NP-hard and belongs to NP it is NP-complete. ■

7.3.1 Algorithm and Complexity of *XIP*^{NOT}

Our algorithm solving XIP^{NOT} for two path expressions $p, p' \in XP^{\{[],*,//,NOT\}}$is based on the introduced algorithm checking the intersection of linearized path expressions (see figure 7.3). If the linearized path expressions of p and p' have an intersection we now have to check if their additional qualifiers and keys are mutually exclusive. This implies that we have to check the satisfiability of $p \ \wedge \ p'$. Because of the NP-completeness of XIP^{NOT} this step has an exponential worst case complexity: every permutation of $true/false$ that is assigned to the variables in the qualifiers has to be generated and checked if it satisfies $p \ \wedge \ p'$. If we find no satisfying configuration the intersection remains empty.

The algorithm uses a simple optimization step that reduces the number of variables that have to be checked: A variable may only be responsible for mutual exclusiveness if it appears negated and non-negated in the set of qualifiers of p and p'. Therefore, we can remove all variables from the expressions that are only negated or non-negated. This simple optimization has a similar aim as the *Davis Putnam* algorithm [23] for deciding 3SAT. The extended algorithm is presented in figure 7.4:
In the first step the algorithm checks whether the linearized path expressions have an intersection. If this is not the case the algorithm stops and returns $false$. In lines 2 to 4 we reduce the numbers of relevant variables by removing the trivial ones. The function *getNegVars* returns all variables from a path expression that appear negated. The function *getPosVars* is defined vice versa. In step 5 the algorithm iterates over all possible $true/false$ values for the remaining variables

```
1.  if (!intersection(linearize(p), linearize(p)′)) return false;
2.  NEG = getNegVars(p) ∪ getNegVars(p′);
3.  POS = getPosVars(p) ∪ getPosVars(p′);
4.  NonTrivialV = NEG ∩ POS;
5.  forEach permutation of vars ∈ NonTrivialV
        if (is satisfiable(p ∧ p′,vars)) return true;
6.  return false;
```

Figure 7.4: Pseudo code of the intersection algorithm for XIP^{NOT}

and checks their satisfiability. If we find no satisfying configuration the algorithm returns $false$ because the expressions are mutually exclusive.

The optimization steps lead to a substantial acceleration as we show in the next section. Anyhow, the algorithm still has an exponential complexity leading to exhaustive computations in general. In the context of real world's XPath based database operations with a limited number of variables and qualifiers the satisfiability of path expressions can be decided in acceptable time: With an average case simulation (see next section) we determined an expected value of 155 milliseconds for testing the satisfiability of XPath expressions with 100 qualifiers and up to 150 variables.

7.4 Evaluation

In this section we present performance measurements for the introduced algorithms that decide the intersection and satisfiability of two path expressions. The implementations were done in Java; the tests were performed on a Pentium 4 with 2.66 GHz and 1 GB main memory.

7.4.1 Evaluation of Intersection

The first evaluation tests the intersection algorithm of figure 7.3 for path expressions $p, p' \in XP^{\{[],*,//\}}$. We increased the length of the path expressions p and p' from 1 to 20; the location steps of the expressions were created randomly from a finite alphabet Σ. The time measurements include the parsing of the path expressions, the creation of the three automata and the analysis whether the final state is reachable. In order to obtain stable values we executed the algorithm some thousand times with path expressions of the same length. The measured total time is divided afterwards to get a stable average execution time for one run. As the diagram in figure 7.5 shows the execution time of the optimized algorithm increases almost linearly.
The optimized algorithm that avoids the construction of the full product automaton has an average runtime of less than 1 millisecond for the largest path expression.

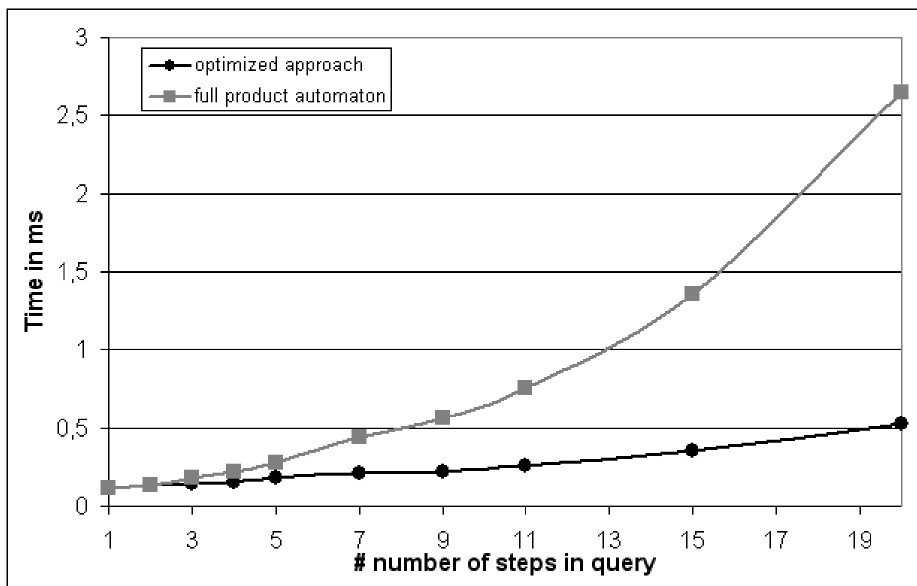


Figure 7.5: Measurements of the intersection algorithm

7.4.2 Evaluation of Satisfiability

In a second scenario we evaluated the more complex algorithm checking the satisfiability of two path expressions $p, p' \in XP^{\{[],*,//,NOT\}}$. The algorithm has an exponential runtime because all variables must be checked in order to satisfy the qualifiers. In order to show that the exponential complexity is no significant limitation in the database context we determine the expected value for the times expenses for typical operations with an average case simulation.
In general, the expected value $\mathbb{E}X$ of a random variable $X : \Omega \rightarrow \mathbb{R}$ is

$$\mathbb{E}X = \int_{\omega \in \Omega} X(\omega) \cdot p(\omega)\ d\omega$$

with $\Omega = XP^{\{[],*,//,NOT\}}$ the set of all path expressions, $X(\omega)$ is the runtime of the algorithm for a specific path expression $\omega \in \Omega$ and $p(\omega)$ its probability.

There are three variables for a path expression: the length l_p of the path $linearize(p)$ which affects the intersection algorithm. Because intersection is determined very quickly (see previous experiment) l_p has only a little effect. The second and third variables are the number of qualifiers and element names in a path expression. The element names correspond to the variables in 3SAT. Because Ω is infinite in general we have to restrict it to a reasonable boundary: We think that 100 qualifiers in a path expression provide a realistic upper value for database operations (comparable to a SELECT statement with 100 WHERE clauses in SQL). Second, because we have no distribution function for the probabilities of path expressions we assume an equal distribution.

With these assumptions and the fact that Ω is discrete the expected value $\mathbb{E}X$ can be approximated by the average value:

$$\mathbb{E}X \approx \frac{1}{|\Omega|} \sum_{\omega \in \Omega} X(\omega)$$

In the experiment we iterated the number of variables in the qualifiers up to $n = |\Omega| = 150$. With a probability of 40% a variable is used in negated form in a qualifier. This means that nearly half of the variables are negated - this is close to a worst case scenario.

Having very few variables means that only few variables have to be permuted. If we increase the number of variables the execution time increases exponentially up to a certain point where the optimization begins to take effect: for a bigger number of variables in p and p' it becomes more and more likely that some of them are only used non-negated or negated. These variables are not regarded when checking the satisfiability.
These measurements are near to some work evaluating SAT problems (e.g. [84]); the major difference is that we use Boolean formulas in XPath syntax. We present the measurements in figure 7.6.

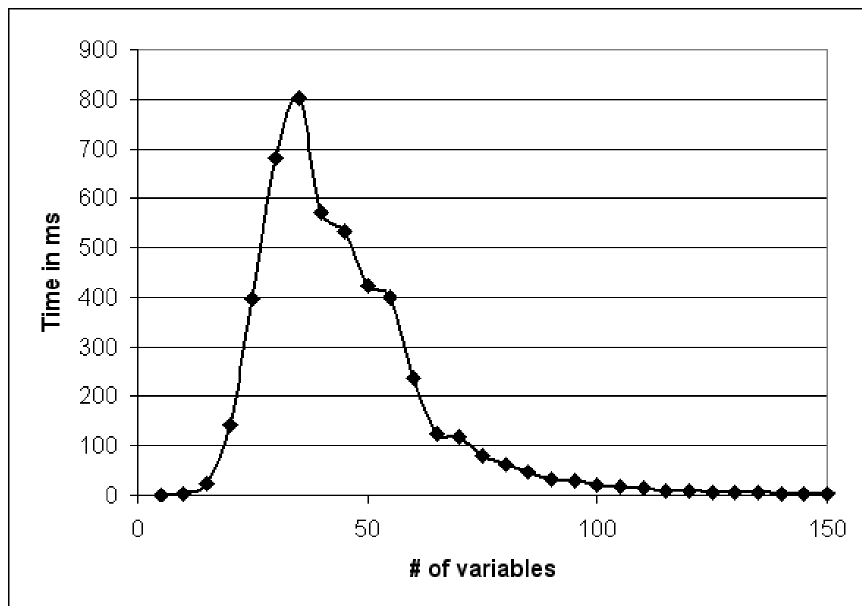


Figure 7.6: Measuring the satisfiability for a path expression with 100 qualifiers

For this experiment we measured an expected value for the duration of the algorithm of 155 milliseconds - this is the time that is spent on average to determine whether two path expressions are satisfiable. We think that this is still acceptable for most database applications - especially if the complexity of typical database operations does not come close to 100 qualifiers.

7.4.3 Evaluation of Satisfiability II

In the previous experiment we changed the numbers of variables to 100 qualifiers. In the next experiment we increased the number of variables *and* the number of qualifiers so that the path expressions become more and more complex. The variables and qualifiers have a ratio of 1 : 2 - this is the worst case as you can see in figure 7.6. For this ratio the optimization has only a restricted influence. Because the XIP^{NOT} is NP-complete the deterministic algorithm has an exponential complexity leading to more exhaustive computations of up to 420 milliseconds. But also in this scenario the expected value is significantly less: For expressions with up to 100 qualifiers and 50 variables we have an expected value of 36 milliseconds. The measurements of this experiment are shown in figure 7.7.

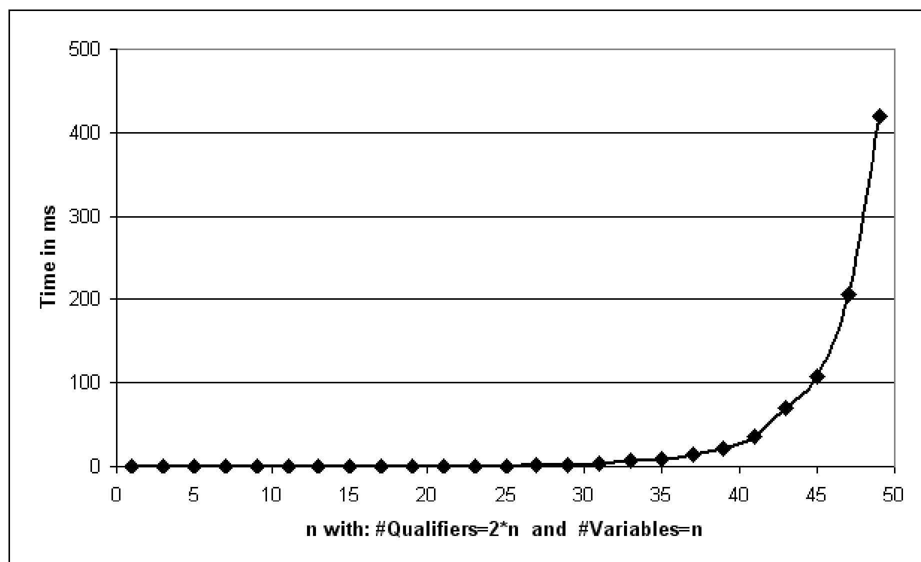


Figure 7.7: Measuring the satisfiability for expressions of growing complexity

With the three experiments we showed that the intersection algorithm for the XPath fragment $XP^{\{[],*,//\}}$ is very efficient. For the fragment $XP^{\{[],*,//,NOT\}}$ the check of satisfiability requires exponential expenses. Because the expected values for the runtime of the algorithms are acceptable for path expressions in database environments they can still be applied.

