Dissertation zur Erlangung des akademischen Grades
Dr. rer. soc. oec.
im Doktoratsstudium der Sozial- und Wirtschaftswissenschaften

Flexible and Selective Indexing in XML Databases

Angefertigt am

Institut für Wirtschaftsinformatik -
Data & Knowledge Engineering
Johannes Kepler Universität Linz

Eingereicht von

Mag.ª Katharina Grün

Betreut von

o. Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl
a. Univ.-Prof. Mag. Dr. Werner Retschitzegger

Linz, August 2008

Johannes Kepler Universität Linz, Altenberger Straße 69, 4040 Linz, Austria

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen deutlich als solche kenntlich gemacht habe.

Linz, August 2008

(Katharina Grün)

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich beim Erstellen dieser Dissertation unterstützten. Allen voran möchte ich meinem Betreuer Michael Schrefl für die wertvollen Ratschläge und Ideen danken, welche in unseren Diskussionen entstanden sind. Michael Schrefl bemühte sich ständig, mein wissenschaftliches Interesse zu fördern, und gewährte stets die notwendige Freiheit bei der Erfüllung wissenschaftlicher Aufgaben. Ebenfalls danke ich Werner Retschitzegger, dass er die Rolle als Zweitgutachter übernommen hat.

Großer Dank gebührt meinen InstitutskollegInnen Margit Brandl, Michael Karlinger, Bernd Neumayr, Christian Eichinger, Stefan Berger sowie Georg Nitsche. Sie sorgten nicht nur für ein perfektes Arbeitsklima, sondern lieferten auch wichtige Unterstützung in zahlreichen Diskussionen. Weiters bedanke ich mich bei meinen DiplomandInnen Peter Lasinger, Anita Engleder, Walter Dorninger und Christoph Wöllinger, welche in ihrer Diplomarbeit Teile dieser Dissertation prototypisch implementierten.

Mein besonderer Dank richtet sich natürlich auch an meine Familie, die meinen universitären Werdegang ermöglichte und förderte, sowie allen Freunden, welche für den notwendigen Ausgleich sorgten.

Kurzfassung

XML ist eine Auszeichnungssprache, welche ursprünglich vor allem als Datenaustauschformat eingesetzt wurde. Auch als Datenmodell gewinnt XML zunehmend an Bedeutung, weshalb Datenbanken zur Speicherung und Abfrage von XML Dokumenten benötigt werden. Um effizient Abfragen durchführen zu können, müssen XML Datenbanken den Inhalt und die Struktur häufig abgefragter Dokumentfragmente indizieren. Bislang unterstützen XML Datenbanken keine flexiblen, selektiven Indexstrukturen, welche den speziellen Anforderungen des hierarchischen, semi-strukturierten Datenmodells von XML gerecht werden.

XML Datenbanken benötigen Indizes, welche Abfragen auf den Inhalt und die Struktur von XML Dokumenten unterstützen. Diese *Flexibilität* stellt folgende Herausforderungen: Wie können Indizes die hierarchische Dokumentstruktur darstellen und verarbeiten? Welche Indexstrukturen sind notwendig, um beliebige Abfragen auf den Dokumentinhalt und die Dokumentstruktur zu unterstützen? Um nicht Dokumente als Ganzes indizieren zu müssen, sollen XML Datenbanken Indizes auf häufig abgefragte Dokumentfragmente definieren können. Diese *Selektivität* wirft folgende Fragen auf: Wie kann ein Datenbankverwaltungssystem Indizes auf beliebige Dokumentfragmente verarbeiten? Wie kann es beliebige Indizes mit Änderungen am Datenbestand konsistent halten?

Diese Dissertation stellt einen neuen Indizierungsansatz namens SCIENS (Structure and Content Indexing with Extensible, Nestable Structures) vor. SCIENS repräsentiert und verarbeitet die Dokumentstruktur mit Hilfe von Labels, welche strukturelle Eigenschaften kodieren. Während bestehende XML Indizierungsansätze auf proprietären Indexstrukturen basieren, passt SCIENS existierende Indexstrukturen an das XML Datenmodell an. Durch Erweitern und Schachteln dieser Indexstrukturen stellt SCIENS Indizes für beliebige Abfragen zur Verfügung. Das Indexframework von SCIENS kann beliebige Indizes basierend auf einem Indexmodell verarbeiten. Um Indizes mit Dokumentänderungen konsistent zu halten, verwendet es einen neuen Algorithmus, welcher Indexänderungen aus den zu ändernden Dokumentfragmenten extrahiert.

Die Vorteile von SCIENS sind vielfältig. Flexibilität ermöglicht die Definition jener Indizes, welche die zu erwartenden Abfragen am besten unterstützen. Selektivität reduziert den Speicherbedarf und beschleunigt den Indexzugriff. Obwohl SCIENS lediglich auf einer kleinen Anzahl an Indexstrukturen basiert, kann es mehr Abfragen als bestehende Ansätze unterstützen. Das Indexframework garantiert, dass Indizes angelegt werden können, ohne Abfragen und Änderungen an Dokumenten zu beeinflussen. Der Algorithmus, welcher Indizes mit Dokumentänderungen konsistent hält, ist effizienter als bestehende Ansätze.

Abstract

XML, the eXtensible Markup Language, is becoming more and more popular not only as data exchange format on the Web but also as data format in database applications. The emerging trend towards XML applications creates the need for persistent storage of XML documents in databases. To efficiently query documents, XML databases require indices on the content and structure of frequently queried document fragments. Currently, XML databases still fail in offering flexible and selective indexing support for the specific requirements of the hierarchical, semi-structured XML data model.

Flexibility in indexing refers to supporting arbitrary queries on the content and/or structure of XML documents. Providing flexibility poses the following challenges: How can indices represent and process the hierarchical document structure? Which index structures are necessary to support arbitrary queries on the document content and structure? *Selectivity* refers to indexing frequently queried document fragments instead of entire documents. Providing selectivity raises the following research questions: How can a database management system process indices that refer to arbitrary document fragments? How can it keep arbitrary indices consistent with updates on documents?

The indexing approach SCIENS (Structure and Content Indexing with Extensible, Nestable Structures), which is presented in this thesis, provides flexible and selective indexing for XML databases. To represent and process the document structure, SCIENS uses a labeling scheme that encodes structural relationships into labels. While existing XML indexing approaches propose proprietary index structures, SCIENS adapts existing index structures to the XML data model. By extending and nesting index structures, SCIENS provides indices for arbitrary query workloads. The index framework enables SCIENS to process arbitrary indices based on an index model. To keep indices consistent with document updates, the maintenance algorithm of SCIENS exploits the document fragments being updated to extract relevant index updates.

The advantages of SCIENS are manifold. Flexible indices enable the definition of those indices that best match the query workload. Selectivity reduces index size and accelerates index traversal. Compared to existing XML indexing approaches, SCIENS only requires a small number of existing index structures, but can support a wider range of queries. The index framework guarantees that querying and updating documents remains unaffected by specific indices used. By exploiting the structure of update fragments, the maintenance algorithm can process updates more efficiently than existing approaches.

Notations

	Document
D	document
N	nodes
F	edges between nodes
L	node names
V	node values
E	element nodes
A	attribute nodes
T	text nodes
P	rooted paths

	Schema
$\mathbf{S}$	schema
$\mathbf{N}$	node schemas
$\mathbf{F}$	edges between node schemas and types
$\mathbf{L}$	node schema names
$\mathbf{E}$	element node schemas
$\mathbf{A}$	attribute node schemas
$\mathbf{T}$	text node schemas
$\mathbf{P}$	rooted path schemas

	Types
$\mathcal{T}$	types
$\mathcal{T}_\mathcal{C}$	complex types
$\mathcal{T}_\mathcal{S}$	simple types
$\mathcal{L}$	type names
$\mathcal{H}$	type hierarchy
$\mathcal{H}^+$	transitive closure of type hierarchy
$\mathcal{H}^*$	reflexive, transitive closure of type hierarchy

	Labels
Z	instance labels
$\mathbf{Z}$	path schema labels
$\mathcal{Z}$	type labels

	Index
I	index
E	index entries
K	index keys
V	index variables
T	index structures
O	search conditions
C_I	index configuration
C_S	search configuration

	Index Pattern
P	index pattern
N	pattern nodes
F	edges between pattern nodes
L	pattern node names
S	stacks

	Operators & Symbols
$\sim$	contains word
$\dot\sim$	contains word prefix
$\prec$	precedes
$\models$	instance of
$\vdash$	parent
$\Vdash$	ancestor
$\dashv$	child
$\dashv\!\mid$	descendant
$\perp$	null

Contents

Chapter 1

Introduction

Contents

This chapter gives a general introduction to the topics of this thesis, starting with the motivation in Section 1.1. Section 1.2 describes its main challenges and Section 1.3 reviews related work. Based on the objectives of Section 1.4, Section 1.5 introduces the approach taken in this thesis and its main contributions. Finally, Section 1.6 outlines the overall structure of this thesis.

1.1 Motivation

XML, the eXtensible Markup Language [38], is becoming more and more popular not only as data exchange format on the Web but also as data format in database applications. The emerging trend towards XML applications results in an increase in the document size and in the number of documents to be processed, creating the need for persistent storage of XML data in XML database management systems (DBMS). In contrast to storing XML documents in files and processing queries in main memory, a DBMS provides amongst others specific storage and index structures, efficient query processing techniques, transactions, concurrency control and recovery[1].

XML is a hierarchical, semi-structured data format. XML documents contain structured data (data-centric XML) and unstructured data (document-centric XML). While structured data has been investigated by relational databases, unstructured data is the focus of information retrieval techniques. Many applications need to handle both structured and unstructured data (e.g. book catalogs, product specifications), generating the demand for new techniques to efficiently store, query and update semi-structured data.

Currently, there exist two approaches for storing and accessing XML documents in databases: XML-enabled and native XML databases [47]. To cope with the popularity of XML, many database vendors offer XML extensions for their databases (cf. [36]). These extensions, which are typical for relational databases, map XML documents to relations and XML queries to SQL, or they store XML as text and provide proprietary XQuery extensions. As they apply relational storage, query and indexing techniques to XML, they fail in supporting the specific requirements of processing hierarchical, semi-structured data. Native XML databases are based on the XML data model and integrate specific techniques for storing and accessing XML documents. These techniques include, for example, new storage and index structures and join algorithms.

Databases typically distinguish the primary index from secondary indices [83]. The primary index defines how to store entire documents and physically organizes documents. To evaluate search conditions without traversing entire documents, secondary indices provide additional search structures. They map certain properties of documents, such as values, names or types, to the corresponding document fragments in the primary index using physical or logical addresses. By accessing secondary indices, a database can retrieve the fragments that match indexed properties without scanning the entire primary index. Secondary indices are therefore indispensable to efficiently process arbitrary queries in databases.

In contrast to the primary index, secondary indices are created on demand in consideration of the query workload. Important requirements for secondary indices include (i) *flexibility* to handle arbitrary queries and (ii) *selectivity* to index frequently queried document fragments instead of entire documents only[2].

[1]In literature, the term database system usually refers to both the database, i.e. the collection of data, and the database management system, which is the software to manage databases. In this thesis, we use the term database as a short form for database system.

[2]In our terminology, selectivity does not refer to the number of distinct values in an index (e.g. gender vs. social security number), but to the portion of the document being indexed.

While flexibility enables databases to adapt indices to arbitrary query workloads, selectivity reduces the index size, which accelerates index traversal. Secondary indices are well-established in relational databases. XML databases, however, still fail in offering flexible and selective indexing support for the specific requirements of hierarchical, semi-structured data.

1.2 Challenges

XML is an extensible markup language that hierarchically structures document content with the help of tag names. The hierarchical document structure logically consists of an ordered tree of nodes. Each node has a name and optionally a type. The document content is logically represented in the leaf nodes that have associated values. The path of a node is the sequence of nodes from the root of the tree to the node. The labelpath consists of the node names on a node's path. The document structure may be defined in a schema document, such as a DTD [38] or an XML Schema [187].

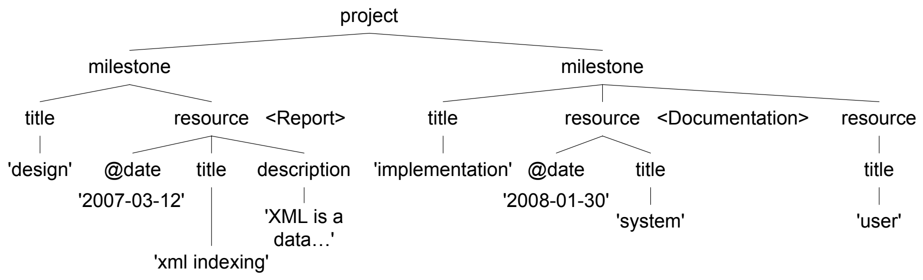


Figure 1.1: XML document representing project resources.

Example 1.1 (XML document) *Figure 1.1 shows a sample XML document comprising information about project resources grouped into milestones. Each resource has a title and optionally a date and a description. The nodes, which are represented by their tag names (e.g.* `title`*), form a hierarchical structure. The content of the document is made up of values, which are represented as leaf nodes in the figure (e.g. '*`xml indexing`*'). In the example, resource nodes can have an explicit type, which is either* `<Report>` *or* `<Documentation>`*. A sample path is* `/project [1]/milestone [2]/resource [1]`*, which is the path from the root of the tree to the first resource of the second milestone. The corresponding labelpath is* `/project/milestone/resource`*.*

The XML data model clearly differs from other data models, such as the relational, object-oriented or object-relational data model. Queries on XML documents typically constrain both the document structure (names, paths, labelpaths, types) and the document content (values).

Example 1.2 (queries) *Regarding Figure 1.1, a sample query workload may comprise the following queries: (i) retrieve all resources with title '*`xml indexing`*'; (ii) return all resources with type* `<Report>` *that have been written in the year '2007'; (iii) retrieve the titles of resources within the milestone '*`design`*' that contain the word '*`xml`*' in their description.*

To support queries on XML documents, secondary indices need to provide the *flexibility* to index structure- and content-oriented properties and to support various operations on these properties. Relational databases index values and object-oriented databases provide simple indices on paths and types. XML databases require indices on names, paths, labelpaths, types and values of nodes as well as indices on arbitrary combinations of these properties.

Example 1.3 (flexibility) *Evaluating the first query of Example 1.2 on the document of Figure 1.1 may require scanning the titles of all resources. This query can be accelerated with a secondary index (e.g. a hash table) on the values of titles. The second query can be supported with an index on resource types and a value index on resource dates. To avoid the need for accessing each index and joining index results, it would also be possible to create an index on both the resource types and date values. A full-text index on resource descriptions accelerates the third query. If the index groups resource descriptions according to their milestone, the database need not traverse the entire index, but only the part of the index that addresses the requested milestone.*

To support arbitrary queries, a database requires indices on the document structure and content. Each index builds a data structure on indexed properties and uses algorithms to traverse and update the data structure. For this purpose, it requires a representation of properties that allows for comparing indexed properties. A database further requires a representation of nodes returned by indices that allows for subsequent query processing, e.g. for joining index results. Example 1.3 shows that each index more efficiently supports different kinds of queries. However, a database cannot provide one index structure for each possible kind of query. It therefore needs to adapt index structures to handle various properties of the XML data model. Offering flexibility poses the following challenges:

- How can an index represent indexed properties as well as nodes to be returned by the index?
- Which index structures are necessary to index structure- and/or content-oriented properties?

To efficiently support various queries, a database requires several secondary indices. Defining indices on entire documents does not only slow down index traversal, but also increases required storage space. Databases should therefore provide *selectivity* in indexing, denoting the ability to define indices on relevant document fragments instead of on entire documents only. For example, a value index should not need to index all node values, but only the values of nodes with the requested labelpath or name. Selective indices are smaller in size, which accelerates index traversal.

Example 1.4 (selectivity) *Considering Examples 1.2 and 1.3, the index that supports the query on resource titles should not index all values of the document, but only the values of resource titles. Such a selective index is smaller in size and supports queries on resource titles more efficiently. However, the database cannot use the index to process every query on values. While the index can return all resources with a certain title, it cannot select resources by their date.*

Similarly, when updating a node value, the database needs to decide whether the update affects the index. When adding a new resource, for example, the index need not be updated with each node value of the new resource, but only with the value of the new title.

As each selective index refers to different document fragments, selectivity entails that (i) each index only supports the queries that refer to the indexed fragments, and (ii) an index only needs to be updated when updates on the document affect indexed fragments. Selectivity poses the following challenges:

- How can a database define, select and process indices that refer to arbitrary document fragments?
- How can indices that are defined on arbitrary document fragments be maintained when updating documents?

1.3 State of the Art

Indexing plays an important role in any database to retrieve requested data without scanning all data. The most well-known index structures have been developed for relational databases, including hash tables and B-trees [83] as well as multidimensional index structures [82]. Object-oriented index structures [28] extend these index structures for aggregation graphs and inheritance hierarchies. Information retrieval (IR) [20] uses inverted lists, signature files and suffix arrays. The Generalized Search Tree (GiST) [101] is an extensible index structure that can handle different data types and operations.

To conform to the specific requirements of the XML data model, various XML index structures have been proposed recently [43]. Most of them rely on a labeling scheme that assigns a unique label to each node of a document (e.g. [8, 39, 57]). Labels do not purely serve as node identifiers, but also capture structural relationships between nodes. They enable the processing of nodes returned by indices without accessing documents, e.g. to perform structural joins between nodes. XML indices can primarily be classified into structure- and content-oriented indices.

Structure-oriented approaches first divide the search space according to the document structure and reflect the document hierarchy in the index structure (e.g. [59, 85]). While they favor queries on the document structure, content-oriented indices (e.g. [70]) group nodes with identical node values to efficiently retrieve nodes with a specific value. A tradeoff between structure- and content-oriented indices are those approaches that either build indices on node names and values and perform structural joins to restore the node hierarchy (e.g. [135]) or use multidimensional search structures (e.g. [202]). To answer frequent queries more efficiently, some approaches dynamically adapt the index structure to the query workload (e.g. Apex [56]). However, the possibilities for adaption are limited and only few queries are well supported.

Current XML databases [47] offer limited support for secondary indices. They provide simple structural and value indices, but cannot adapt their index structures to support more complex queries. Some object-relational databases enable the extension of the database with new index structures. The Generalized Search Tree (GiST) [101] is an extensible index structure that can handle different data types and operations. Oracle data cartridges [184], DB2 database extenders [64] and Informix data blades [32] provide uniform index interfaces that enable the integration of new index structures into the database.

A database that provides flexible and selective indices needs to be able to process various indices. Several join-based, navigational and hybrid XPath processing techniques have been proposed that optimize queries based on DTDs, cost models and/or heuristics [93, 122, 123, 146, 205]. They focus on specific evaluation approaches, e.g. on whether to execute a query by navigating the document structure or by accessing indices.

To process arbitrary indices in XML, a database requires a more generic approach. In literature, two approaches are worth mentioning: XML Access Modules (XAMs) [17] and KeyX [94]. XAMs are generic descriptions of what is stored in a storage or index structure. The primary purpose of this approach is to achieve physical data independence by selecting XAMs for queries without requiring knowledge of underlying storage structures. KeyX describes algorithms to select, maintain and suggest indices based on a more limited index model than XAMs. The maintenance algorithm of KeyX has rather poor update performance as it processes each update node individually.

Current XML index structures are better suited for different kinds of queries and index entire documents. They therefore do not provide the flexibility and selectivity required by secondary XML indices. While the XAM approach is appropriate for selecting arbitrary indices for queries, an efficient algorithm to maintain arbitrary indices is still missing.

1.4 Objectives

The primary focus of this thesis is to provide flexible and selective indices for XML databases. To handle the challenges presented in Section 1.2, the thesis pursues the following objectives:

- *Labeling scheme:* To represent indexed properties and nodes in an index, define a labeling scheme that assigns unique labels to structural properties, such as paths, labelpaths and types. To compare indexed properties when traversing or updating an index, labels must support evaluating structural relationships and keep document order. They further should support updates without relabeling and provide an efficient encoding.

- *Index structures:* To index arbitrary properties, select index structures that support typical queries on both the structure and the content of XML documents. As there already exists a large number of index structures each of which can accelerate different kinds of queries, the objective is not to develop new index structures. Instead, the focus is on extending and combining existing index structures to adapt them to the requirements of the XML data model.

- *Index framework:* Provide an index framework that allows for processing indices defined on arbitrary properties and document fragments. Index processing requires an index model that can represent arbitrary indices. Based on this index model, the framework requires algorithms to select[3] and access indices when processing queries and to maintain indices when updating documents.

- *Index maintenance:* Define an index maintenance algorithm to keep secondary indices consistent with updates on documents. The algorithm should handle arbitrary indices regardless of what they index and of which specific index structures they use. Further, it should be efficient in the sense that it should require the minimum possible number of accesses to documents to determine necessary updates.

This thesis focuses on providing flexible and selective indexing, but not on query optimization with indices. To select appropriate indices for queries, the thesis adapts the XAM approach [17]. A cost model to find the optimal set of indices for a query during query optimization as well as an algorithm to suggest the optimal set of indices for a query workload during database design are out of scope of this thesis.

1.5 Approach

This thesis has emerged from a research project called SemCrypt [176][4], which is a secure, native XML database. To query and update outsourced, encrypted XML documents, integral parts of SemCrypt are a schema-aware labeling scheme and secondary indices. The overall steps to achieve the objectives presented in the previous section followed the design-science research paradigm (cf. [103, 189]). After becoming aware that existing XML indexing approaches do not offer the necessary flexibility and selectivity, we suggested a new approach that extends, combines and improves existing approaches. All concepts have been implemented and integrated into the XML database SemCrypt and tested analytically through performance studies.

In the following, we outline and exemplify the approach of this thesis and its contributions. Details will be given in the subsequent chapters.

Labeling scheme: The labeling scheme assigns a unique label to each node of a document as well as to the labelpaths and types of a document. It does not only encode structural relationships into node labels, but also integrates schema information to improve query and update processing. Labels enable the representation and comparison of indexed properties as well as the representation and processing of nodes returned by indices.

Example 1.5 (labeling scheme) *Figure 1.2 assigns a label to each node of a sample document fragment, which is depicted to the right of each node. The first part of the label identifies the labelpath, e.g. the labelpath of each resource*

[3]The term index selection is sometimes used to denote the process of suggesting possible indices for query workloads during database design, which is out of scope of this paper.