7.5 Related Work

In this section we give an overview over related work on analyzing the characteristics of path expressions. To the best of our knowledge the intersection of two XPath expressions ($p \cap p'$) has not been treated yet. The intersection of p and p' is the set that contains all XML nodes that are selected both by p and p'. In the context of indexes in XML databases the emptiness of the intersection of p and p' is a major issue when updating the index.

7.5.1 XPath 2.0

In XPath2.0 [132] it is possible to express the intersection using the operator `intersect` that intersects the result sets of two path expressions. But because this method operates on the result set we cannot check the intersection without any concrete XML data. Additionally, the evaluation of the two path expressions may take prohibitively long for huge XML data. In contrast to the `intersect` operator of XPath 2.0 our algorithm operates only on the path expressions themselves.

7.5.2 Containment and Satisfiability of XPath Expressions

The containment relation of XPath expressions is analyzed in a multitude of recent work [24, 74, 81, 82, 90, 106, 117]: Deutsch and Tannen [24] analyze the complexity of the containment relation for some XPath fragments. They give tight NP bounds for the disjunction-free fragment and show that containment remains decidable for more complex fragments.
The work of Miklau and Suciu [81, 82] provides an algorithm deciding the *containment* relation for two path expressions without accessing concrete XML data. The algorithm can also be used to check semantic equality if containment holds for both directions. Although possible in theory, we cannot use this algorithm to check the intersection (see Introduction).
Neven and Schwentick [90, 106] extend the work of Miklau and Suciu by analyzing the containment relation in the presence of disjunction and DTDs. They prove that the containment problem is hard for EXPTIME and DTDs complicate the problem.
A similar work is done by Wood [117] analyzing the decidability of the containment problem under DTD constraints.

In the context of XML indexes the containment relation is important for the query optimizer in order to find appropriate indexes for a given query. In general, an index covers all queries with a result set being a *subset* of the query that defined the index.

The satisfiability of path expressions is evaluated by Hidders [51] showing that it is NP-complete for various XPath fragments. The work [68] proves that the satisfiability of a path expression can be determined in polynomial time for some cases in the absence of a DTD or XML schema.
In contrast to [51, 68] we do not ask for the satisfiability of *one* XPath expression. Instead, we have two path expressions that are satisfiable independently. If we allow the *NOT* operator the question is whether they are both satisfiable with the same assignments of variables.

7.6 Updating KeyX Indexes

The intersection algorithm decides if a path expression of an update operation affects an index that is defined by the linear path expressions to its keys, qualifiers and the return value (see section 5.1.1). Therefore, the intersection has to be checked for all these path expressions because an affection may be caused by any of them. This section refines the update problem for XML indexes and describes how the underlying search trees that contains the keys and references to nodes in the XML data are maintained.

As defined in definition 20 in section 6.3 a database operation o is a tuple (p,$type$,i) consisting of a path expression $p \in XP^{\{[],*,//\}}$, a type $\in \{$*query*, *insert*, *change*, *delete*$\}$, and optional information i containing a new node n and/or a value v_n if the operation is an insert or a change. It may happen that the path expression p selects multiple nodes $p_1...p_m$. In this case the database operation is executed for each of the selected node. For instance the path expression $//article[year = 2005]$ selects a large set of `article` nodes. If the operation is a *delete* all of theses nodes are removed. If o is an *insert* the new node n (with an optional value v_n) is added to each of the nodes $p_1...p_m$.
In the following algorithms we concentrate on atomic database operations that operate on exactly one node n selected by p. If p selects more than one node, the algorithms are called for each $p_i \in \{p_1...p_m\}$.

The following atomic modifying database operations are supported:

- *insert:* insertion of a new node n that may have an optional value v_n,
- *delete:* deletes a node n,
- *change:* changes the value v_n of an existing node n.

If a node n is deleted its value does not matter because it cannot exist without n. Before deleting a node n that is selected by p we have to delete all descendants of n. This is done by calling a *delete* with the descendant operator added to p. For example, if $p = //book[year = 1999]$ deletes all books from a given year all elements that are attached under `book` are also removed. An index that covers the authors of books is therefore affected and must remove the 1999 entries. Before deleting the books we first remove all its descendants by calling a *delete* with $p' = p + //* = //book[year = 1999]//*$.

A *replace* operation that exchanges one node n by another node m is executed in two steps: A deletion of the n node followed by an insertion of the m node. Therefore, we can concentrate on update operations that handle only one node at a time.

7.6.1 Update Algorithm

The first step of the algorithm checks if an index affection has happened or not and calls the corresponding sub-routines. The intersection is analyzed for p and all path expressions that define the established indexes. If the operations is an *insert* or *delete*, the keys, qualifiers and the return value of each index must be checked because these two operations change the structure and the content of the XML data. A *change* operation changes only the content, therefore only the keys have to be checked whether an existing value has been modified. The pseudo code in listing 7.8 shows the first step:

```
1  forall i in indexes{            //check all indexes
2    K = i.getKeyPaths();          //extract the path expressions
3    Q = i.getQualiferPaths();     //that define an index
4    v = i.getReturnPath();

6    if (type==insert || type == delete){

8      if (type == insert)
9        execute o on XML data; // modify the XML data

11     forall key in K{            //check all keys
12       if intersection(key, p)!=ø
13         maintain(type,i,p,"keyAff");
14     }
15     forall qual in Q{            //check all qualifiers
16       if intersection(qual, p)!=ø
17         maintain(type,i,p,"qualAff");
18     }
19     if intersection(v, p)!=ø     //check the return value
20       maintain(type,i,p,"retAff");

22     if (type==delete)
23       execute o on XML data;  // modify the XML data
24   }

26   else if type==change){
27     forall key in K{             //check all keys
28       if intersection(key, p)!=ø
29         maintain(type,i,p,,"keyAff");
30     }
31     execute o on XML data;     // modify the XML data
32   }
33 }
```

Figure 7.8: Pseudocode of the update algorithm

Besides maintaining the indexes the modifying database operation must be performed on the underlying XML data as well. If o is a *delete* or a *change* this operation is performed after maintaining the indexes because we need access to the information that is going to be deleted/changed. For an *insert* o is performed before maintaining the indexes because we need the context of the new node.

7.6.2 Index Maintenance Algorithms

In the following we introduce the three algorithms that maintain the index's search tree after an insert, delete or change operation. These methods are called with the path expressions p, an index i that is affected by p and the optional new node n and/or its value v_n. Even if an index is affected it may be possible that it stays unchanged: If a delete operation addresses a non-existing node there are no entries in the index to be deleted. All relevant entries have to be found and maintained by the algorithms. For adding, removing or changing a value in a search tree like the B$^+$-Tree we refer to the literature, e.g.[26].

Insert Operation

Given a particular node n (that is reached by p) to be inserted we create a node set that consists of n plus its corresponding keys, qualifiers and return value. This is done by calculating the relative path expressions from p to the keys $k_1...k_n$, the qualifiers $q_1...q_m$ and the one return value v. These path expressions are evaluated on n leading to a set of nodes $k_{n_1}...k_{n_n}$ representing the keys, respectively $q_{n_1}...q_{n_m}$ representing the qualifiers and v_n representing the return value. Because the index is affected, one of the nodes $\{k_{n_1}...k_{n_n}\} \cup \{q_{n_1}...q_{n_m}\} \cup v_n$ is the node n. If one path expression has an empty result set the corresponding node is denoted with λ.
If n shall lead to a new entry in the index, all qualifiers must be fulfilled. Formally this means that all $q_{n_1}...q_{n_m} \neq \lambda$. If this is the case the tuple of the keys and the return value is inserted into the index.

Example 24 An XML data t consists of two books and one article. One `book` node has no children; the second book has one `isbn` child. The article has one `author` child. An insert operation with the path expressions $p = /dblp/book$ adds an `author` node to both books. The corresponding DOM-trees and the XUpdate statement are presented in figure 7.9.

```
1  <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2     <xupdate:append select="/dblp/book" >
3        <xupdate:element name="author">Jim</xupdate:element>
4     </xupdate:append>
5  </xupdate:modifications>
```

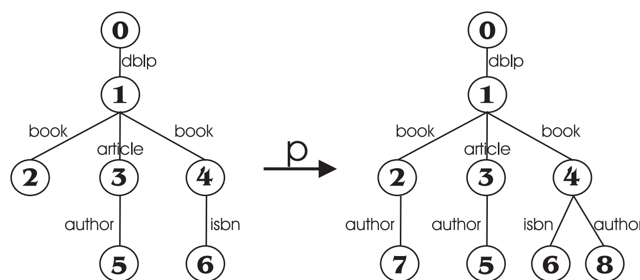


Figure 7.9: Adding an author node to the book nodes.

The one `article` node remains unchanged. Let us assume that an index i is defined by the path expression $p' = /dblp/book[isbn][author =' x']$. The path expression p selects the two book nodes with the id 2 and 4. Starting from p the qualifiers are reached by $/dblp/book/isbn - /dblp/book = /isbn$ and evaluated on the nodes 2 and 4 return the node 6 (the book with the id 2 has no `isbn` child). Analogously the keys are evaluated to the nodes with the ids 7 and 8. The path to the return values is the self axis (.) and therefore 2 and 4 are the two return values. The corresponding node sets N_1 and N_2 are:

$$N_1 = \{7\} \cup \{\lambda\} \cup 2, \quad N_2 = \{8\} \cup \{6\} \cup 4.$$

Because the first node set N_1 has a λ value for a qualifier the corresponding entry is not part of the index. The second node set has no λ values therefore a tuple consisting of the value of the key (8) and a reference to the return value (4) is inserted to the index i. □

A formal definition of the insert algorithm can be found below. The algorithm stops for a given $n \in p(t)$ whenever it is sure that a new entry for i is not necessary.

Given:
- a) Database operation $o = (p, \textit{insert}, i)$ with $i = (n, v_n)$,
- b) Affected index i with declaration $D = (K, Q, v)$,
- c) XML data t.

Output: Index consistent to t after execution of an **insert operation** o

Algorithm:

1 $\forall n \in p(t)$
2    $keyNodes = \emptyset;$
3    $qualNodes = \emptyset;$
4    $\forall q \in Q$
5      $qualPath = (q - n);$
6      $qualNode = qualPath(n);$
7   if $(qualNode == \lambda)$ break;
8   else $qualNodes = qualNodes \cup qualNode;$
9    $valPath = (v - n);$
10    $returnNode = valPath(n);$
11   if $(returnNode == lambda)$ break;
12    $\forall k \in K$
13      $keyPath = (k - n);$
14      $keyNodes = keyNodes \cup keyPath(n);$
15   if $(K == \emptyset)$ break;
16   else $i.add(keyNodes.value, returnNode);$
17 return $i;$

Delete Operation

For a delete operation that removes a node n from the XML data t we will analogously determine the node set $\{k_{n_1}...k_{n_n}\} \cup \{q_{n_1}...q_{n_m}\} \cup v_n$ containing the qualifiers, keys and the return value. If p selects no nodes at all, n and all other nodes are λ and there is no entry affected in the index i to be maintained.
Having the node set $\{k_{n_1}...k_{n_n}\} \cup \{q_{n_1}...q_{n_m}\} \cup v_n$ we now look at the type of affection: If a *qualifier* is affected the tuple of the keys and the return value can be immediately deleted because all qualifiers must be fulfilled by each entry of the index. Analogously, if p selects a *return value* of the index, the tuple is removed, because each entry need a reference to the return value. If p affects a *key* of i it makes a difference whether i is a single-key or multi-key index: For a single-key index the corresponding entry can be deleted because each entry needs a key. For a multi-key index the deleted key value is changed to `null` without affecting the other keys or the return value. A reorganization of this entry might be necessary.