[4]SemCrypt is a secure, native XML database that has been developed at the Department of Business Informatics - Data & Knowledge Engineering at Johannes Kepler University Linz with partners Electronic Competence Center Vienna and EC3 Networks GmbH. The research project has been funded by FIT-IT Semantic Systems under grant 809262/9315-KA/HN.

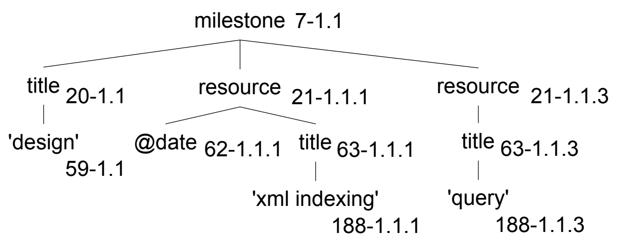


Figure 1.2: Document fragment with labels.

is identified by 21*. The second part of the label identifies the position of a node in the document and enables structural comparisons. E.g. the* `milestone` *with label* 7–1.1 *is the parent of the* `resource` *with label* 21–1.1.1*.*

Index structures: To support flexibility, we select a small set of index structures and describe how to extend them to both structure- and content-oriented properties. We further propose the concept of index nesting to adapt these index structures to specific query workloads on the hierarchies of XML documents. We refer to this indexing approach as SCIENS (Structure and Content Indexing with Extensible, Nestable Structures).

query	21-1.1.3, 21-5.3.7,...
XML indexing	21-1.1.1
...	...

63	63-1.1.1, 63-1.1.3, 63-1.3.1,...
20	20-1.1, 20-1.3, 20-1.5,...
...	...

Figure 1.3: Hash table on title values (left) and hash table on title labelpaths (right).

Example 1.6 (index structures) *To retrieve all resources with a specific title (cf. Example 1.2), we can define a hash table on the values of resource titles. We can also define an index that returns titles with the same labelpath. Figure 1.3 depicts parts of the corresponding hash tables. To keep the figure simple, we assume that the hash table is collision-free. The left hash table maps each distinct title value to the labels of its resource nodes, whereas the right hash table associates each distinct labelpath of title nodes with its node labels. This example demonstrates that the hash table can easily be used to index values or labelpaths. To support more complex indices, we propose to extend range and multidimensional indices by adapting their comparison operators to structural properties in this thesis.*

Index framework: We present a framework to process arbitrary indices independently of their physical data structures. The framework enables the database to select indices during query optimization, to access indices during query execution and to maintain indices when updating documents. The basis of the index framework is an index model that defines the properties and the nodes on which an index is defined.

Example 1.7 (index model) *Two sample index definitions are depicted in Figure 1.4. The left index indexes the values of resource titles* ($= p_v$)*. The*

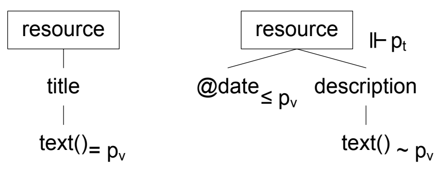


Figure 1.4: Index model for defining an index on resource titles (left) and an index on types, dates and descriptions of resources (right).

border around the resource node indicates that the index maps each distinct title value to the label of its resource. The right index definition is more complex. It defines a multidimensional index on types of resources and values of resource dates and descriptions. This index supports different operations on indexed properties, i.e. hierarchical queries on type hierarchies ($\Vdash p_t$), range queries on dates ($\leq p_v$) and full-text queries on descriptions ($\sim p_v$).

Index maintenance: We propose a generic index maintenance algorithm that can update arbitrary index structures defined on arbitrary document fragments. By exploiting the structure of fragments that are inserted into or deleted from an XML document, the proposed algorithm requires a minimum number of queries to extract relevant updates and to propagate them to affected index structures.

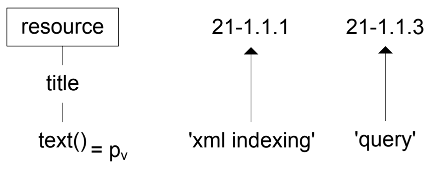


Figure 1.5: Updating an index on resource titles with the document fragment of Figure 1.2.

Example 1.8 (index maintenance) *Assume that we insert the document of Figure 1.2 and update the sample index on resource titles as depicted in Figure 1.5. By comparing the index definition with the update fragment, the maintenance algorithm extracts indexed properties and nodes to be returned by the index. It determines that value '`xml indexing`' maps to the node with label* 21–1.1.1 *and value '`query`' to label* 21–1.1.3 *according to the index definition. In this case, the algorithm can extract all updates from the update fragment. However, if we modify the title of the first resource, the update only affects the title, but does not contain the resource label. To update the index, the maintenance algorithm has to determine the resource label before modifying the title in the index.*

The proposed concepts allow for defining those indices that best match the query workload. Indexing frequently queried fragments instead of entire documents accelerates index traversal and reduces required storage space. The developed index processing techniques handle arbitrary indices that can be expressed by the index model and guarantee that querying and updating documents remains unaffected by specific indices used.

1.6 Outline

The remainder of this thesis is organized into three parts. Part I addresses labeling and indexing XML documents. Part II presents an index framework and a maintenance algorithm to process secondary indices. Part III evaluates the concepts of this thesis by describing their integration into the native XML database SemCrypt as well as performance studies.

Chapter 2 - Preliminaries defines the data and schema model used in this thesis. Based on typical queries on XML documents, it then derives requirements for secondary indices in XML databases.

Chapter 3 - Labeling Scheme proposes a labeling scheme that assigns labels to nodes of documents as well as to labelpaths and types defined in schemas. The labeling scheme bases on existing dynamic labeling schemes and their encodings. The chapter also describes how to process labels to support the required indexing operations.

Chapter 4 - Index Structures describes the indexing approach sciens. After reviewing related work on index structures, the chapter presents the main concepts of sciens. To index structure- and/or content-oriented properties, the chapter shows how to extend and nest a hash table, a B+-tree and a KDB-tree and how to adapt their comparison operators to structural properties. Finally, the chapter contrasts various indexing alternatives and compares them to existing indexing approaches.

Chapter 5 - Index Framework presents an index framework to select, access and maintain arbitrary indices. The framework bases on an index model that represents index definitions as tree patterns. To process arbitrary indices based on this index model, the chapter describes how to select indices using the XAM approach [16].

Chapter 6 - Index Maintenance describes an index maintenance algorithm to propagate relevant updates to affected index structures when updating documents. After outlining shortcomings of existing approaches, the chapter presents the main concepts of the proposed approach and its algorithms.

Chapter 7 - Case Study: The XML Database SemCrypt presents SemCrypt, which is a native, secure XML database in whose context the concepts of this thesis have been developed and implemented. After giving a general introduction to SemCrypt, the chapter describes its architecture and main concepts. Thereby, it uses a case study to show how to process documents, schemas, queries and indices in SemCrypt.

Chapter 8 - Performance Studies evaluates the performance of the indexing approach sciens. It includes performance studies on the index structures and the maintenance algorithm that show the efficiency of the developed approach and its improvements compared to existing approaches.

Chapter 9 - Conclusion summarizes the concepts of flexible and selective indexing in XML databases and gives an outlook on possible future work.

Part I

Labeling and Indexing XML Documents

Chapter 2

Preliminaries

Contents

This chapter introduces documents and schemas as well as queries and indices on documents. Section 2.1 first describes documents and schemas in general and then defines the data and schema model used in this thesis. Based on typical queries on documents presented in Section 2.2, Section 2.3 derives requirements for indices in XML databases.

2.1 XML Documents and Schemas

XML, the eXtensible Markup Language [38], is a recommendation of the World Wide Web Consortium (W3C) with the primary purpose of facilitating data exchange on the Web. While HTML [163] uses tags to specify how to format data, XML tags do not describe how to present data but the meaning of data. XML is extensible as it is possible to define custom tags that describe the data enclosed by them. Further, XML documents are self-describing as the tags define the document structure. While tags enclose so-called elements and structure data hierarchically, attributes further describe elements.

```
<project>
  <resource date='2008-03-12'>
    <title>XML</title>
    <description>
      XML is a language for data representation that has been
      designed by the W3C. This resource describes storage,
      query and indexing techniques for XML.
    </description>
  </resource>
</project>
```

Figure 2.1: XML document.

Example 2.1 *Figure 2.1 shows a sample XML document describing a project resource. The tag `<title>` defines that 'XML' is a title. As the tag `<title>` is enclosed by the tag `<resource>`, its value 'XML' is the title of a resource. The resource has a date, which is modeled as an attribute of the resource element.*

While XML has primarily been designed as data exchange format on the Web, it has become increasingly popular for data modeling, information integration and document representation. Dependent on what XML is used for, there exist various types of XML documents. *Data-centric* documents contain structured data, such as financial data, while *document-centric* XML refers to unstructured data as in books. Structured data is typical for relational databases, whereas information retrieval techniques focus on unstructured data. In many application domains, data is neither completely structured nor unstructured. This kind of data is referred to as *semi-structured* data. Examples for semi-structured data are emails, contracts and health records.

Example 2.2 *The XML document in Figure 2.1 can be regarded as semi-structured. It contains structured parts, such as the date and title of a resource. The description is unstructured, entailing that the relationship between 'XML' and the 'W3C' is not explicitly marked, but only described textually.*

The main characteristics of semi-structured data are [6]:

- The structure is irregular and can vary among and within documents, i.e. data items may add or miss information.

- Documents are self-describing and do not depend on a schema.
- The structure is partial, i.e. documents can contain unstructured parts.
- The structure of documents may change in the course of time.

Example 2.3 *When adding a second resource to the XML document of Figure 2.1, this resource need not have the same structure as the existing resource. It may miss the date or additionally contain information about the editor of the resource.*

XML shares the characteristics of semi-structured data and additionally imposes the following restrictions:

- XML documents have a hierarchical structure.
- The order among elements is important.

XML documents do not depend on a schema, but there exist schema languages for defining the structure of XML documents. The most popular ones are DTDs [38] and XML Schemas [187] of the W3C. Schemas describe which elements and attributes can be contained in a document, how they are structured, how often they can occur and which data types are allowed for values. A document which conforms to a schema is said to be valid. Schemas are especially useful for data-centric applications and many public vocabularies have been defined to enhance interoperability, e.g. for e-government, healthcare and human resources [84].

Although XML documents do not depend on a schema, it is reasonable to extract schema information and use it for optimizing and accelerating query and update processing and for defining storage and index structures [7, 190]. The indexing approach proposed in this thesis exploits the schema of XML documents to facilitate index processing. As the W3C recommendations for representing documents and schemas are very complex, we propose a simplified data and schema model in the following, which capture all information necessary for processing indices in XML databases.

2.1.1 Data Model

Each database requires an underlying data model, defining the type of data that can be stored and operations on it. As XML has not been designed as data model, there exist various data models for XML. The object exchange model (OEM) [161] is a data model for semi-structured data, but compared to XML, it has the form of a graph and is not ordered. The XML Information Set [60] defines XML with a set of information items, while the Document Object Model (DOM) [104] describes the logical structure of documents and an application programming interface to access and manipulate documents. In contrast to these data models, the XQuery 1.0 and XPath 2.0 data model [76] of the W3C additionally supports types.

The data model used in this thesis is based on the XQuery 1.0 and XPath 2.0 data model and defines an XML document as a tree of element, attribute and text nodes with one element node forming the root of the tree. Element nodes can contain nested nodes, attribute and text nodes have a value. Each node has a name and may have a type. The name is a qualified name, consisting of an optional namespace URI and a local name. The type of a node is either implicitly defined by the value of the node, or it can be assigned to element nodes via the `type` attribute of the XML Schema specification [187]. In contrast to the XQuery 1.0 and XPath 2.0 data model, our data model does not consider document, namespace, processing instruction and comment nodes. To index these node kinds, we could adopt the same concepts as for element, attribute and text nodes.

Definition 2.1 (document) A *document* D is an ordered, directed tree with node names, values and types, $D = (N, F, name, value, type, root, L, V, \mathcal{T})$, where

- $N = E \cup A \cup T$ is a finite set of nodes classified into the following node kinds: element nodes E, attribute nodes A and text nodes T.
- $F \subseteq E \times N$ is a finite set of directed edges.
- Nodes have a name, represented by function $name : N \rightarrow L$, where L is a set of node names. The name of attribute nodes is preceded with the symbol `@` and text nodes are assigned the special name `text()`.
- Attribute and text nodes have associated a value, represented by function $value : A \cup T \rightarrow V$, where V is a set of values.
- Nodes may have a type, represented by partial function $type : N \rightarrow \mathcal{T}$, where $\mathcal{T}$ is a set of types. Each type is represented by a unique type name.
- Function $root : D \rightarrow N$ returns the root node of document D, which has no incoming edge.

□

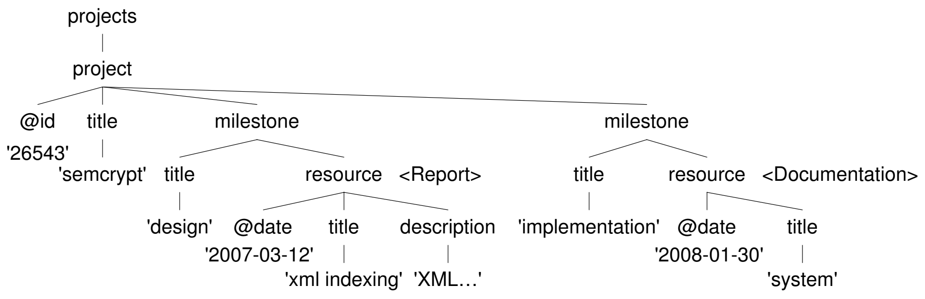


Figure 2.2: Document as a tree of nodes.

Example 2.4 (document) *Figure 2.2 depicts a sample XML document, showing the name of element and attribute nodes and the value of attribute and text nodes. The types of element nodes are written within tags `<>`. For example, there is an element node with name `resource` and type `<Report>`, an attribute node with name `@id` and value '26543' and a text node with value '`xml indexing`'. The node `projects` is the root of the document.*

Except for the root node, each node has a parent, $parent : N \to E$, where $parent(n) = e$ iff $(e, n) \in F$. Element nodes have children, defined by $children : E \to 2^N$, where $children(e) = \{n \in N \mid parent(n) = e\}$. The *ancestor* relation is the transitive closure of the parent relation, whereas the *descendant* relation is its inverse. Note that this definition simplifies the XQuery 1.0 and XPath 2.0 data model, which ignores attribute nodes in the children and descendant relations. Nodes with the same parent are referred to as *siblings*. A node, which is not a leaf of the tree, is referred to as internal node. Attribute and text nodes are always leaves.

Definition 2.2 (document order) The document order among all nodes $n, n' \in N$ of a document is defined as follows. Node n precedes node n', denoted by $n \prec n'$, if n is visited before n' in a preorder traversal of the document tree. Attribute nodes immediately follow the element nodes with which they are associated.

Pursuant to the XQuery 1.0 and XPath 2.0 data model, each node has a unique identity. Attribute nodes with the same parent must have distinct names. The content of an element node corresponds to the concatenation of the values of all its text node descendants in document order. If an element node has both element and text nodes as children, it is said to have mixed content. The children of an element node must not contain two consecutive text nodes.

To reduce complexity, we disallow mixed content. We further define that the descendants of a node must not have the same name as the node, which impedes recursion. However, we describe necessary extensions to support mixed content and recursion in the chapters that are affected by these extensions.

Edges connect nodes and define paths to traverse the XML tree. There is one path from the root of the document to each node.

Definition 2.3 (path) Let D be a document according to Definition 2.1 with the set of nodes N.

- A *rooted path* p is a sequence of k connected nodes $\langle /n_1/ \ldots /n_k \rangle$, where $n_1 = root(D) \wedge parent(n_{i+1}) = n_i$ for $n_i \in N$ and $i = 1 \ldots k-1$.
- Let P be the set of rooted paths in D.
- Function $last : P \to N$ returns the last node of a path $p \in P$, $last(\langle /n_1/ \ldots /n_k \rangle) = n_k$ for $k \geq 1$.
- The *access path*, or shortly *path*, of a node $n \in N$ is defined by function $path : N \to P$, such that $path(n)$ is the path $p \in P$ where $last(p) = n$.
- The *labelpath* of a node corresponds to the sequence of node names on its path.

□

Example 2.5 (path) *In the document of Figure 2.2, a sample path is* `/projects [1]/project [1]/milestone [2]/ title [1]/ text()`*, whereby the numbers in parenthesis denote the position of the corresponding nodes within siblings with the same name. This path is the access path of the text node with value '*`implementation`*'. The corresponding labelpath is* `/projects/project/ milestone/ title/ text()`*.*

2.1.2 Schema Model

While the primary purpose of schema languages, such as DTDs [38] and XML Schema [187] of the W3C, is to provide a grammar for validating documents, databases use schema information to efficiently store, index and query documents. Although XML databases should be able to handle schemaless documents, integrating structural information about documents accelerates and facilitates query and update processing.

Previous schemas considered for XML databases are mainly structural summaries, such as DataGuides [85]. They comprise labelpaths to summarize the tag hierarchy of documents. Ludäscher et al. [143] argue that schemas also need to incorporate information about types to support queries on the conceptual level instead of on the syntactic level only. In our context, a schema is a structural summary that integrates types and type hierarchies. It can be constructed from a DTD or an XML Schema or it can directly be extracted from XML documents, similarly to DataGuides. Our schema model is based on XML Schema, but simplifies it by omitting all constructs that only serve for validation and by adapting it to our data model. It basically contains only information present in XML documents to be independent of specific schema languages. The only exception are optional type hierarchies, which can be defined in XML Schema, but not directly in documents.

A schema is a graph of node schemas and types. Node schemas define the kind and name of nodes in documents. Analogous to nodes in the document, there are element, attribute and text node schemas. The cardinality of a node schema determines how often a node with a certain kind, name and parent can appear in a document. Each node schema has a type, which is either simple or complex. Simple types specify allowed values of attribute and text nodes, whereas complex types comprise node schemas to define which children an element node may have. Various node schemas may refer to the same type. Complex types can extend complex types, forming a tree hierarchy of types.

Definition 2.4 (schema) A *schema* $\mathbf{S}$ is an ordered, acyclic, connected, directed graph, $\mathbf{S} = (\mathbf{N}, \mathcal{T}, \mathbf{F}, type, defines, extends, name, card, root, \mathbf{L}, \mathcal{L})$, where

- $\mathbf{N}, \mathcal{T}$ represent vertices.
- Functions *type*, $defines$ and *extends* define edges between node schemas and types, $\mathbf{F} \subseteq \mathbf{N} \times \mathcal{T} \cup \mathcal{T} \times \mathbf{N} \cup \mathcal{T} \times \mathcal{T}$.
- $\mathbf{N} = \mathbf{E} \cup \mathbf{A} \cup \mathbf{T}$ is a finite set of node schemas classified into element node schemas $\mathbf{E}$, attribute node schemas $\mathbf{A}$ and text node schemas $\mathbf{T}$.
- $\mathcal{T} = \mathcal{T}_{\mathcal{S}} \cup \mathcal{T}_{\mathcal{C}}$ is a finite set of types classified into simple types $\mathcal{T}_{\mathcal{S}}$ and complex types $\mathcal{T}_{\mathcal{C}}$.
- Function *type* assigns a type to each node schema, $type : \mathbf{N} \to \mathcal{T}$. More precisely, attribute and text node schemas have a simple type, $type : \mathbf{A} \cup \mathbf{T} \to \mathcal{T}_{\mathcal{S}}$, whereas element node schemas have a complex type, $type : \mathbf{E} \to \mathcal{T}_{\mathcal{C}}$.

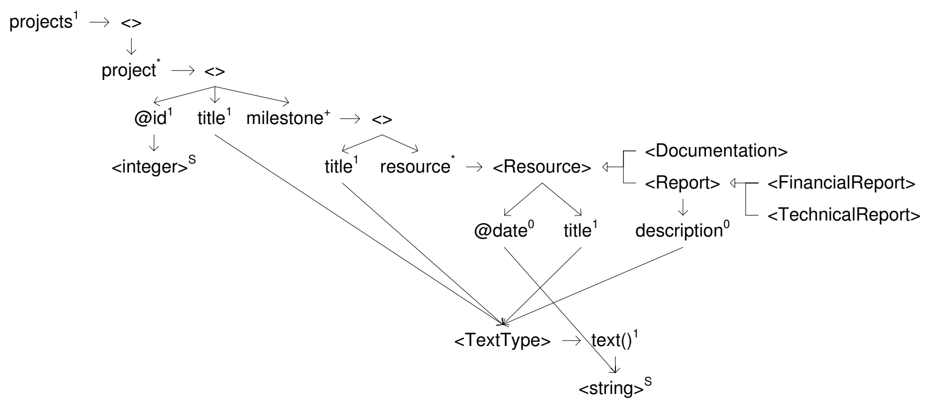
- Function $defines$ assigns node schemas to complex types, $defines : \mathcal{T_C} \rightarrow 2^{\mathbf{N}}$.
- Complex types may be derived from complex types, forming a tree of types, represented by function $extends : \mathcal{T_C} \rightarrow \mathcal{T_C}$. The tree hierarchy $\mathcal{H}$ represents all subtype-supertype relationships, $\mathcal{H} \subseteq \mathcal{T_C} \times \mathcal{T_C}$ such that $(t, t') \in \mathcal{H}$ iff $extends(t) = t'$. A complex type that has subtypes can be declared as abstract. An abstract type cannot be instantiated in a document (cf. Definition 2.7).
- Node schemas have a name, $name : \mathbf{N} \rightarrow \mathbf{L}$, where $\mathbf{L}$ is a set of node schema names. The name of attribute node schemas is preceded with the symbol `@` and text node schemas are assigned the special name `text()`.
- A type either has a name or is anonymous, as defined by partial function $name : \mathcal{T} \rightarrow \mathcal{L}$, where $\mathcal{L}$ is a set of type names. The type name uniquely identifies a type. Anonymous types cannot have subtypes.
- Each node schema has a cardinality, represented by function $card : \mathbf{N} \rightarrow \{0, 1, *, +\}$. A node schema is called single-valued if $card(\mathbf{n}) \in \{0, 1\}$, otherwise it is referred to as multi-valued. Attribute node schemas are always single-valued. If $card(\mathbf{n}) \in \{0, *\}$, the node schema is referred to as optional, otherwise it is required. Text node schemas are single-valued and required to disallow mixed content.
- Function $root : \mathbf{S} \rightarrow \mathbf{N}$ returns the root node schema of schema $\mathbf{S}$, which has no incoming edge.