Example 25 An example is illustrated in figure 7.10.

```
1 <xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate">
2    <xupdate:remove select="/dblp/book/isbn"/>
3 </xupdate:modifications>
```

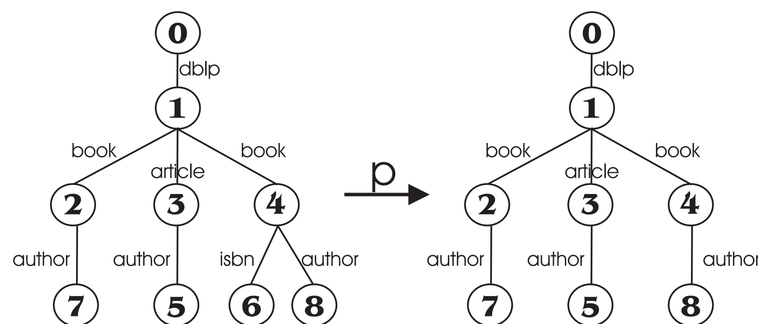


Figure 7.10: Deleting the isbn node.

The one existing `isbn` node n with the id 6 is removed by the a delete operation with the path expression $/dblp/book/isbn$. The intersection algorithm indicates that a qualifier of an index i defined by the path expression $p' = /dblp/book[isbn][author =' x']$ is affected. After building the node set $N_3 = \{8\} \cup \{\lambda\} \cup 4$ one can see that a qualifier is λ. Therefore, the corresponding entry must be removed from i. The removal of n in the XML data is performed after maintaining all affected indexes. This is done as the last step because we need n to determine its corresponding nodes. □

A formal definition of the delete algorithm can be found below.

Given:

a Database operation $o = (p, \textit{delete}, i)$ with $i = (n, v_n)$,

b Affected index i with declaration (K, Q, v) , .

c XML data t.

Output: Index consistent to t after execution of a **delete operation** o

Algorithm:

1 $\forall n \in p(t)$
2 $keyNodes = \emptyset$
3 $qualNodes = \emptyset$
4 $\forall q \in Q$
5 $qualNodes = qualNodes \cup n.eval(q - n)$
6 $\forall k \in K$
7 $keyNodes = keyNodes \cup n.eval(k - n)$
8 $returnNode = n.eval(v - n)$
9 $i.remove(keyNodes.value, returnnode)$
10 if key is affected AND i is a multi-key index
11 $i.add(keyNodes.value \backslash n.value, returnnode)$
12 return i

Change Operation

The algorithm for maintaining the index for a change operation is only called if a key is affected. The return value and the qualifiers are not affected by a change operation. This is because both are structural conditions that needs only attention if a node is deleted or added. The modification of value does not change the existence of a node.
If p selects at least one node n there must be an entry in i with a key that is not valid anymore. Therefore, the key value of the old entry is replaced by the new value v_n. The return value of the entry stays unchanged. The qualifier nodes need no attention, as they are not affected by value modification.

Chapter 8

KeyX Implementation Details

This chapter provides an overview of the architecture of the KeyX indexing system and gives some implementation details. KeyX with all its modules is implemented in over 110 Java classes, interfaces and exceptions that cannot be completely introduced in this thesis. Therefore, we concentrate on the most important aspects, methods and paradigms. The implementations were mainly done in the software developing environment *Eclipse*[25].

8.1 Architecture

KeyX is an application that works on top of a native XML database management system (XDBMS) and uses services that are provided through defined interfaces. This layered architecture abstracts from the particular implementation of the XML persistency and makes the exchange of the underlying modules a lot easier as all communication uses the interfaces. We have chosen the XDBMS Infonyte DB [55] because it comes with all required standardized interfaces like DOM or XPath. The architecture of Infonyte DB is shown in figure 8.1.

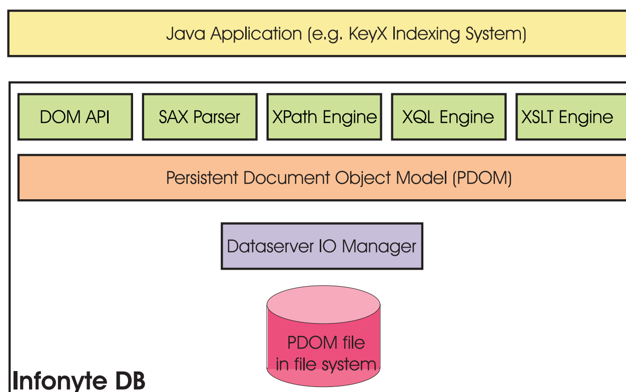


Figure 8.1: The basic architecture of the Infonyte DB XDBMS

KeyX is an application on top of Infonyte DB that may use any of the provided interfaces. Basically, all XML data are represented in the tree-structure DOM that is stored persistently as a PDOM document in the file system. Persistency is

a task of the XDBMS Infonyte DB so that the developer of an application has not to regard any persistency aspects like invoking serializations, etc.
Compared to highly sophisticated commercial database products, Infonyte DB is a rudimentary database management system. Infonyte DB does not offer advanced features like multi-user control mechanisms or transactions. In particular, it cannot be guaranteed that some stored XML data is always valid concerning a given schema, because elements can be inserted or deleted in the DOM without consistency controls. Validation is always performed on demand and then on the whole XML data so that it is too expensive to validate the data after each modification. Infonyte comes with no satisfying indexing approach; it supports some structural queries in a rudimentary way. The performance measurements of section 5.5 show that the query execution time of Infonyte DB without a KeyX index can become very long. Infonyte offers no XML update language like XUPdate. Therefore, all manipulations must be performed in a low level manner on the DOM tree. Recapitulating, one can say that Infonyte's state of development is not comparable to commercial relational products, so a company typically would not use it in productive environments. But this holds for most XML DBMS that are often initiated as research projects. An exception could be Tamino from the Software AG that is a fully commercial XDBMS. Anyhow, because Infonyte DB offers all features required for KeyX and the Software AG published only very few technical informations about Tamino the choice to use Infonyte DB was no fault.

KeyX uses the Infonyte DB interfaces DOM and XPath. The XPath engine is used to evaluate queries when building an index or when no index is available for processing a query. For a selected node in the DOM tree Infonyte can return a unique id. The time to dereference an id is constant and not dependent on the size of the XML data or the position of the element. Any XDBMS that offers these two features may be used with the KeyX implementation in principle. The XPath interface of the XDBMS is handed over to the database applications of the highest layer with the difference that covered queries are executed upon an index. The database application is not aware if an index is used or not because it still sends XPath queries and gets XML nodes. This architecture makes it easy to integrate an indexing system in existing applications. Therefore, the results of this thesis may be used universally for native XDBML; they do not dependent on the specific DBMS Infonyte DB. A block diagram of the KeyX system can be found in figure 8.2.

The architecture consists mainly of three parts: the query engine with the query optimizer as the central part, the ISP Tool that analyzes the workloads and optimizes the index configuration and a collection of the XML data, the workload and statistics. These data have to be stored persistently in order to survive system restarts. In a realistic environment it makes sense to log the workload for several weeks or months to get an impression of the typical usage of the database. The workload is a compressed file logging all occurred database operations in a

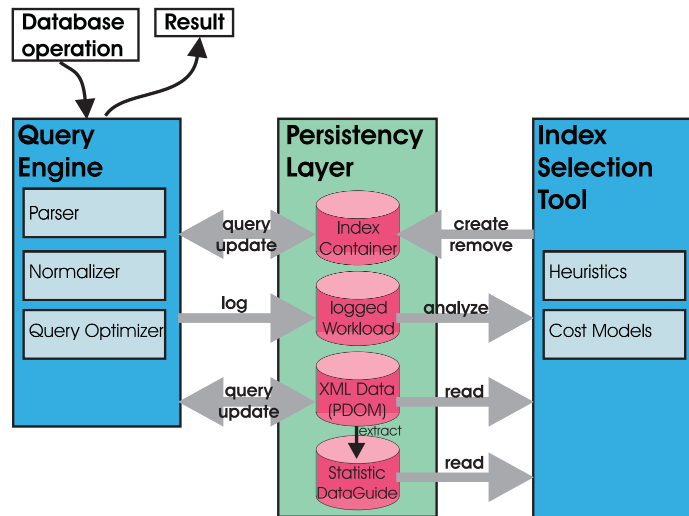


Figure 8.2: Block diagram of the KeyX indexing system

given period of time: Instead of logging each database operation independently, the workload extracts the path expression, creates an entry for each unique path expression and keeps some counters for their frequency and type. This way the workload has a moderate size. The indexes are Java classes that consist mainly of the Java `HashMap` as tree structure with the entries and some information about the index's declaration. The indexes are collected in the Index Container being a collection of indexes. In order to store all indexes persistently the Index Container and its content - the indexes - can be serialized into a stream and deserialized from the stream. The serialization uses standard Java technologies, i.e. the Index Container and the Index classes must implement the `Serializable` interface. The stream is redirected to the harddisk, so that it survives a system restart. The same holds for the statistic DataGuide. All theses persistent Components plus the XML data in the Infonyte DBMS are organized in a further layer - the KeyX `Persistency Layer`. Due to the layered architecture of KeyX it is possible to exchange or modify parts of the implementation while reducing side effects and costs that global changes would effect. The Persistency Layer is shown in figure 8.3.

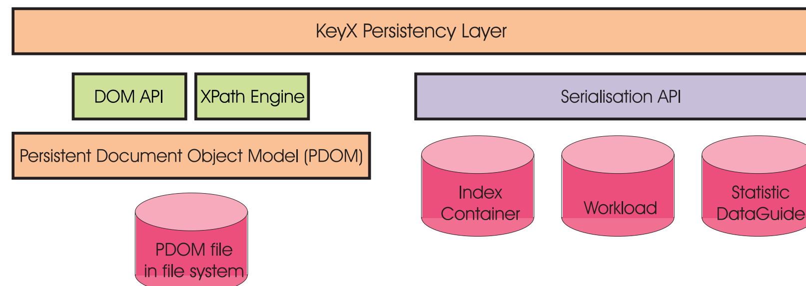


Figure 8.3: Architecture of the persistency layer of KeyX components

8.2 XPath expressions

Infonyte DB comes with an XQuery engine that supports XPath 1.0 expressions given by a String object. The internal representation of XPath expressions in Infonyte DB that is necessary to normalize or analyze path expression is not available (as open source) for developers and can therefore not be adopted or customized. Therefore, we defined own classes that represent database operations and path expressions for the XPath fragments used within this thesis. Additionally, XPath supports no modifying database operations. XPath expressions given by a String are parsed into our representation. A simplified UML diagram is shown in figure 8.4

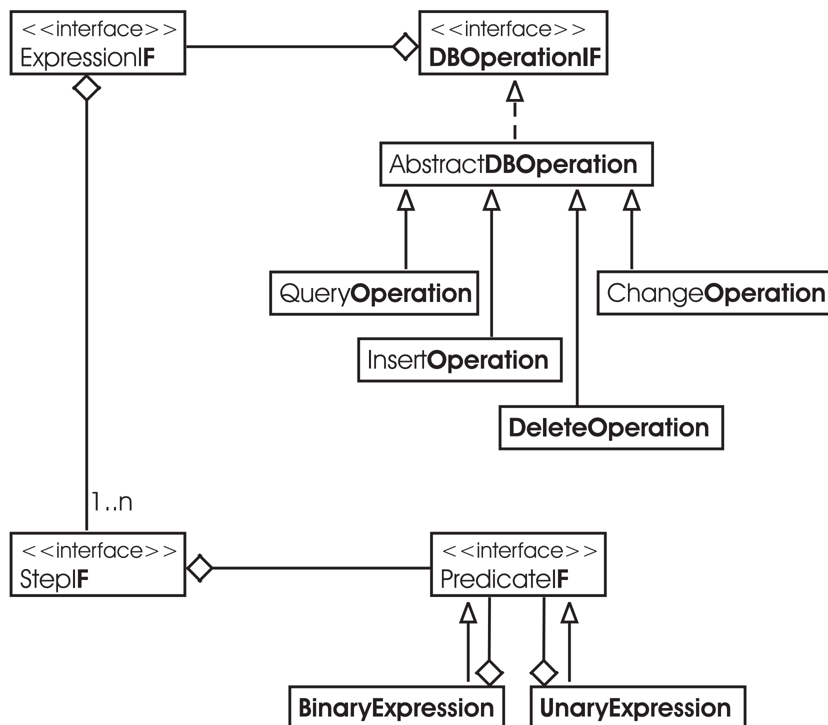


Figure 8.4: Simplified UML-diagram representing an XPath based database operation

Basically, the implementation follows the structure of path expressions as defined in section 2.3.1. An expression consists of one to many Steps with an axis and each step may contain predicates. Unary predicates have only one argument; they are used to express structural conditions (qualifiers) or to negate other predicates. Binary predicates have three arguments: The key, value for the comparison and the type of comparison $(=; \neq, <, >, \leq, \geq)$. Unary predicates and binary predicates both implement the `PredicateIF` interface. Because one predicate may contain others we have a recursive data structure. The evaluation of the path expressions with and without indexes is introduced extensively in section 5.4. The KeyX query engine offers an XPath interface used by database applications.

8.3 The Index Selection Tool

The module that finds a good index configuration and creates and drops them in the index container is a major part of the KeyX indexing approach. It can be called from a database administrator or automatically by a scheduler. In order to estimate the costs the ISP Tool needs access to the XML data, the workload and the statistic DataGuide. By the use of interfaces we again abstract the general functionality from its specific implementation. The ISP tool calls methods of the `AbstractISPSolver` that are implemented in an inheriting class that has a specific way of finding a solution for the optimization problem. We have implemented three approaches: an exact solution, that needs exponential time and two heuristics (see section 6.3.6 for algorithms). Due to the `AbstractISPSolver it` is possible to implement new heuristics without changing the remainder of the code. An interface to an `IndexCreator` allows the automatic creation of new indexes. An UML diagram of the ISP tool is shown in figure 8.5

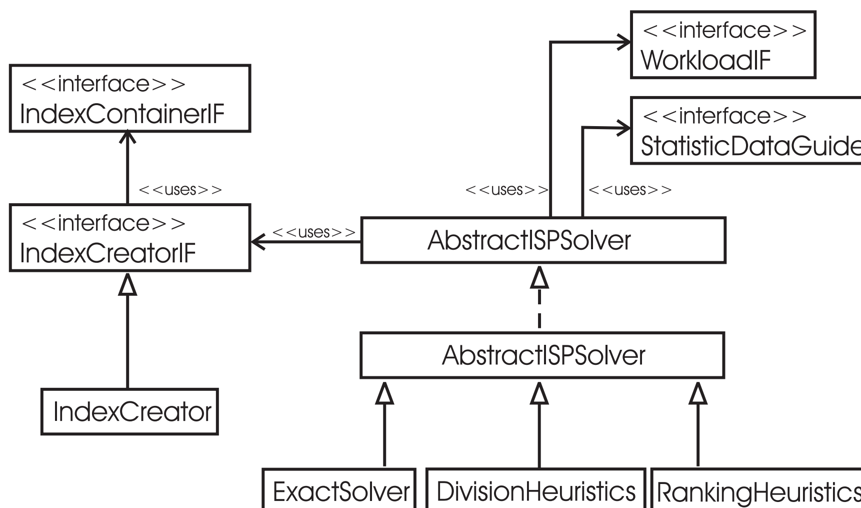


Figure 8.5: UML-diagram the index selection tool

8.4 Index Update Problem

In the context of the Index Update Problem that is reduced to the intersection of path expressions (see section 7) and solved by finite automata it is advantageous to use *Tree Patterns* as used in theoretical publications like [82]. Tree patterns are tree representations of path expressions, that can be parsed from and to strings. A Tree Pattern consists of a set of nodes (the node tests of XPath) and a set of edges (XPath axes) that connect the nodes. One node is marked to be the root of the tree. The nodes are denoted by `TPVertex`, the edges are `TPEdge` objects. Their UML diagrams are presented in figure 8.6.
Because we concentrate on linear path expressions when analyzing the inter-

Figure 8.6: UML-diagram for the tree pattern and its components

section problem the nodes of a Tree Pattern have one child at most. A non-deterministic final automaton is created for each Tree Pattern. The automaton is non-deterministic because the descendant axis in a Tree Pattern leads to multiple transitions for the same symbol.

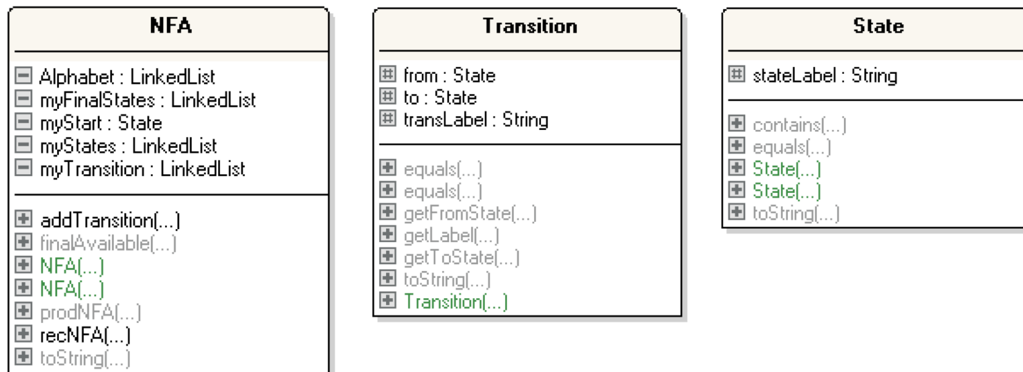


Figure 8.7: UML-diagram the finite automaton

The two automata A and A' representing the two path expressions to be analyzed for an intersection are the starting point for a product automaton. The product automaton can rely on the same implementation as the non-deterministic finite automaton with the difference that each input consists of two symbols and the number of states and transitions is significantly higher. A simple optimization waives the expensive construction of the full product automaton and processes an input on both automata A and A' if they have outgoing transitions for the same symbol. The UML diagrams are taken from [27].

8.5 XDLT - A Graphical User Interface for KeyX

A database management system is usually an underlying component that provides persistency services for applications of a higher level. Therefore, the core functions of a DBMS have no graphical user interface (GUI). Most commercial database products offer user GUIs for a more convenient configuration of settings. We use simple text files to define a few settings like temporary folders and input files. Output messages like errors are written to a log file that can be read with a text editor. One could create GUIs to facilitate these issues but the impact would be of minor value because these files are not touched very often.

KeyX offers an XPath interface to higher Java applications. Basically it is the task of this application to create XPath based queries and to process the results. In order to have a convenient interface that allows ad-hoc queries supported by an index we integrated KeyX into the *XML Distance Learning System (XDLT)* [48]. XDLT is a web based application for teaching and learning XML and major related technologies. It is based on Java Servlets (e.g. [96]) creating HTML pages to be viewed in a web browser.

The aim of XDLT is to facilitate and accelerate the learning and teaching of XML. The usual approach to study XML is to use a book or an interactive online tutorial. Both approaches use fixed examples that may be far away from the learner's context, requiring additional time to determine relevant parts. This may be a discouraging process and runs counter to instant success. A student who consults different books or tutorials has to integrate and transfer the examples him/herself. Lastly, there is usually no feedback for a teacher who wants to monitor the learning progress of the students in conventional approaches.

Figure 8.8: The file management of XDLT

With XDLT a teacher can provide his students an arbitrary XML data that cover the instructional scope the best. The same data can be used in exercises of adjustable granularity and scope. Currently, an exercise can be expressed upon the following XML technologies: XML itself, XPath, Document Type Definitions (DTD) and KeyX Indexes. Figure 8.9 illustrates how a teacher can manage his/her courses.

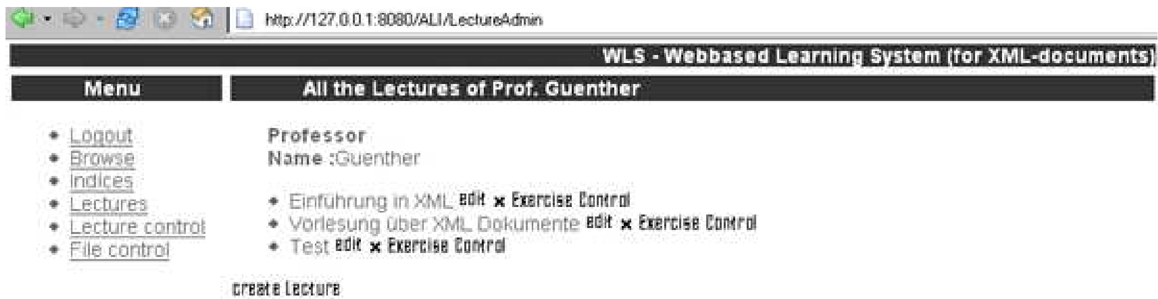


Figure 8.9: Managing lectures with XDLT

The teacher groups exercises to courses and assigns them to students. The learning progress can be monitored because exercises passed by an individual student are tracked and stored on the web server. An exercise based on XPath is shown in figure 8.10. This exercise has a task description that is not part of the figure. The selected nodes are highlighted and compared with a predefined correct solution. A context-sensitive assistance is offered by the system if the students proposed answer is not fully correct. For further details on this educational system we refer to [48].

Figure 8.10: The XDLT query interface

The XPath interface of KeyX and its established indexes are accessed by a servlet that generates HTML pages to create new indexes and to use them in order to execute ad-hoc queries. The queries can be performed with and without a suitable index in order to see the difference in execution time. Additionally, XDLT measures the time needed for the query execution. Therefore, XDLT can be seen

as a basic interface to the KeyX indexing system. Figure 8.11 shows the HTML page used to create or drop indexes whereas figure 8.12 shows an ad-hoc query with some measurements.

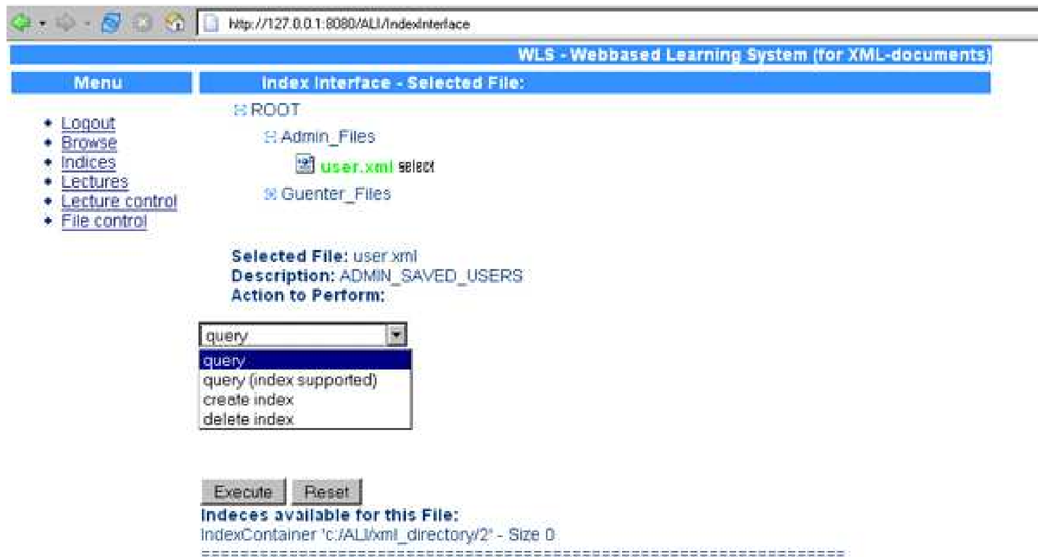


Figure 8.11: Maintaining KeyX indexes

Figure 8.12: Ad-hoc query supported by KeyX

XDLT is currently developed in a student research project (Studienarbeit). Because it is not finished yet further features and improvements are likely.