□

Example 2.6 (schema) *Figure 2.3 depicts a sample schema with the root node schema* `projects`*. Each node schema has a name, the superscript indicates the cardinality of the node schema. The node schema* `milestone` *references an anonymous type and the node schema* `resource` *has type* `<Resource>`*. This type has two subtypes,* `<Documentation>` *and* `<Report>`*, which itself has two subtypes. The subtype* `<Report>` *defines that reports additionally have a* `description`*. There are two simple types, namely* `<integer>` *and* `<string>`*.*

Except for the root node schema, each node schema is defined in a complex type. We define function $definedAt : \mathbf{N} \rightarrow \mathcal{T_C}$ as the inverse of function $defines$. An extended type does not only define new node schemas, but also inherits the node schemas of its supertypes. Function $inherits : \mathcal{T_C} \rightarrow 2^{\mathbf{N}}$ comprises the inherited node schemas of a type, where $inherits(t) = \{\mathbf{n} \in \mathbf{N} \mid \exists (t, t') \in \mathcal{H}^+ : \mathbf{n} \in defines(t')\}$. The node schemas which a type defines and inherits must have distinct names.

Two node schemas are connected via a type if the type of one node schema defines the second node schema or if the second node schema is defined in a subtype of the first node schema. We express this with function $parent : \mathbf{N} \rightarrow 2^{\mathbf{N}}$, where $parent(\mathbf{n_2}) = \{\mathbf{n_1} \in \mathbf{N} \mid (definedAt(\mathbf{n_2}), type(\mathbf{n_1})) \in \mathcal{H}^*\}$. One node schema has several parents if the type that defines the node schema is referenced by several (parent) node schemas (cf. Example 2.7). Node schemas that are defined in the same type hierarchy are siblings.



	vertex/edge of schema graph	notation in the figure
vertices	nodeschema $\mathbf{n}$	$name(\mathbf{n})^{card(\mathbf{n})}$
	simple type t	$< name(t) >^S$
	complex type t	$< name(t) >$
edges	type	$\mathbf{n} \rightarrow t$
	defines	$t \rightarrow \mathbf{n}$
	extends	$t \multimap t'$

Figure 2.3: Schema model for the document of Figure 2.2.

Definition 2.5 (document order) The order between two sibling node schemas $\mathbf{n}, \mathbf{n'} \in \mathbf{N}$, written as $\mathbf{n} \prec \mathbf{n'}$, corresponds to the order in which their type defines the node schemas. If a type inherits node schemas, its own node schemas succeed inherited node schemas. Attribute node schemas always precede element node schemas.

The main differences between this schema model and XML Schema are that XML Schema defines text node schemas only implicitly by assigning a simple type to an element declaration. To handle all node kinds analogously, we use node schemas instead of element and attribute declarations. To reduce complexity, the schema model does not support recursion. As the purpose of our schema model is not validation, it does not support e.g. enumerations, restriction and extension of simple types and more complex cardinality constraints. XML Schema defines a wide range of simple types. In this thesis, we only consider string and integer, which are the most important simple types.

Similar to nodes in a document, node schemas are connected via edges, defining path schemas to traverse the schema graph. There is one path schema from the root of the schema to each node schema.

Definition 2.6 (path schema) Let $\mathbf{S}$ be a schema according to Definition 2.4 with the set of node schemas $\mathbf{N}$.

- A *rooted path schema* $\mathbf{p}$ is a sequence of k connected node schemas $\langle/\mathbf{n_1}/\ldots/\mathbf{n_k}\rangle$, where $\mathbf{n_1} = root(\mathbf{S}) \wedge \mathbf{n_i} \in parent(\mathbf{n_{i+1}})$ for $\mathbf{n_i} \in \mathbf{N}$ and $i = 1 \ldots k{-}1$.
- Let $\mathbf{P}$ be the set of rooted path schemas in $\mathbf{S}$.

- Function *last* : $\mathbf{P} \rightarrow \mathbf{N}$ returns the last node schema of a path schema $\mathbf{p} \in \mathbf{P}$, $last(\langle/\mathbf{n_1}/\ldots/\mathbf{n_k}\rangle) = \mathbf{n_k}$ for $k \geq 1$.
- The sequences of node schema names of the path schemas represent the *labelpaths* of a schema. Each path schema has a distinct labelpath.

□

Example 2.7 (path schema) *In the schema of Figure 2.3, sample path schemas consist of the following sequences of node schemas, which are represented by their labelpaths:* `/projects/project/milestone/title/text()` *and* `/projects/project/milestone/resource/description`*. Note that node schema* `text()` *has four parent node schemas (the three* `title` *node schemas and the node schema* `description`*). With each parent node schema it forms a distinct path schema. Regarding the second labelpath, node schema* `description` *has the parent node schema* `resource`*.*

A document is instance of a schema if each path of the document is instance of one path schema and each node of the document is instance of one node schema. The following definitions formally define when a document is a valid instance of a schema.

Definition 2.7 (path and node instance) Let D be a document with nodes N and paths P and $\mathbf{S}$ be a schema with node schemas $\mathbf{N}$ and path schemas $\mathbf{P}$.

- A path $p = \langle/n_1/\ldots/n_k\rangle \in P$ is instance of a path schema $\mathbf{p} = \langle/\mathbf{n_1}/\ldots/\mathbf{n_k}\rangle \in \mathbf{P}$, written as $p \vDash \mathbf{p}$, iff for $i = 1\ldots k, k \geq 1$
  - $name(n_i) = name(\mathbf{n_i})$, and
  - if the type of n_i is defined, it either equals the type of $\mathbf{n_i}$ or it is a subtype of the type of $\mathbf{n_i}$, $(type(n_i), type(\mathbf{n_i})) \in \mathcal{H}^*$. The type of n_i must not be declared as abstract in the schema.

  The first property ensures that the path and the path schema have the same labelpath. In a valid schema, there exists at most one path schema that fulfills this property because each path schema has a distinct labelpath. The second property checks whether the nodes on a path have types that are defined in the schema.
- The path schema of a node $n \in N$, represented by function *pathschema* : $N \rightarrow \mathbf{P}$, is defined to be the path schema $\mathbf{p} \in \mathbf{P}$ such that $path(n) \vDash \mathbf{p}$. The node schema of a node is defined by *nodeschema* : $N \rightarrow \mathbf{N}$, where $nodeschema(n) = last(pathschema(n))$. If the type of a node is undefined, it implicitly adopts the type of its node schema.

□

Example 2.8 (path and node instance) *In the document of Figure 2.2, a sample path is* `/projects[1]/project[1]/milestone[2]/title[1]/text()`*. This path is instance of the path schema* `/projects/project/milestone/title/text()`*, which belongs to the schema of Figure 2.3. It follows that this is the path schema of the text node with value '*`implementation`*' and that the node schema of this node is the node schema named* `text()`*.*

In SCIENS, each document is instance of one schema, i.e. it is said to be valid against the schema or to conform to the schema. Based on Definition 2.7, Definition 2.8 defines when a document is a valid instance of a schema.

Definition 2.8 (document instance) A document D is a valid instance of a schema $\mathbf{S}$, $D \vDash \mathbf{S}$, iff

- The set of types defined in D is a subset of the types defined in $\mathbf{S}$, i.e. for each type name in D there exists a type in $\mathbf{S}$ with the same name.
- Each path p in P is instance of exactly one path schema $\mathbf{p}$ in $\mathbf{P}$:
  - $\forall p \in P : \exists \mathbf{p} \in \mathbf{P} : p \vDash \mathbf{p}$ and
  - $\forall p \in P, \mathbf{p}, \mathbf{p}' \in \mathbf{P} : p \vDash \mathbf{p} \wedge p \vDash \mathbf{p}' \Rightarrow \mathbf{p} = \mathbf{p}'$.
- There do not exist two distinct nodes which have the same parent and are instances of the same node schema if the node schema is single-valued, $\forall n, n' \in N : \mathit{parent}(n) = \mathit{parent}(n') \wedge \mathit{nodeschema}(n) = \mathit{nodeschema}(n') \wedge \mathit{card}(\mathit{nodeschema}(n)) \in \{0, 1\} \Rightarrow n = n'$.
- Each node whose type defines or inherits a required node schema has at least one child for each required node schema, $\forall n \in N, \forall \mathbf{n} \in (\mathit{defines}(\mathit{type}(n)) \cup \mathit{inherits}(\mathit{type}(n))) : \mathit{card}(\mathbf{n}) \in \{1, +\} \Rightarrow \exists n' \in N : \mathit{parent}(n') = n \wedge \mathit{nodeschema}(n) = \mathbf{n}$.
- The order between node schemas is respected within sibling nodes, $\forall n, n' \in N : \mathit{parent}(n) = \mathit{parent}(n') \wedge \mathit{nodeschema}(n) \prec \mathit{nodeschema}(n') \Rightarrow n \prec n'$.

□

Example 2.9 (document instance) *The document of Figure 2.2 is a valid instance of the schema of Figure 2.3. Each of its types is defined in the schema, each path is instance of a path schema and it fulfills the cardinality constraints of the schema. For example, each resource has exactly one title (the node schema* `title` *is single-valued and required) and at most one date. Sibling nodes respect the order of their node schemas. For example, the* `title` *of the first resource precedes the* `description` *of this resource.*

2.2 Processing Queries

The most well-known XML query languages are XPath [27] and XQuery [33], which have been proposed by the W3C. XPath provides path expressions to address nodes of XML trees, whereas XQuery extends XPath by expressions for combining, restructuring and sorting information and constructing XML structures.

XPath expressions consist of one or more steps that are evaluated from left to right. Each step generates a sequence of nodes, which are input for the subsequent step. A step consists of an axis, defining the direction of movement, and a node test, selecting nodes based on their kind, name and/or type. Optionally it is followed by predicates that filter the node sequence to which the step has evaluated and only retain the nodes that fulfill the predicates. The result of a path expression contains a sequence of nodes that is duplicate-free and in document order.

Example 2.10 (path expression) *A sample path expression is* `//project [title = 'semcrypt']//resource [@date < '2008-01-01']`*, selecting all resources of project '*`semcrypt`*' that have been edited before '*`2008-01-01`*'. Looking at the document of Figure 2.2, the path expression is evaluated as follows. The first step is* `//project [title = 'semcrypt']` *and selects all descendants of the document that have the name* `project`*. In the example, this step generates the node sequence consisting of the one and only* `project` *node. The predicate filters this sequence, retaining only projects with title '*`semcrypt`*'. More precisely, it selects the child nodes named* `title` *and tests whether they have a text node with value '*`semcrypt`*'. As the project node has a corresponding child, the project node is input to the second step expression,* `//resource [@date < '2008-01-01']`*. This step first selects all descendants of the project node with name* `resource` *and then discards the resources edited after 2007. The project has two resources, but only the first one fulfills the predicate and is returned by the path expression.*

This thesis uses a subset of XPath 2.0 [27], which is expressive enough to formulate common queries on XML documents. Compared to Core XPath [86], which is a subset frequently referred to in literature, this subset also supports full-text queries and queries on type hierarchies. In contrast to XPath or XQuery, it does not support e.g. sorting, grouping, aggregation functions and result construction as the focus is on queries to be supported by indices. For better readability, we use a simplified version of the XPath syntax for these queries.