Chapter 9

Conclusion and Future Work

This thesis has introduced a new approach -called KeyX- for indexing semistructured XML data in a native XML Database Management System. A KeyX index is defined using keys that express a condition on the content and qualifiers expressing structural conditions. This approach supports the major path expressions of the XML query language XPath including queries with one to multiple keys, range queries, queries with wildcards or the descendant axis and structural queries without any value comparisons. Because the returned value of a path expression can directly be referenced by an index's entry, further costly processing in the XML data can be omitted. With structural qualifiers it is possible to demand further conditions that must be fulfilled by indexed elements; for instance it is possible to index only books that have an ISBN without paying attention to its particular value.
KeyX is selective to specific queries, this means that frequent queries are covered by indexes whereas less frequent queries are evaluated conventionally. This approach consumes less space, behaves better concerning updates and is tunable for specific workloads. For instance, in an auction scenario, it is advantageous to index the items' names and keywords but not their current price, that may change very often. Very often, the return value of a query that is wanted by a user or an application is not the indexed element, e.g. when querying a book by its title or an auction item by its keywords. The index entries in the KeyX search structure pay regard to this fact: The path to the return value of an index may differ from its key. Of course, it is possible that the return value and the indexed element are the same.
Performance measurements have proven the quality of KeyX for many query types.
In contrast to many other XML indexing approaches, KeyX needs no schema information of the data. This is an important requirement for semistructured data, where new element types may appear and disappear at runtime.

The problem of finding good indexes for a given workload consisting of querying and modifying database operations - the so called Index Selection Problem - was

transferred from the relational world to XML databases. Because XML database operations contain structural and content conditions the number of index candidates to be evaluated is significantly higher than in relational database management systems. Anyhow, it was shown that existing relational methods and heuristics can still be used with some XML specific adoptions.

Todays larger business applications with an underlying database cannot exist without indexes covering the most frequent queries. Usually, a human database administrator analyzes the current usage of the database and defines the indexes manually. Even having an optimal initial set of indexes when setting up a database, there is no guarantee that these indexes will suit future demands. Rather, it is realistic that the typical usage of the database will change after a while because new queries appear, for instance. In consequence, the existing indexes are suboptimal. The typical approach to face this problem is that the database administrator logs the workload, analyzes the performance of the database and the existing indexes and redefines indexes when necessary. These tasks are time consuming and require a skilled expert. In XML database management systems (XDBMS) this problem becomes even worse: Because XML queries cover both content and structure the number of possible queries and indexes is significantly higher. Additionally, for XML data without schema information, queries and indexes cannot be defined in advance, because the structure and the content of the data is not restricted. Both facts tend to result in higher maintenance costs for XML indexes compared to relational indexes. For large databases with a complex structure where a multitude of queries is expressible (e.g. DNA databases) it may be impossible to determine the best indexes manually.
We turned the KeyX-enriched XML database management system into an adaptive self-optimizing system that drops/creates indexes autonomously. With performance measurements in different scenarios we have shown that our approach adapts well to a changing usage of the database. This is a significant contribution to self-optimizing software systems that require less manual interaction and lead to lower costs of operation.

Whenever changing the indexed data in the database, the corresponding indexes must be updated in order to keep both data structures consistent. In relational databases where an index is defined upon a table and one or multiple columns the question whether an SQL update command affects an index is trivial: Whenever an indexed column receives a modification the corresponding index is affected. For XML indexes this problem is more difficult because path expressions may contain additional structure information with wildcards and the descendant axis. It is possible that an indexed node is affected that is not explicitly selected by the path expressions. We formalized this problem and introduced an algorithm based on finite automaton theory. The index affection problem is reduced to the intersection of two path expressions.

The implementations of this work operate on top of the native XML database management systems Infonyte DB [55], that can be interpreted as a persistent DOM with a Java API interface so that it can be used by any Java class. Because the KeyX implementation uses only the XPath engine of Infonyte DB and its capability to assign unique identifiers to XML nodes, any other XML database management system that offers the same functionality can be used in principle. Therefore, the KeyX approach is independent of any particular database product.

The results of this work have been published on four international conferences [44, 46, 47, 48], two workshops [43, 45] and two technical reports [41, 42] and can be used for the development of next-generation database management systems. Examples include new hybrid systems that unify relational and native XML paradigms like the SystemRX [11] from IBM.

Although the aims of this project were achieved there are a multitude of points where further work can start:

- So far only value comparisons with the operators $=, \neq, <, >, \leq, \geq$ are supported. For a lot of scenarios a value comparison with substrings or regular expressions would make sense. For instance it might be useful to find all books with XML appearing somewhere in the title. In principle full text indexing methodologies from the relational world can be reused within the KeyX structure.
- The intersection algorithms and the index maintenance algorithms may profit from the existence of schema information of the indexed data if available. We did not focus on data with a schema because XML database management systems have to support schemaless data in order not to restrict the capabilities and the expressiveness of the semistructured XML model. Having a schema information like a DTD or an XML schema the intersection problem may be easier because some combinations that cause an intersection are not enabled by the schema. On the other hand, a schema allows to define dependencies between elements (e.g. each book, that has a price has also a currency). If these dependencies are taken into account it is likely that the intersection problem is not getting easier. For XML data without a schema one could try to create a schema for the statistic DataGuide, that is a summary of the data
- The statistic DataGuide is a first approach to summarize the frequency of elements and the selectivity of their values. In particular, the statistic DataGuide assumes that all elements are statistically independent. If some elements are statistically dependent (see example above) the estimation of the query execution time may lead to inaccuracies. Basically, statistic dependencies can arise for any two elements a and b, and even worse for any

combinations of arbitrary many elements $a_1...a_k$. Therefore, there are exponentially many combinations to be checked for statistic dependency. Even if one is willing to spend this computational expenses there is the problem of storing the exponential many results - the factors that express the statistic dependence between elements. A first improvement may be to restrict the analysis on two elements and to store the factors only if a given threshold is exceeded. The time consuming analysis could take place when the database has a low working load.

- So far, KeyX directly supports a major fragment of XPath expressions that allow to express most relevant queries. Advanced XPath features like functions (e.g. count(), sum()) are not supported. XQuery expressions rely strongly on XPath for navigation and selection purposes, so that the KeyX approach can also be applied for the accelerated execution of XQuery expressions. Anyhow, because XQuery has a higher expressiveness than XPath, it might be interesting if and how KeyX can be extended in order to support XQuery more efficiently. For instance, XQuery has the concept of XML constructors embedding the selected element in a newly created XML structure. For frequently occurring Query expressions it might be profitable to precompute the embedding and store it in the KeyX search tree as return value. This way, KeyX would not only return the referenced element, but also its surrounding elements that are usually created by the XQuery constructors.

Chapter 10

Appendix

10.1 XQuery Example

```
1  let $data := doc("c:\xml\xmark\out.xml")
2  for $auction in $data//closed_auction
3  return
4   <summary>
5          <item>
6                  {       $data//item[@id=$auction/itemref/@item]/name     }
7          </item>
8          <seller>
9                  {       $data//person[@id=$auction/seller/@person]/name          }
10         </seller>
11         <buyer>
12                 {       $data//person[@id=$auction/buyer/@person]/name  }
13         </buyer>
14         <price>
15                 {data($auction /price)}
16         </price>
17 </summary>
```

10.2 Method testQualifier

```
1  void createIndex(XMLNode context,
2                   SetOfKeyPaths Kp,
3                   SetOfQualifierPaths Qp,
4                   ReturnValuePath rp,
5                   SearchTree tree){
6    if ( |Kp| == 1){           //Kp has only 1 key path
7      keyP = Kp[1];            //get path to this keys
8      relKeyP = keyP-context; //relative path to the keys from context node
9      relValP = rp-keyP;       //relative path to the return value

11     KEYS = eval(context, relKeyP);  //retrieve the key elements
12     forall (k in KEYS){
13       val = eval(k, relValPath);    //retrieve the return value
14       if (testQualifier (val, Qp)){ //test qualifier
15         int id = val.getId();       //get id of return node
16         tree.add(k.getText(), id);  //add key-value tuple
17       }
18     }
19   }
20   else{                             //build a multikey index
21     keyP = Kp[1];                   //take first KeyPath
22     Kp = Kp \ Kp[1];                //and remove it from Kp
```

```
23     relKeyP = keyP−context;          //relative path to keys from context

25     KEYS = eval(context, relKeyP);  //retrieve the key elements
26     forall (k in KEYS)
27       SearchTree tree2 = new SearchTree(); // create empty search tree

29       createIndex(k, Kp, Qp, rp, tree);    // recursive call of method

31       tree.add(k.getText(), tree2);        //nest the second tree
32     }
33   }
34 }
```

Figure 10.1: Pseudocode to create a single-key or multi-key index

10.3 XML Schema

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
       qualified" attributeFormDefault="unqualified">
3    <xs:element name="books">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="book" type="author_base_typ"/>
7        </xs:sequence>
8      </xs:complexType>
9    </xs:element>
10   <xs:complexType name="author_base_typ">
11     <xs:sequence>
12       <xs:element name="title" type="xs:string"/>
13     </xs:sequence>
14   </xs:complexType>
15   <xs:complexType name="author1_type">
16     <xs:complexContent>
17       <xs:extension base="author_base_typ">
18         <xs:sequence>
19           <xs:element name="author" type="author_typ"/>
20           <xs:element name="price" type="xs:integer"/>
21         </xs:sequence>
22       </xs:extension>
23     </xs:complexContent>
24   </xs:complexType>
25   <xs:complexType name="author2_type">
26     <xs:complexContent>
27       <xs:extension base="author_base_typ">
28         <xs:sequence>
29           <xs:element name="author" type="xs:string"/>
30         </xs:sequence>
31       </xs:extension>
32     </xs:complexContent>
33   </xs:complexType>
34   <xs:complexType name="author_typ">
35     <xs:sequence>
36       <xs:element name="first" type="xs:string"/>
37       <xs:element name="last" type="xs:string"/>
38     </xs:sequence>
39   </xs:complexType>
40 </xs:schema>
```

10.4 List of Publications

All publications of the KeyX project are listed here in chronological order.

Title: KeyX: ein selektiver schlüsselorientierter Index für das Index Selection Problem in XDBMS
Authors: Beda Christoph Hammerschmidt
Abstract: In relationalen Datenbank-Management-Systemem (RDBMS) werden Indizes verwendet, um spezifische und häufig wiederkehrende Anfragen zu beschleunigen. Die Auswahl von passenden Indizes ist ein wichtiger Prozess beim Anlegen und Optimieren der Datenbank, der meist von einem Administrator oder einem Index-Auswahl-Tool durchgeführt wird, welches eine Menge von passenden Indizes vorschlägt. Für die Indizierung von XML-Datenbanken/Dokumenten existieren zur Zeit noch keine Standardverfahren sondern verschiedene Ansätze in der wissenschaftlichen Literatur, die oft rein Pfad-basiert sind und keine oder wenig Auswahl bezüglich der indizierten Elemente zulassen. Ansätze, die das gesamte Dokument indizieren - sogenannte Structural Summaries- haben zwangsläufig einen hohen Speicherplatzbedarf und garantieren keine Leistungssteigerung, wenn häufig Änderungen am XML-Dokument vorgenommen werden, da Änderungen der Datenbank immer auch an der Indexstruktur vorgenommen werden müssen.
In dieser Arbeit wird das Konzept von spezifischen Indizes auf native XML-Datenbank-Management-Systeme (XDBMS) übertragen. Es wird eine Implementierung präsentiert, die frühere Anfragen nutzt, um die Datenbank zu optimieren, indem sie automatisch passende Indizes erstellt. Mit KeyX stellen wir einen Indizierungsansatz vor, der XML-Element- und Attributwerte mit spezifischem Pfad als Schlüssel interpretiert und die dazugehörenden oder benachbarten Knoten im Original-Dokument als Rückgabewert referenziert. Da sich Schlüssel und Wert unterscheiden können, entfällt der Aufwand, der für navigierende Pfadverfolgung zwischen beiden benötigt wird.
Wir transferieren das in der relationalen Welt wohlbekannte Index Selection Problem (ISP) auf XDBMS. Das ISP wird verwendet, um eine Menge von Indizes zu bestimmen, für die die Ausführungszeit eines Workloads von Datenbankoperationen minimal ist. Da der Workload periodisch analysiert wird und durch das ISP günstige Indizes automatisch angelegt und ggf. aufgelöst werden, garantiert die KeyX-Implementierung eine hohe Leistung über die gesamte Lebenszeit der Datenbank. Experimentell bestimmte Ergebnisse der auf einem nativen XDBMS basierenden prototypischen Implementierung zeigen, dass unser Ansatz die Ausführungszeit von Anfragen signifikant beschleunigt.
Published: in [43]

Title: On the Index Selection Problem applied to Key oriented XML Indexes
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann

Abstract: In the world of Relational Database Management Systems (RDBMS) indexes are used to accelerate specific queries. The selection of indexes is an important task in database-tuning which is performed by a database administrator or an index selection tool which suggests a set of suitable indexes. In this paper we transfer the concept of specific indexes to XML Database Management Systems (XDBMS) and present an implementation that uses occurring queries to optimize the performance of an XML database system by automatically creating suitable indexes . We introduce an index approach, called key oriented XML index, that uses specific XML element values and attribute values as keys referencing arbitrary nodes in the data. We transfer the wellknown Index Selection Problem (ISP) to XDBMS. Solving the ISP, a workload of database operations is analyzed and a set of specific indexes that minimizes the total execution time is suggested. Because the ISP is an NP complete problem, we apply heuristics to find a solution with reduced complexity. Experimental results of the prototypical implementation of the key oriented XML indexes on top of a native XDBMS demonstrate that our approach significantly improves the query execution time with only moderate additional storage requirements. Because the workload is analyzed periodically and suitable indexes are created or dropped automatically by solving the ISP, our approach guarantees high performance over the total life time of a database.
Published: in [41]

Title: A selective key-oriented XML Index for the Index Selection Problem in XDBMS
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann
Abstract: In relational database management systems indexes are used to accelerate specific queries. The selection of indexes is an important task when tuning a database which is performed by a database administrator or an index propagation tool which suggests a set of suitable indexes. In this paper we introduce a new index approach, called key-oriented XML index (KeyX), that uses specific XML element or attribute values as keys referencing arbitrary nodes in the XML data. KeyX is selective to specific queries avoiding efforts spent for elements which are never queried. This concept reduces memory consumption and unproductive index updates.
We transfer the Index Selection Problem (ISP) to XDBMS. Applying the ISP, a workload of database operations is analyzed and a set of selective indexes that minimizes the total execution time for the workload is suggested. Because the workload is analyzed periodically and suitable indexes are created or dropped automatically our implementation of KeyX guarantees high performance over the total life time of a database.
Published: in [44]

Title: Comparisons and Performance Measurements of XML Index Structures
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann
Abstract: Indexes are used to accelerate queries in database management systems (DBMS). In relational DBMS indexes are broadly explored whereas indexes in XML DBMS are still an active field of research. A multitude of approaches with different characteristics were introduced in the past. Approaches that are not *selective* to specific queries require the whole XML data to be indexed and may lead to enormous space consumption and poor performance if changes to the XML data occur often.
With KeyX we have introduced a selective and key-oriented approach for indexing only relevant parts of XML data in a database. This work provides qualitative comparisons and performance measurements of recent approaches in XML indexing. We motivate why key-oriented indexing that is derived from the relational world performs as well in the XML context.
Published: in [42]

Title: Autonomous Index Optimization in XML Databases
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann
Abstract: Defining suitable indexes is a major task when optimizing a database. Usually, a human database administrator defines a set of indexes in the design phase of the database. This can be done manually or with the help of so called *index wizard* tools analyzing predefined database operations. Even having an optimal initial set of indexes when setting up a database, there is no guarantee that these indexes will suit future demands. Rather, it is realistic that the typical usage of the database will change after a while because new queries appear, for instance. In consequence, the existing indexes are suboptimal. The typical way to handle this problem is that a database administrator maintains the database permanently.
In XML database management systems (XDBMS) this problem becomes even worse: Because XML queries cover both content and structure the number of possible queries and indexes is significantly higher. Additionally, for XML data without schema information, queries and indexes cannot be defined in advance, because the structure and the content of the data is not restricted. Both facts tend to result in higher maintenance costs for XML indexes compared to relational indexes.
In this paper we show by performance measurements that an adaptive XDBMS that analyzes its workload periodically and creates/drops XML indexes automatically guarantees a high performance over the total life time of a database. Although we present our index system called *KeyX* the idea and the results are transferable to other XML indexing approaches.
Published: in [45]

Title: The Index Update Problem for XML Data in XDBMS
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann
Abstract: Database Management Systems are a major component of almost every information system. In relational Database Management Systems (RDBMS) indexes are well known and essential for the performant execution of frequent queries. For XML Database Management Systems (XDBMS) no index standards are established yet; although they are required not less. An inevitable side effect of any index is that modifications of the indexed data have to be reflected by the index structure itself. This leads to two problems: first it has to be determined whether a modifying operation affects an index or not. Second, if an index is affected, the index has to be updated efficiently - best without rebuilding the whole index. In recent years a lot of approaches were introduced for indexing XML data in an XDBMS. All approaches lack more or less in the field of updates. In this paper we give an algorithm that is based on finite automaton theory and determines whether an XPath based database operation affects an index that is defined universally upon *keys*, *qualifiers* and *a return value* of an XPath expression. In addition, we give algorithms how we update our KeyX indexes efficiently if they are affected by a modification. The *Index Update Problem* is relevant for all applications that use a secondary XML data representation (e.g. indexes, caches, XML replication/synchronization services) where updates must be identified and realized.
Published: in [47]

Title: XDLT: A Distance Learning Tool for consistent teaching of XML and related Technologies
Authors: B. C. Hammerschmidt, P. Stursberg, J. Jungclaus and V. Linnemann
Abstract: The eXtended Markup Language (XML) has become an important data format in the e-learning world during the past years. A multitude of e-learning systems take advantage of XML for various purposes: to represent knowledge or content, for information exchange between distributed applications or just for platform-independent storage of data. Although XML reflects a technical issue of data representation and application architecture in most cases, an emerging need for students and teachers to learn XML and XML related technologies can be observed. For instance, a person who describes entities of a given domain with an XML-based ontology needs domain-specific knowledge and a certain degree of XML skills to express the knowledge. Current approaches to learn XML such as tutorials and XML editors lack in the field of guidance, monitoring of the learning process and interoperability of different XML related technologies like XML data modeling (DTD), XML transformation and query as well as update languages (XPath, XUpdate).
With this paper we introduce a web-based distance teaching and learning system teaching fundamentals of XML and major XML related technologies. In contrast to interactive tutorials that operate mostly with fixed XML examples and XML editors which offer no guidance for the learner, our approach enables a student to learn XML and related technologies based on custom data and exercises that can be defined and monitored by a teacher.
Published: in [48]

Title: On the Intersection of XPath Expressions
Authors: B. C. Hammerschmidt, M. Kempa and V. Linnemann