PathExpr	::=	(StepExpr PredicateList)$^+$
StepExpr	::=	"/" | "//" NodeTest
NodeTest	::=	NodeName | "*" ("<" TypeName "*"? ">")?
PredicateList	::=	("[" Predicate "]")*
Predicate	::=	ComparisonExpr ("and" ComparisonExpr)?
ComparisonExpr	::=	NodeTest (StepExpr)* (CompOp Value)?
CompOp	::=	"$<$" | "$\leq$" | "$=$" | "$\geq$" | "$>$" | "$\sim$" | "$\dot{\sim}$"

Table 2.1: Simplified EBNF for queries on XML documents.

Table 2.1 depicts the grammar used to formulate queries in this thesis. The syntax corresponds to the EBNF of the XPath 2.0 specification. The query language only supports the child (/) and descendant (//) axes. Each node test consists of a node name or a wildcard (*) and an optional type name. If the type name is followed by a star (*), the query refers to the corresponding type hierarchy, otherwise it only retrieves nodes with the specific type and not nodes belonging to any of its subtypes. Predicates allow for evaluating structural conditions on nodes or comparing node values. The special comparison operators $\sim$ and $\dot{\sim}$ are used for full-text queries, similar to the `matches` function in XPath 2.0. The operators determine whether any word of a node value equals ($\sim$) or starts with ($\dot{\sim}$) the specified value, respectively. Node names, type names and values consist of characters, taken from the set of node names, type names and values of the data model (cf. Section 2.1.1). Pursuant to the XPath 2.0 specification, value comparisons take the content of element nodes or the value of attribute or text nodes and compare it with the specified value. The query result comprises the nodes returned by the query and their descendants.

Typical queries on XML documents specify structural conditions using path expressions and search the document content with predicates (cf. XML query use cases [44] and XML benchmarks [173, 199]). Catania et al. [43] propose to classify queries according to each one of the following three dimensions:

- *Simple/branching path query:* In contrast to a simple path query, a branching path query contains branches, i.e. it visits several labelpaths.
- *Total/partial matching query:* While a total matching query begins at the root of a document, a partial matching query starts from some nodes of the document that are selected via the descendant axis.
- *Structure/content-oriented query:* Queries with predicates on node values are content-oriented, whereas remaining queries are referred to as structure-oriented queries.

Queries on the document structure are hierarchical queries as both the paths as well as the types in a document form hierarchies. Queries on the document content compare node values by exact match, range or textual constraints. Generally, queries evaluate the following conditions on XML documents (cf. Example 2.11 for sample queries):

- **Queries on the document structure:**
  - Queries select nodes by their node names. They either specify all names of a path or omit some of them by using wildcards or the descendant axis. As the node names of a path form the labelpath of a node, we refer to such queries as queries on the *labelpath hierarchy*.
  - As several nodes may share the same labelpath, labelpaths alone are insufficient to select nodes. Apart from node names, the queried hierarchy can be restricted by any kind of predicate. Predicates limit search to specific paths (subtrees) of documents. These queries thus restrict the *path hierarchy* of documents.
  - Not only paths but also types form a tree hierarchy in XML. Queries either select all nodes belonging to a specific type, to a partial type hierarchy or to the entire type hierarchy. We refer to such queries as queries on the *type hierarchy*.

  As labelpaths, paths and types form trees in XML, queries on the document structure are typically hierarchical queries, selecting either the entire hierarchy, a partial hierarchy or only a specific part of the hierarchy.
- **Queries on the document content:**
  - To filter nodes returned by a path expression, queries can compare the values of these nodes with a specific value and discard all nodes whose value does not equal the specified value. These queries are referred to as *exact match* queries.

– Queries do not compare values with one specific value only. Instead, queries test whether node values (e.g. numbers, dates) are within a certain value range. In this case, we talk about *range* queries.

– An important characteristic of XML documents is that they can contain unstructured parts. To enable search within unstructured data, an essential class of queries determines whether a node value contains a specific word or word prefix via a regular expression. We refer to such queries as *full-text* queries.

Queries on the document content select nodes whose value equals a certain value, is within a value range or contains a specific word or word prefix.

Structure-oriented queries	
Q1	/projects/project/title
Q2	//milestone//title
Q3	//title
Q4	//resource[description]/title
Exact match queries on labelpath hierarchy	
Q5	/projects/project[title = ‘semcrypt’]
Q6	//*[title = ‘xml’]
Q7	//resource[@date = ‘2007-10-31’][title = ‘xml’]
Range queries on path hierarchy	
Q8	//resource[@date $\geq$ ‘2007-01-01’ and @date $<$ ‘2008-01-01’]
Q9	//project[title = ‘semcrypt’]//resource[@date $\geq$ ‘2008-01-01’]
Q10	//project[@id = ‘26543’]/milestone[title = ‘design’]/resource[@date $\geq$ ‘2008-01-01’]
Full-text queries on type hierarchy	
Q11	//resource<Report*>
Q12	//resource[description $\sim$ ‘database’]
Q13	//resource<TechnicalReport>[description $\dot{\sim}$ ‘data’]

Table 2.2: Queries on the document of Figure 2.2.

Example 2.11 (queries) *Table 2.2 depicts sample queries on the document of Figure 2.2 using the query syntax of Table 2.1. Q1-Q4 are structure-oriented queries and select nodes according to the node names contained in their labelpaths. Q1 is a total matching query and selects all project titles. Q2-Q4 are partial matching queries. Q2 retrieves all titles which are descendants of milestones, i.e. the titles of milestones and resources, whereas Q3 returns all titles. Q4 is a branching path query with a structural predicate, returning the titles of resources that have a description.*

The subsequent queries do not only restrict the structure, but also contain value-based predicates. Thereby, each query group focuses on one structure- and one content-oriented condition. Q5-Q7 are exact match queries, comparing titles with a specific value. While Q5 exactly specifies the labelpath of the nodes to be returned, Q6 refers to the entire hierarchy and returns all nodes with the specified title. Q7 retrieves all resources with a specific date and title.

Q8-Q10 are examples for range queries that select resources according to their date. All resources have the same labelpath, but they are located in different path hierarchies. While Q8 looks at all resources, Q9 and Q10 limit search to certain subtrees of the document using predicates. Q9 selects resources of project '`semcrypt`', Q10 resources of the first milestone of this project in the sample document of Figure 2.2. The first two predicates of Q10 select the partial hierarchy with root `/projects[1]/project[1]/milestone[1]`, within which the remaining part of the query is evaluated.

The remaining queries of Table 2.2 query the type hierarchy of resources, whereby Q12 and Q13 additionally contain full-text queries. Q11 returns all nodes with type `<Report>`, whereas Q12 selects resources whose description contains the word '`database`'. As only nodes with type `<Report>` have a description, Q12 implicitly refers to the partial hierarchy of type `<Report>`. Q13 retrieves all nodes with type `<TechnicalReport>` whose description contains any word starting with the prefix '`data`' (such as '`data`' or '`database`'). All sample queries use the simplified XPath syntax of Table 2.1. In XPath 2.0, Q13, for example, would be written as `//element(resource, TechnicalReport) [description/fn:matches(., '\bdata.*')]`.

Various approaches exist to evaluate queries. Gottlob et al. [86] present algorithms for processing XPath queries in main memory. Pattern matching algorithms (e.g. [42, 50, 142, 200]) represent queries as tree patterns and find occurrences of these patterns in the document. To avoid scanning whole documents, various approaches use indices on the document structure and/or content. To evaluate queries by accessing several indices, structural join algorithms (e.g. [135]) join nodes returned by indices according to their structural relationships.

Example 2.12 (query evaluation) *Basically, the following steps are necessary to evaluate Q8 of Table 2.2 (`//resource[@date` $\geq$ `'2007-01-01' and @date` $<$ `'2008-01-01']`) without indices: (1) traverse the document to locate resource nodes, (2) for each resource node, navigate to its date and determine whether the date value is within the specified value range. To accelerate the first step, we could use an index on labelpaths which returns all nodes belonging to the labelpath of resources. Instead of indexing the document structure, we could also use a B-tree on date values to accelerate the second step of the query processing algorithm. If we access both indices, we need a structural join algorithm to determine which dates belong to which resources.*

2.3 Indexing Requirements

Databases organize records into files, which consist of pages. To process queries without scanning all records, databases rely on indices. Ramakrishnan and Gehrke [164] define an index as '*a data structure that organizes data records on disk... to efficiently retrieve all records that satisfy search conditions on the search key fields of the index*'. The search key fields are any properties of the indexed records. An index either returns (i) the actual data records that match the search keys or returns (ii) references to the corresponding data records. Garcia-Molina et al. [83] refer to alternative (i) as the primary index, whereas alternative (ii) is a secondary index. Ramakrishnan and Gehrke [164] use a different definition, which does not focus on what the index returns but on what it indexes. According to them, an index on a set of fields including the primary key is a primary index, otherwise it is a secondary index.

In XML, records comprise nodes of documents. In our terminology, an *index* is a search function that consists of a set of index entries. Each index entry maps a list of index keys to nodes. An index search compares the index keys with the search conditions and returns all nodes associated with index keys that match the search conditions. An index key can be any property of a node that can be indexed. Indexable *properties* are the value, type, labelpath, path or name of a node. An *index structure* is the specific data structure used to organize index entries, such as a hash table or a B-tree.

Concerning primary and secondary indices, we follow the definition of Garcia-Molina et al. [83]. An index which organizes the nodes of a document is referred to as the primary index. On the contrary, a secondary index returns node identifiers. We assume that node identifiers are logical identifers that can be used to obtain the corresponding nodes from the primary index. Typically there is one (default) primary index defined on each document. However, it is possible to add several secondary indices defined on various properties of the XML data model.

Example 2.13 (index) *To support Q8 of Table 2.2 (`//resource[@date≥'2007-01-01' and @date<'2008-01-01']`), we can define a secondary index on dates which uses the value of dates as index keys and returns the node identifiers of resource nodes. To support range queries, a B-tree is an adequate data structure for this index. Regarding the sample document of Figure 2.2, a sample index entry maps the index key '`2007-03-12`' to the node identifier of the first resource. When accessing the index with search keys '`2007-01-01`' and '`2008-01-01`', specifying the requested date range, the index returns the node identifier of the first resource as its index key '`2007-03-12`' matches the search condition.*

In consideration of frequent queries presented in Section 2.2, a database requires indices on the document structure and on the document content as well as indices on both the structure and the content of documents:

- **Indices on the document structure:**
  - *Labelpath hierarchy:* Indices should enable the database to select nodes by their labelpath or node name to return all nodes belonging to a specific labelpath or having a certain node name, respectively. As queries frequently refer to several labelpaths that form a subhierarchy, indices should be able to select all nodes of a subhierarchy without having to access the index for each individual labelpath of the queried subhierarchy.

    Example 2.14 (labelpath hierarchy index) *An index on labelpaths supports Q1 of Table 2.2 by returning all nodes with the queried labelpath. Similarly, an index on node names is efficient for Q3, which selects all nodes with the requested name. As Q3 selects several labelpaths, it is also possible to define an index on the labelpath* *hierarchy of title nodes. This index supports Q1-Q3 as it can select titles within different subhierarchies, i.e. within projects, milestones or resources.*

- *Path hierarchy:* Similar to indices on the labelpath hierarchy, indices on the path hierarchy support queries on specific subtrees of documents. When a query restricts search to a specific subtree, e.g. with the help of a predicate, or starts search from a previous query result, an index on paths can identify the queried subtree and limit index traversal to the queried subtree. Indices on paths therefore avoid the need of looking at the entire document and performing structural joins between partial query results.