Abstract: XPath is a common language for selecting nodes in an XML document. XPath uses so called *path expressions* which describe a navigation path through semistructured data. In the last years some of the characteristics of XPath have been discussed. Examples include the containment of two XPath expressions p and p' ($p \subseteq p'$). To the best of our knowledge the intersection of two XPath expressions ($p \cap p'$) has not been treated yet. The intersection of p and p' is the set that contains all XML nodes that are selected both by p and p'. In the context of indexes in XML databases the emptiness of the intersection of p and p' is a major issue when updating the index. In order to keep the index consistent to the indexed data, it has to be detected if an index that is defined upon p is affected by a modifying database operation with the path expression p'.
In this paper we introduce the intersection problem for XPath and give a motivation for its relevance. We present an efficient intersection algorithm for XPath expressions without the NOT operator that is based on finite automata. For expressions that contain the NOT operator the intersection problem becomes NP-complete leading to exponential computations in general. With an average case simulation we show that the NP-completeness is no significant limitation for most real-world database operations.
Published: in [46]

Bibliography

[1] The AberdeenGroup. Database cost study. URL: http://relay.bvk.co.yu/progress/aberdeen/aberdeen.htm, 1998.

[2] Serge Abiteboul, Dan Suciu, and Peter Buneman. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, USA, 1st edition edition, 1999.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman:. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Altova. XMLSpy . URL: http://www.altova.com.

[5] Anders Møller and Michael I. Schwartzbach . The XML Revolution: Technologies for the future web. URL: www.brics.dk/NS/01/8/BRICS-NS-01-8.ps.gz, 2001.

[6] Andreas Laux . Lexus: XML Update Language, Working Draft. URL: http://www.infozone-group.org/lexusDocs/html/wd-lexus.html.

[7] Apache. Xalan XSLT processor . URL: http://xml.apache.org/xalan-j/.

[8] Apache. Xindice. URL: http://xml.apache.org/xindice/.

[9] M. Barg and R. Wong. A Fast and Versatile Path Index for Querying SemiStructured Data. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA 2003)*, Kyoto, Japan, 26 - 28 March 2003.

[10] Michael G. Bauer, Frank Ramsak, and Rudolf Bayer. Multidimensional Mapping and Indexing of XML. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings of the 10th BTW Conference: Datenbanksysteme für Business, Technologie und Web*, volume 26 of *LNI*, pages 305–323, Leipzig, Germany, February 26.-28 2003. GI.

[11] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang1. System RX: One Part Relational, One Part XML . In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA, June 14-16 2005. ACM Press.

[12] Alberto Caprara, Matteo Fischetti, and Dario Maio. Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):955–967, December 1995.

[13] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 34–43. ACM Press, 1998.

[14] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997.

[15] Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 367–378. ACM Press, 1998.

[16] Surajit Chaudhuri and Vivek R. Narasayya. Microsoft Index Tuning Wizard for SQL Server 7.0. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 553–554. ACM Press, 1998.

[17] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 134 – 144, San Diego, California, USA, 2003.

[18] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[19] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: an adaptive path index for XML data. In *SIGMOD 2002, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA*, pages 121–132. ACM Press, 2002.

[20] Douglas Comer. The Difficulty of Optimum Index Selection. *ACM Transactions on Database Systems*, 3(4):440–445, December 1978.

[21] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Ojio, USA, 1971. Shaker Heights.

[22] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of 27th International Conference on Very Large Data Bases*, Roma, Italy, September 11-14 2001. Morgan Kaufmann.

[23] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[24] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proceedings of the 8th Int. Workshop on Knowledge Representation Meets Databases, 2001*, 2001.

[25] Eclipse.org Main Page -. Eclipse - an open extensible IDE . URL: `http://www.eclipse.org/`.

[26] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fourth edition edition, 2003.

[27] Konstantin Ens. Updates in XML-Indizes. Research thesis, Institute for Information Systems, University of Lübeck, 2004.

[28] Peter Fankhauser, Gerald Huck, and Ingo Macherius. PDOM and XQL-Prozessor Components for Data Intensive XML Applications. Technical report, German National Research Center for Information Technology (GMD), Integrated Publication and Information Systems Institute (IPSI), 2000.

[29] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292 – 314, 2002.

[30] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Natix: A Technology Overview. In *Revised Papers from the Node 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 12–33, London, UK, 2003. Springer-Verlag.

[31] Vladimir Gapeyev and Benjamin C. Pierce. Regular Object Types. In *ECOOP*, pages 151–175, 2003.

[32] Tim Gehrmann and Alex Pfalzgraf. Indizierung von schemalosen XML-Dokumenten anhand von XPath-Anfragen. Bachelor thesis, Institute for Information Systems, University of Lübeck, 2004.

[33] J. P. Gelb. System–managed storage. *IBM Systems Journal*, 28(1):77–103, 1989.

[34] Georg Gottlob and Christoph Koch and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 95–106, Hong Kong, China, August 20-23 2002.

[35] Georg Gottlob and Christoph Koch and Reinhard Pichler. The complexity of XPATH query evaluation. In *In Proceedings of the ACM Symposium on Principle of Databases Systems (PODS)*, pages 179–190, San Diego, CA, USA, June 9-12 2003.

[36] Charles F. Goldfarb and Yuri Rubinsky. *The SGML Handbook*. Oxford University Press, 1991.

[37] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.

[38] Torsten Grust. Accelerating XPath location steps. In *SIGMOD 2002, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA*, pages 109 – 120. ACM Press, 2002.

[39] Khaled Haj-Yahya. Erkennen der Teilmengen-Beziehung von zwei XPath-Anfragen. Bachelor thesis, Institute for Information Systems, University of Lübeck, 2005.

[40] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis D. Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed mode XML query processing. In *VLDB'03, Proceedings of 29rd International Conference on Very Large Data Bases*, Berlin, Germany, September, 9-12 2003.

[41] Beda C. Hammerschmidt, Martin Kempa, and Volker Linnemann. On the Index Selection Problem applied to Key oriented XML Indexes. Technical report, A-04-09, Institute of Information Systems, University of Lübeck, 2004.

[42] Beda C. Hammerschmidt, Martin Kempa, and Volker Linnemann. Comparisons and Performance Measurements of XML Index Structures. Technical report, A-05-09, Institute of Information Systems, University of Lübeck, 2005.

[43] Beda Christoph Hammerschmidt. KeyX: ein selektiver schlüsselorientierter Index für das Index Selection Problem in XDBMS. In Stefan Conrad Mireille Samia, editor, *Tagungsband zum 16. GI-Workshop über Grundlagen von Datenbanken (16th GI-Workshop on the Foundations of Databases)*, pages 63–67, Monheim, Germany, June 1-4 2004.

[44] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. A selective key-oriented XML Index for the Index Selection Problem in XDBMS. In Galindo F., Takizawa M., and Traunmüller R., editors, *Proceedings of the 15th International Conference on Database and Expert Systems Applications - DEXA '04*, number 3180 in Lecture Notes in Computer Science, pages 273–284, Zaragoza, Spain, August 30 - September 3 2004. Springer.

[45] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. Autonomous Index Optimization in XML Databases. In *Proceedings of the International Workshop on Self-Managing Database Systems (SMDB 2005)*, pages 56–65, Tokyo, Japan, April 8-9 2005.

[46] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. On the Intersection of XPath Expressions. In *Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS 2005)*, Montreal, Canada, July 25-27, 2005.

[47] Beda Christoph Hammerschmidt and Volker Linnemann. The Index Update Problem for XML Data in XDBMS. In *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, pages 27–34, Miami, USA, May 24-28 2005.

[48] Beda Christoph Hammerschmidt, Philipp Stursberg, Jan Jungclaus, and Volker Linnemann. XDLT: A Distance Learning Tool for consistent teaching of XML and related Technologies. In *EDMEDIA*, Montreal, Canada, 2005.

[49] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell.* O'Reilly, third edition, 2004.

[50] M. Harren, B. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical report rc23007, IBM Research, 2003.

[51] Jan Hidders. Satisfiability of XPath Expressions. In *Proceedings of the 9th International Workshop on Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 21–36, Potsdam, Germany, 2004. Springer.

[52] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley Publishing Company, 2001.

[53] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions of Internet Technology (TOIT)*, 3(2):117–148, 2003.

[54] IBM Corporation. IBM DB2 XML Extender. URL: `http://www-3.ibm.com/software/data/db2/extenders/xmlext/`.

[55] Infonyte GmbH. Infonyte DB. URL: `http://www.infonyte.com`, 2003.

[56] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native XML database. Technical report, University of Michigan, http: //www.eecs.umich.edu/db/timber/, 2002.

[57] H. Jiang, H. Lu, W. Wang, and B. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Join. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), India*, 2003.

[58] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133 – 144. ACM Press, 2002.

[59] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 779 – 790. ACM Press, 2004.

[60] Wassilios Kazakos, Andreas Schmidt, and Peter Tomczyk. *Datenbanken und XML. Konzepte, Anwendungen, Systeme* . Springer, Berlin, 2002.

[61] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In *Proceedings of the international conference Datenbanksysteme für Business,Technologie und Web(BTW)*, pages 227–246, Leipzig, Gemany, February 26-28 2003.

[62] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An XML Indexing Structure with Relative Region Coordinate. In IEEE Computer Society 2001, editor, *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, pages 313–320, Heidelberg, Germany, April 2-6 2001.

[63] Leonid Khachian. A Polynomial Algorithm in Linear Programming. *Soviet Mathematics Doklady*, 10:191–194, 1979.

[64] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[65] Nils Klarlund, Anders Møller, , and Michael I. Schwartzbach. DSD: A Schema Language for XML. In *Proceedings of the second workshop on Formal methods in software practice (FMSP 2000)*, Portland, Oregon, USA, August 22-25 2000. ACM Press.

[66] Josef Kratica, Ivana Ljubić, and Dušan Tošić. A Genetic Algorithm for the Index Selection Problem. In G. R. Raidl et al., editor, *Applications of Evolutionary Computing: EvoWorkshops2003*, volume 2611 of *LNCS*, pages 281–291, University of Essex, England, UK, 14-16 April 2003. Springer-Verlag.

[67] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In Rainer Unland Akmal B. Chaudhri Zohra Bellahsene, Erhard Rahm, Michael Rys, editor, *Proceedings of the First International XML Database Symposium (XSym 2003)*, pages 1–18, Berlin, Germany, 2003. Springer-Verlag.

[68] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui (Wendy) Wang, and Zheng (Jessica) Zhao. On Testing Satisfiability of Tree Pattern Queries. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 120–131, Toronto, Canada, 2004. Morgan Kaufmann.

[69] Latex Project. LaTeX A document preparation system . URL: `http://www.latex-project.org/`.

[70] Michael Ley. Digital Bibliography & Library Project. URL: `http://dblp.uni-trier.de`, 2001. Computer Science Bibliography.

[71] Linux Doc. The Linux Documentation Project . URL: `http://www.tldp.org/`.

[72] Guy M. Lohmann. Self-Managing Research at IBM Almaden. In *Invited Talk of the International Workshop on Self-Managing Database Systems (SMDB 2005)*, pages 56–65, Tokyo, Japan, April 8-9 2005.

[73] Manfed Päßler and Jane Xu and Josephine Cheng. An XML Enablement of DB2 – DB2 XML Extender. URL: `http://www.old.netobjectdays.org/pdf/00/slides/paessler.pdf`, October 10-12, 2000.

[74] Maarten Marx. XPath with Conditional Axis Relations. In *Proceeding of the 9th International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 477 – 494, Heraklion, Crete, Greece, 2004.

[75] Florian Massel. Implementierung und Evaluation von Index Fabrics zur Indizierung von XML-Daten. Bachelor thesis, Institute for Information Systems, University of Lübeck, 2004.

[76] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database managment system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[77] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing semistructured data. Technical report, Stanford University, 1998. Computer Science Department.

[78] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, pages 169–183, 2002.

[79] Microsoft Corporation. SQL Server 2005 Express. URL: `http://www.microsoft.com/sql/express/`.

[80] Microsoft Corporation. SQL Server 2000 Web Services Toolkit. URL: `http://www.microsoft.com/sql/techinfo/xml/default.asp`, 2002.

[81] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 65–76. ACM Press, 2002.

[82] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[83] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of Database Theory - ICDT '99, 7th International Conference*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295, Jerusalem, Israel, January 10-12 1999. Springer.

[84] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and Easy Distributions for SAT Problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[85] Donald R. Morrison. PATRICIA-Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, Oktober 1968.

[86] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[87] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Transactions of Internet Technology (TOIT)*, 2005.

[88] Peter Murray-Rust. Chemical Markup Language: A Simple Introduction to Structured Documents. *World Wide Web Journal*, 2(4):135–147, 1997.

[89] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N.n Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.

[90] Frank Neven and Thomas Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables . In *Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *Lecture Notes In Computer Science*, pages 315 – 329. Springer, 2003.

[91] Norman Walsh. A Technical Introduction to XML . URL: `http://xml.com/pub/a/98/10/guide0.html`, 1998.

[92] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. Symmetry in XPath. In *Proceedings of Seminar on Rule Markup Techniques, no. 02061, Schloss Dagstuhl, Germany (7th February 2002)*, 2001.

[93] Open University of the Netherlands. Educational Modelling Language (EML). URL: `http://eml.ou.nl`.

[94] Oracle Corporation. Oracle Tuning Pack. URL: `http://technet.oracle.com/products/oem/files/tp.html`, 2002.

[95] Oracle Corporation. Oracle XML DB. URL: `http://otn.oracle.com/tech/xml/xmldb/index.html`, 2003.

[96] Bruce W. Perry. *Java Servlet & JSP Cookbook* . O'Reilly, first edition, 2003.

[97] Gartner Group/Dataquest report. Server Storage and RAID Worldwide. URL: `http://www.gartner.com`, 1999.

[98] ITCentrix report. Storage on Tap: Understanding the Business Value of Storage Service Providers. URL: `http://www.itcentrix.com/`, 2001.

[99] Rick Jelliffe). Schematron: An XML Structure Validation Language using Patterns in Trees. URL: `http://xml.ascc.net/resource/schematron/schematron.html`.

[100] Robin Cover. The Essence and Quintessence of XML. Retrospects and Prospects . URL: `http://www.xml.org/xml/essence_of_xml.shtml`, 1998.

[101] Sablotron. XSLT, DOM and XPath processorcurrent version: 1.0.2. URL: `http://www.gingerall.com/charlie/ga/xml/p_sab.xml`.

[102] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: Continuous Query-driven Index Tuning. In *VLDB 2003,Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*. Morgan Kaufmann, 2003.

[103] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, volume LNCS 2992, pages 237–255, Heraklion - Crete, Greece, March 14 - 18 2004.

[104] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, 2002.

[105] Harald Schöning. Tamino - A DBMS designed for XML. In *Proceedings of the 17th International Conference on Data Engineering*, pages 149–154, Heidelberg, Germany, April 2-6 2001. IEEE Computer Society.

[106] Thomas Schwentick. XPath query containment. *ACM SIGMOD Record*, 33(1):101 – 109, 2004.

[107] Sleepycat Software. Berkeley DB XML. URL: `http://www.sleepycat.com/products/xml.shtml`.

[108] Stylus Studio. Xpath Processor. URL: `http://www.stylusstudio.com/docs/v62/d_xpath2.html`.

[109] Troff. The Text Processor for Typesetters . URL: `http://www.troff.org/`.

[110] Uddi.org. Universal Description, Discovery and Integration (UDDI). URL: `http://www.uddi.org/`.

[111] Gary Valentin, Michael Zuliani, and Daniel C. Zilio. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*. IEEE Computer Society, 2000.

[112] Jean-Yves Vion-Dury and Nabil Layaïda. Containment of XPath expressions: an Inference and Rewriting based approach. In *Proceedings of Extreme Markup Languages*, Montreal, Quebec, Canada, 4-8 august 2003.

[113] W3 Schools. XML Tutorial. URL: http://www.w3schools.com/xml/.

[114] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In ACM Press, editor, *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110 – 121, San Diego, California, USA, June 9-12 2003.

[115] Felix Weigel, Holger Meuss, François Bry, and Klaus U. Schulz. Content-Aware DataGuides for Indexing Large Collections of XML Documents. Forschungsbericht/research report PMS-FB-2003-14, Institute of Informatics, University of Munich, 2003.

[116] Felix Weigel, Holger Meuss, Klaus U. Schulz, and François Bry. Content and structure in indexing and ranking XML . In *Proceedings of the 7th international workshop on the web and databases (WebDB)*, pages 67 – 72, Paris, France, 2004. ACM Press.

[117] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *Proceedings of the 9th International Conference of Database Theory (ICDT)*, number 2572 in Lecture Notes in Computer Science, pages 300–314. Springer, 2003.

[118] World Wide Web Consortium (W3C). Document Object Model (DOM). URL: http://www.w3.org/DOM/.

[119] World Wide Web Consortium (W3C). Extensible Markup Language (XML). URL: http://www.w3.org/XML/.

[120] World Wide Web Consortium (W3C). HyperText Markup Language (HTML) . URL: http://www.w3.org/MarkUp/.

[121] World Wide Web Consortium (W3C). Mathematical Markup Language (MathML). URL: http://www.w3.org/Math/.

[122] World Wide Web Consortium (W3C). Overview of SGML Resources . URL: http://www.w3.org/MarkUp/SGML/.

[123] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.1. URL: http://www.w3.org/TR/soap/.

[124] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1. URL: http://www.w3.org/TR/wsdl.

[125] World Wide Web Consortium (W3C). XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). URL: http://www.w3.org/TR/xhtml1/.

[126] World Wide Web Consortium (W3C). XML Path Language (XPath). URL: http://www.w3.org/TR/xpath.

[127] World Wide Web Consortium (W3C). XML Schema. URL: http://www.w3.org/XML/Schema.

[128] World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language. URL: `http://www.w3.org/TR/xquery`.

[129] World Wide Web Consortium (W3C). XSL Transformations (XSLT). URL: `http://www.w3.org/TR/xslt`.

[130] World Wide Web Consortium (W3C). Document Definition Markup Language (DDML) Specification. URL: `http://www.w3.org/TR/NOTE-ddml`, 1999.

[131] World Wide Web Consortium (W3C). Schema for Object-Oriented XML 2.0. URL: `http://www.w3.org/TR/NOTE-SOX/`, 1999.

[132] World Wide Web Consortium (W3C). XML Path Language (XPath) 2.0. URL: `http://http://www.w3.org/TR/xpath20/`, 2004.

[133] XML DB Initiative. XML:DB Initiative for XML Databases. URL: `http://xmldb-org.sourceforge.net/`.

[134] XUpdate Working Group . XUpdate: XML Update Language. URL: `http://xmldb-org.sourceforge.net/xupdate/`.

[135] Pavel Zezula, Giuseppe Amato, and Fausto Rabitti. Processing XML Queries with Tree Signatures. In H. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, and G. Weikum, editors, *Intelligent Search on XML Data Applications, Languages, Models, Implementations, and Benchmarks*, Lecture Notes in Computer Science , Vol. 2818, pages 247 – 258, 2003.

Index

D, 78, 108, 110, 111, 144
H_{div}, 115
H_{rank}, 115
P_l, 48
$P_{l_{abs}}$, 50
$P_{l_{rel}}$, 50
T, 125
$XP^{\{[],*,//,NOT\}}$, 123
$XP^{\{[],*,//\}}$, 49, 108, 123, 141
$XP^{\{[],*,//\}}_{abs}$, 50
$XP^{\{[],*,//\}}_{rel}$, 50
Mod(p), 50, 125
$\Sigma(p)$, 52, 129–132
nodes(p), 51
$p(n)$, 50
$p(t)$, 50
r, 92
sl, 92
$t.nodes$, 47
$t.root$, 47
1-Index, 60
1NF, 103
2-Index, 60
3NF, 39

bisimilar, 58

absolute location path, 23
adaptive, 117
Ancestor, 11
APEX, 63, 73
Attribute, 22, 77
Attributes, 11
Axis, 23

B^*Tree, 111
B^+Tree, 71, 77, 82, 143
BCNF, 39
Bisimulation, 58
BLOB, 42

Cache, 123
CALS, 8
Change, 141
Chemical ML, 20
Child, 10
CLOB, 42
Collection, 23, 42, 84
Complement, 125
Condition, 83
Connection index, 66
Containment, 90, 91, 94, 112, 125, 126, 140
content - centric, 68
Content Management System, 20, 38
Content-Aware DataGuide, 68
Context-node, 24
Covering index, 66

D(k)-Index, 66
Data type, 16, 29
Database administrator, 103, 106, 108, 117, 151
Database operation, 6, 47, 103–106, 108, 110, 113–115, 117, 118, 120, 121, 136–138, 141, 142, 149, 157
DataGuide, 58, 68, 92
DB2, 117
DBA, 103
DBLP, 11, 12, 84, 85, 101, 124
DBMS, 104, 117
DDML, 19
Delete, 141
Descendant, 11, 93
Descendant query, 28
descriptive markup, 7
Design phase, 103
Designator, 71
Document Access Definition, 41

Document Definition Markup Language, 19
Document Object Model, 19, 45
Document Structure Description, 19
Document Type Definition, 8, 14, 17, 103, 159
DOM, 19, 43, 45, 86, 148
DSD, 19
DTD, 8, 14, 17, 39, 103, 124, 159

Eclipse, 147
Element, 10, 14, 22, 51, 77, 82
eXist, 42
Extent, 58, 64

Finite automata, 129
Finite automaton, 123
FLWOR expressions, 32
For, 32
Formal languages, 52
Forward-and-Backward-Index, 66

General path expression, 48, 49, 78
Genetic algorithm, 107
GUI, 153

Hashtable, 64, 73, 77, 82
Heuristics, 6, 105, 107, 158
Homomorphism, 130
Hopi, 66
HTML, 8
Hybrid index, 56
Hybrid indexing approaches, 67

IBM XML Extender, 41
Index adviser, 103
Index candidates, 158
Index declaration, 78, 108, 110, 111, 144
Index Fabric, 56, 71
Index Selection Problem, 6, 104, 157
Index Update Problem, 151
Index wizard, 103
Infonyte DB, 42, 97, 115, 117, 147
Insert, 141
Intersection Problem, 125
Inverted list, 66, 68
ISO 8879, 7

Java, 20, 22, 72, 82, 83, 97, 101
Join, 5, 34, 39–41, 58, 68, 71

Key, 39, 77, 78, 82, 89, 95, 111
key, 150
Key Path, 82
Key query, 56
KeyX, 56, 71, 73, 75, 77, 104, 157
KeyX Index Declaration, 78
KeyX Persistency Layer, 149
Kleene closure, 52
Knapsack problem, 105, 107

Layer, 82
Let, 32
Lexus, 35
Linear path expression, 48, 62, 78
Linear programming, 106
Linux HOWTO, 8
Linuxdoc, 8
Location step, 23
Lore, 58, 66

Markup, 7
MathML, 20
Microsoft Internet Explorer, 31
Microsoft SQL Server, 41
mixed content, 15
Model, 50
Multi-key index, 72, 83, 84, 87, 95, 109, 145, 146, 157
Multi-key query, 28
MySQL, 38

Namespace, 45
Native XML DBMS, 5, 31, 42, 43, 97, 104, 115, 117, 157
Natix, 42
Node, 45, 51, 105, 108, 109, 111, 113, 117, 131, 132, 134, 135, 141, 143–146
Node test, 23, 26, 51
Node-set function, 26
Non-selective index, 56
Normal form, 45
Normalization, 53
NP-complete, 105, 123
NP-hard, 105
Numbering schema, 65, 69

Oracle XML DB, 41

Parent, 10
Path expression, 47, 64, 150, 157
Path extraction function, 79
Path index, 73
Patricia Trie, 71
PCDATA, 15
PDOM, 42
PostgreSQL, 38
Predicate, 23, 26, 53
Predicate query, 28
procedural markup, 8
Product automaton, 90, 129, 133, 136
Pseudocode, 29, 30, 83
Pure-path index, 55
Pure-path query, 28

Qualifier, 78, 83, 150
Qualifier Node Path, 81
Query optimizer, 89, 91, 106, 112, 115, 117, 140
Query processing, 89
Query type, 157

Range query, 28
Ranking heuristics, 115
Raw path index, 71
RDBMS, 41–44, 58, 61, 75, 77, 103, 163, 166
Reference, 11
Refined Path, 56
Refined Path Index, 72
Regular expressions, 52
Relational Databases, 38
relative location path, 23
replication, 123
REQUIRED, 15
Return value, 83
root, 47

Sch01, 148
Schema for Object-oriented XML, 19
Schematron, 19
Search Structure, 82
selective, 56, 62, 124, 157
Selective Index, 56
Selectivity, 5, 56, 91
semistructured, 5, 157
SEQL, 67
Service oriented architectures, 5
SGML, 7
Sibling, 11
Side tables, 41
Single-Key index, 86, 145
Single-key index, 82
Single-key query, 28, 72
SOAP, 21, 38
SOX, 19
SQL, 40, 158
State, 131
statistic DataGuide, 111, 149, 159
Strong DataGuide, 58, 68, 92, 124
Structural index, 55
Structural query, 28, 67
Structural summary, 58, 61, 67, 68, 124
structure - centric, 68
Synchronization, 123
System RX, 44, 73
SystemRX, 159

T, 46
T-Index, 62
Tag, 7, 10
tailing predicates, 53
Tamino, 42, 148
Template, 62
Template Index, 62
Tex, 7
Timber, 42
Transition, 131
Transitive closure, 131
Tree Patterns, 151
Tree signature, 66
Tree-like, 5, 11, 20, 42, 45, 60
TreeMap, 72, 82, 101
Trie, 66
Troff, 7
Turing-complete, 29, 34, 35

UDDI, 21
UML, 150, 151
Uncapacitated Facility Location Problem, 105

valid, 14
Validity, 14

Value index, 55, 66, 67
Value node path, 79
Value query, 28, 67
Virtual Suffix Tree, 69
ViST, 69

W3C, 9, 16, 20, 22, 29, 31, 35
Web service, 5
Web services, 20
well-formed, 10, 14, 19, 20, 35, 38
where, 33
Wildcard, 28, 30, 52, 53, 58, 59, 64, 66, 71–73, 75, 87, 89, 90, 93, 101, 124, 157, 158
Wildcard query, 28
Workload, 63, 104, 149, 157
WSDL, 20

XALAN, 31
XDBMS, 42, 43, 103, 104, 147
XHTML, 19
Xindice, 42
XIUP, 123, 124, 151
XMark, 9, 10, 12, 14, 84, 85
XML, 9, 10, 45
XML data, 38
XML database management system, 147
XML document, 10, 38
XML editor, 18, 34, 166
XML Extender, 41
XML Index Update Problem, 123, 124
XML Schema, 16, 39, 103, 124
XML:DB, 43
XMLClob, 42
XMLSchema, 14, 16
XMLSpy, 18, 34
XOBE, 19
XPath, 5, 22, 29, 45, 47, 148, 157, 160
XPath axes, 24
XPath evaluation, 29
XPath examples, 27
XPath expression, 23, 73
XQuery, 29, 160
XQuery sequences, 31
XSLT, 29
XUpdate, 35, 148