  Example 2.15 (path hierarchy index) *Queries Q8-Q10 of Table 2.2 select resources within different projects and milestones. As soon as the requested project/milestone has been identified, it should no longer be necessary to consider all resources regardless of the project/milestone within which they are located. To avoid processing structural joins between all resources and the queried project/milestone, an index on the path hierarchy of resources is appropriate. It groups resources according to their project and milestone and can therefore limit search to the queried project or milestone hierarchy.*

- *Type hierarchy:* Indices on types are required to support querying nodes by types without the need of scanning all nodes. They should return all nodes with a specific type or whose type is part of the requested type hierarchy.

  Example 2.16 (type hierarchy index) *Indexing the type hierarchy of resources supports queries Q11-Q13 of Table 2.2, by selecting all resources that are reports or only technical reports.*

As labelpaths, paths and types form a tree hierarchy, indices on the document structure need to support hierarchical queries, selecting either the entire hierarchy, a partial hierarchy or only a specific part of the hierarchy.

- **Indices on the document content:**

  - *Exact match index:* A database should provide indices that can return all nodes with a specific value without scanning the values of all nodes.

    Example 2.17 (exact match index) *To support Q5 of Table 2.2, an exact match index on the values of title nodes can return all nodes with the requested title '`semcrypt`'.*

  - *Range index:* While exact match indices support queries asking for one specific value, range indices are required for retrieving all nodes whose value is within a certain value range.

Example 2.18 (range index) *Queries Q8-Q10 of Table 2.2 compare the values of date nodes with a value range. With the help of a range index on date values, a database can evaluate these queries without having to look at each date.*

– *Full-text index:* To support full-text queries, it is necessary to index the individual words of node values and structure them in a way that allows for comparing the indexed words with words or word prefixes.

Example 2.19 (full-text index) *Queries Q12 and Q13 of Table 2.2 compare node values with a word and word prefix, respectively. A full-text index on the values of description nodes can evaluate these queries without looking at each node value.*

Indices on the document content index node values, but need to support different operations on these values. Required operations include exact and range comparisons as well as word and word prefix search. Instead of indexing all node values, indices on the document content are typically selective indices in that they only index the values of nodes with the same labelpath or name.

- **Indices on the document structure and content:** Queries typically constrain both the document structure and the document content. Indices should therefore support any combination of structure- and/or content-oriented properties. When indexing multiple properties in one index, it should be possible to adapt indices to the query workload. For example, indices on the structure and content should either favor structure-oriented or content-oriented queries or equally support both kinds of queries. Which query is best supported by an index depends on how the index structures its search space, i.e. according to which properties it orders index entries.

  Example 2.20 (indices on the structure and content) *With regard to queries Q8-Q10 of Table 2.2, a range index on dates and a path hierarchy index on resources can be useful. To avoid accessing both indices and joining their results, an index on both the date and the path hierarchy is most appropriate. If Q8 is evaluated most often, this index should favor querying the date, whereas if Q10 is more frequent, the index should favor querying the path hierarchy. In case that Q8-Q10 occur the same number of times, the index should perform equally for all queries.*

To summarize, an XML database requires indices on the document structure and content. For this purpose, indices need to be able to index various properties of nodes, namely labelpaths, paths, types and values. They should support various operations on these properties - hierarchical comparisons on structural properties and exact, range and textual comparisons on values. Queries typically constrain various properties, raising the need for indices on multiple properties that can be adapted to the query workload.

Chapter 3

Labeling Scheme

Contents

This chapter describes how to assign labels to schemas and documents such that indices can use labels for comparing and representing nodes and their properties. Section 3.1 identifies requirements to be supported by labels for indexing and Section 3.2 reviews related work. Based on existing labeling approaches, Section 3.3 proposes how to label path schemas and how to integrate schema information into node labels. Section 3.4 extends this approach by labeling type hierarchies and by integrating type information into node labels. Processing labels to support the required indexing operations is discussed in Section 3.5. Finally, Section 3.6 summarizes the main ideas of the proposed labeling scheme.

3.1 Introduction

Indices require nodes and their properties to be represented in a way that allows for efficiently querying and updating the index as well as compactly storing the index structure. More precisely, indices on the document structure need to represent and compare paths, labelpaths and types. As secondary indices return node identifiers, encoding structural information into node identifiers enables the database to process nodes returned by indices without accessing documents. Integrating schema information into node identifiers further improves query and update processing.

A labeling scheme assigns a unique label to each node of a document. The label is not a simple number, but encodes structural information about the node. Recently, a number of labeling schemes have been proposed (e.g. [72, 130, 135, 158]) with the main purpose of (i) accelerating query and update processing by performing operations on labels instead of accessing documents and (ii) providing a compact storage representation of nodes.

With regard to indexing, a labeling scheme (i) accelerates index traversal, selection and maintenance and (ii) provides a compact storage representation of indexed properties and nodes returned by indices. For this purpose, the labeling scheme should fulfill the following requirements:

- *Path operations:* Labels should encode paths in such a way that the following operations can be performed directly on labels:
  - Evaluate structural relationships between nodes, i.e. determine document order, parent-child and ancestor-descendant relationships between nodes.
  - Facilitate navigation by enabling to calculate the labels of ancestor nodes.

  Labels that support these path operations facilitate processing nodes returned by indices. They enable indices on paths to compare search keys with indexed properties when accessing or updating the index structure. With regard to index maintenance, navigating to nodes via labels speeds up finding all nodes which are affected by an index update operation.

- *Dynamic documents:* XML documents are not static but may be subject to frequent updates. If updates on documents modify existing node labels, it is necessary to update all indices referring to these labels. Only immutable labels impede such expensive relabeling operations and guarantee that existing nodes labels do not change when inserting or deleting nodes.

- *Efficient encoding:* Labels allow for representing nodes returned by indices as well as structural properties that can be indexed. An efficient encoding of labels is necessary to compactly store indices. The encoding should also provide for fast comparison between labels.

- *Schema information:* Labels should be as expressive as possible and capture all structural information about a node. This implies that they should not only encode paths and support operations on paths. Integrating labelpaths and types, which are defined by schemas, enables indices to represent and compare these properties and further improves index maintenance. Comparing labelpaths requires the same operations as on paths, whereas types are typically compared according to super/subtype relationships.

To summarize, the labeling scheme must support representing and comparing paths, labelpaths and types. Labels must be immutable and provide an efficient encoding. In [88], we have outlined a labeling scheme fulfilling these requirements, which we describe in this chapter. Although indexing does not depend on this particular labeling scheme, it improves many index processing tasks. We will point out these advantages in the subsequent chapters after reviewing related work.

3.2 Related Work

Numerous labeling schemes[1] have been proposed recently to support querying and updating XML documents.

Containment labeling schemes[2] assign to each node two numbers (expressing a range) in some kind of tree traversal such that the range of each node is within the range of its ancestors. First presented by Dietz [67] and Agrawal et al. [10], containment labeling schemes were later applied to XML [90, 135]. A containment label mostly consists of three numbers $(order, size, level)$ (cf. Example 3.1). The labeling scheme assigns order and size tags, which are ascending numbers, in a preorder and postorder traversal, respectively. The level indicates the tree depth at a particular node and equals the number of nodes on a node's path.

In a *prefix labeling scheme*, each node encodes the label of its parent as a prefix in its label. This idea originates from the Dewey Decimal Classification [65], which has been proposed to classify topics in libraries. In XML labeling schemes, each node takes the label of its parent as prefix and adds a self label (cf. Example 3.1). The self label is ascending and unique within siblings. As self label, labeling schemes either use numbers [158, 186] or binary strings [57, 120, 130].

Lee et al. [127] represent a document as a k-ary complete tree where k is the largest number of children of an element node in the document. They assign labels to the nodes by traversing the tree in level order (cf. Example 3.1).

Example 3.1 (labeling schemes) *Figure 3.1 labels a sample document using the containment, prefix and k-ary complete-tree labeling scheme. Containment labels are shown left to each node in parenthesis, prefix labels are written below each node and k-ary labels are depicted in brackets to the right of each node. Thereby, $k = 2$ as each node contains at most two children.*

In the following, we look at how existing labeling schemes fulfill the requirements identified in Section 3.1.

[1]The terms numbering scheme or node identification scheme are also frequent.

[2]Containment labeling schemes are also referred to as range or interval labeling schemes.

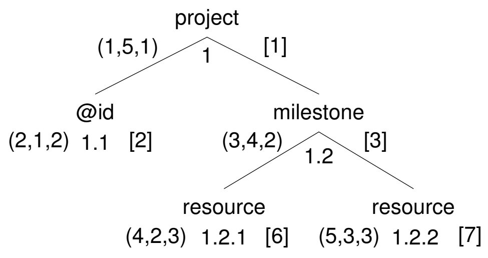


Figure 3.1: Containment, prefix and k-ary complete-tree labeling schemes.

3.2.1 Path Operations

Containment labeling schemes represent document order by assigning order tags in a preorder traversal. Node u is an ancestor of node v iff $order(u) < order(v) \wedge size(v) < size(u)$. If additionally $level(u) = level(v) - 1$ holds, node u is the parent of node v. Despite enabling to evaluate structural relationships, containment labeling schemes fail in supporting navigation to ancestor labels.

Prefix labeling schemes compare document order between nodes by the lexicographic order of their labels. Node u is an ancestor of node v iff $label(u)$ is a prefix of $label(v)$. As prefix labeling schemes encode parent labels, they easily support navigating to ancestors by calculating their node labels.

The k-ary complete-tree labeling scheme [127] can calculate the parent u of node v by $label(u) = (label(v) - 2)/k + 1$ and therefore supports both evaluating structural relationships as well as navigating to ancestor labels. To evaluate document order between labels that are not in an ancestor-descendant relationship, it is necessary to determine the level of each label and compare the numeric order between labels at the same level.

Example 3.2 (path operations) *In Figure 3.1, the node with label* $(3, 4, 2)$ *is the parent of the node with label* $(4, 2, 3)$ *because* $3 < 4 \wedge 2 < 4 \wedge (2 = 3 - 1)$. *By looking at the prefix labels, we see that label* 1.2 *is a prefix of label* 1.2.1. *From the prefix label* 1.2.2, *we can also calculate its ancestor labels, which are* 1.2 *and* 1. *The parent of the 2-ary label* 6 *can be calculated by* $(6 - 2)/2 + 1 = 3$.

3.2.2 Dynamic Documents

The first XML labeling schemes only considered static documents. To support dynamic documents, labeling schemes must guarantee that inserting or deleting nodes does not affect existing node labels. Basically, containment labeling schemes need to relabel all nodes after an insert operation. Prefix labeling schemes need to relabel following-siblings and their descendants. If the maximum number of children increases, the k-ary complete-tree labeling scheme has to relabel all nodes.

Example 3.3 (relabeling) *When inserting a new resource between the existing resources in Figure 3.1, the containment and k-ary complete-tree labeling scheme require relabeling all nodes. Regarding the prefix labeling scheme, the new resource gets the label* 1.2.2 *and the last resource needs to be relabeled to* 1.2.3.

The first approaches which support dynamic documents use floating point numbers [13] or leave gaps when assigning node labels [135]. However, floating point numbers and gaps only support a limited number of insertions. Some approaches try to reduce relabeling costs by computing the necessary gap size based on statistics [73] or by limiting relabeling to certain fragments of a document [115, 181, 204]. The approach of Wu et al. [198] is based on prime numbers. The label of each node is the product of the parent's label and the own self label. They keep document order among nodes by a special value calculated by the Chinese Remainder Theorem. However, this value grows rapidly and its calculation is very expensive [130].

To avoid costly relabeling of nodes after updates, dynamic prefix labeling schemes guarantee immutable node labels. Binary string prefix labeling schemes [120, 130] avoid relabeling by assembling binary strings in document order. For example, the approach of Li and Ling [130] uses the following algorithm to insert a new node n_m between the nodes n_l and n_r. If $size(n_l) \geq size(n_r)$, then $n_m = n_l$+'1'. Otherwise, $n_m = n_r$ with last bit '1' changed to '01'. Kit et al. [119] present a multi-dimensional labeling scheme which adds dimensions when an update would otherwise require relabeling. Another possibility is to use numbers and letters in the labels to support order-sensitive updates (cf. [72, 116]). Ordpath [158] is a prefix labeling scheme which initially only assigns odd numbers and reserves even numbers for updates. Härder et al. [97] pursue a similar approach, but additionally use a distance parameter to leave gaps.

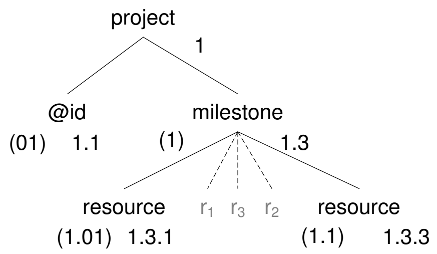


Figure 3.2: Dynamic labeling schemes.

Example 3.4 (dynamic labeling) *Figure 3.2 assigns node labels according to the approach of Li and Ling [130], shown to the left in parenthesis of each node, and Ordpath [158], depicted to the right of each node. When inserting resources r_1, r_2 and r_3 (in this order), Li and Ling assign the labels* 1.011, 1.0111 *and* 1.01101*. Ordpath associates the labels* 1.3.2.1, 1.3.2.3 *and* 1.3.2.2.1 *with the new nodes.*

3.2.3 Encoding Labels

Various encodings have been proposed to compactly store labels. While static labeling schemes can use a fixed-length encoding, dynamic labeling schemes require a variable-length encoding [57, 110].

Li et al. [132] present an encoding based on binary strings, O'Neil et al. [158] for integer based prefix labels. They store the size of each variable-length label in a fixed-length field. However, the fixed-length field may become too small to represent labels [97, 133], entailing that this encoding may be insufficient for dynamic documents.

Separators enable the distinction of the individual parts of a prefix label (e.g. [57]). However, the use of separators impedes fast bit- or byte-level comparison between labels [97]. Li et al. [131, 133] present a more efficient solution based on a quaternary encoding, but it may result in long labels when the size of a document is not known in advance.

Härder et al. [97] present a Huffman encoding for prefix labels. Huffman trees provide prefix-free codes, each of which is associated with a specific length determining the length of a label. Huffman codes support encoding labels without requiring a fixed-length field or a separator. Thus they provide both a compact representation as well as efficient comparisons of labels.

Example 3.5 (encoding) *According to the encoding of Härder et al. [97], we can associate the Huffman code* 0 *with length* 3 *and code* 100 *with length* 4*. We then encode label* 1.3.5 *with* 0 001 0 011 0 101 *and label* 1.3.11 *with* 0 001 0 011 100 0011*. The bit sequence* 0 001 *encodes the first part of the label, 1, with the three bits* 001*, as indicated by the Huffman code* 0*. The lexicographical order between the encodings expresses document order.*

3.2.4 Integrating Schema Information

Only few labeling schemes integrate schema information to improve query and update processing. Bremer and Gertz [39, 40] construct a DataGuide [85] from XML documents, which is a summarization of path information. Each path of the DataGuide is assigned a number that identifies the corresponding labelpath. The label of each node consists of the number of its labelpath and a prefix label. The authors also propose how to improve query processing with the additional schema information encoded into labels. This approach has later been applied to processing the nodes returned by indices [96, 174].

Example 3.6 (schema integration) *Looking at Figure 3.2, assume that we associate labelpath* `/project/milestone` *with number* 3 *and labelpath* `/project/milestone/resource` *with number* 4*. By enhancing prefix labels with these numbers, we get label* 3–1.3 *for the milestone node and label* 4–1.3.1 *for the first resource. With the help of the node label and the schema, it is possible to determine the node name and its labelpath, which facilitates query processing.*

Instead of simply assigning ascending numbers to labelpaths, Li et al. [134] encode structural relationships into the numbers assigned to labelpaths. The BUS indexing scheme [180] assigns numbers to the document structure in the same way as the k-ary complete-tree labeling scheme. Spider [81, 114] uses a similar approach, which does not only support evaluating structural relationships between labelpaths, but also calculating ancestor labelpaths to further improve query processing.

3.2.5 Comparison

Containment labeling schemes cannot calculate ancestor labels and require relabeling after updates. Prefix labeling schemes support the required path operations, dynamic documents and efficient encoding. While existing schema-aware labeling schemes integrate labelpaths, they do not take into account types and type hierarchies and only consider static documents.

None of the existing labeling schemes fully matches the requirements presented in Subsection 3.1. By combining several approaches, we can construct a dynamic labeling scheme that supports representing and comparing paths and labelpaths. We further need to extend this labeling scheme to also include types and type hierarchies.

3.3 Basic Approach

This section looks at how to extend existing labeling schemes with labelpaths. A schema defines all labelpaths that exist in a document via path schemas. By labeling these path schemas, each labelpath gets a unique label as well. Subsection 3.3.1 describes labeling the schema graph, while Subsection 3.3.2 addresses integrating schema labels into node labels. Neither labeling the schema nor labeling the document depends on a specific labeling scheme, as several labeling approaches support the identified requirements (cf. Section 3.2). When describing the labeling algorithms, we therefore first abstract from specific labeling schemes and subsequently depict a sample labeling approach. We basically construct labels that directly support all required operations. Additionally, we discuss some space improvements assuming that the schema resides in main memory and can be accessed efficiently to compare labels.

3.3.1 Schema Labeling

This subsection describes an algorithm to label the schema graph defined in Section 2.1.2 in order to support representing and comparing labelpaths. Required operations on labelpaths include evaluating document order and structural relationships and calculating ancestor labelpaths. The schema is usually very small compared to the documents and fits into main memory. Nevertheless performing operations directly on labels is preferable to accessing the schema to compare labelpaths. In case that the schema is not static, the labeling scheme must further support updates without relabeling.

Labelpaths form a tree. To support the required operations on labelpaths, we convert the schema graph into a tree of path schemas. Each path schema represents a labelpath and is assigned a unique label. Algorithm 3.1 describes how to recursively extract and label the path schemas of a schema graph. The algorithm is first called with the root node schema as current node schema and with the parent and previous-sibling path schema set to null. Each recursion visits a node schema $\mathbf{n}$ and creates a new path schema $\mathbf{p} = \langle \mathbf{n_1}/\dots/\mathbf{n_k} \rangle$, where $\langle \mathbf{n_1}/\dots/\mathbf{n_{k-1}} \rangle$ equals the parent path schema and $\mathbf{n_k} = \mathbf{n}$. After assigning a label to this path schema, the algorithm determines all children of the node schema. The algorithm then recursively creates a path schema for each child node schema.

Algorithm 3.1 Extract and label the path schemas of a schema graph.

Input: Node schema $\mathbf{n}$ of the schema graph; Parent path schema $\mathbf{p}_{par}$, which is the path schema of one of the node schema's parents or null if the node schema does not have a parent; Previous-sibling path schema $\mathbf{p}_{sib}$, which is the path schema of the node schema's previous sibling or null if the node schema does not have a previous sibling.
Output: Labeled path schema $\mathbf{p}$, which has the parent path schema $\mathbf{p}_{par}$ and the last node schema $\mathbf{n}$.
Description: Generate path schema $\mathbf{p}$ for node schema $\mathbf{n}$ with parent path schema $\mathbf{p}_{par}$ and assign a label to this path schema based on its parent and its sibling path schema $\mathbf{p}_{sib}$. Then recursively generate path schemas for descendant node schemas.
Result: Path schema for node schema $\mathbf{n}$ with a unique label.

```
1: procedure labelSchema($\mathbf{n}$, $\mathbf{p}_{par}$, $\mathbf{p}_{sib}$)
2:     $\mathbf{p}$ = createPathSchema($\mathbf{n}$, $\mathbf{p}_{par}$)
3:     $label(\mathbf{p})$ = createLabel($label(\mathbf{p}_{par}), label(\mathbf{p}_{sib})$)
4:     $\mathbf{N}_{child} = \{\mathbf{n}_c \in \mathbf{N} \mid \mathbf{n} \in parent(\mathbf{n}_c)\}$
5:     $\mathbf{p}_{sib} = \perp$                              ▷ set to null
6:     for all $\mathbf{n}_{child} \in \mathbf{N}_{child}$ do          ▷ in document order
7:         $\mathbf{p}_{sib}$ = labelSchema($\mathbf{n}_{child}, \mathbf{p}, \mathbf{p}_{sib}$)
8:     end for
9:     return $\mathbf{p}$
10: end procedure
```

To assign labels to path schemas, procedure createLabel requires (i) the label of the parent path schema to encode structural relationships and (ii) the label of the previous-sibling to keep order among labels. In case that the parent label is null, it assigns a new root label. A sibling label set to null indicates that the label to be assigned represents the first child of the parent. Various labeling schemes and encoding approaches presented in Section 3.2 support these requirements. For example, prefix labeling schemes can label dynamic schemas, whereas the k-ary complete-tree labeling scheme is sufficient for static schemas. Example 3.7 shows a sample labeling, which we will use in this thesis. As we assume the schema to be static, we use the k-ary complete-tree labeling scheme [127] to assign labels. This labeling scheme generates smaller labels than prefix labeling schemes. It requires to traverse the schema once before assigning labels to determine the value of k, which equals the maximum number of child node schemas. Labels can then be calculated with the parent and sibling label and with the static value k.

We define function $label : \mathbf{P} \rightarrow \mathbf{Z}$ to return the label of a path schema, where $\mathbf{Z}$ is the set of path schema labels. As path schemas form a tree, we define functions *children*, *descendant*, *parent* and *ancestor* analogous to the corresponding functions on nodes in a document tree (cf. Subsection 2.1.1).

Example 3.7 (schema labeling) *Figure 3.3 depicts the tree of path schemas generated by Algorithm 3.1 for the schema graph of Figure 2.3. Each path*

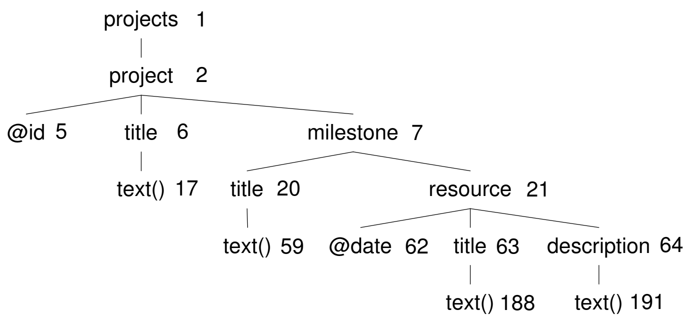


Figure 3.3: Labeling the path schemas of the schema in Figure 2.3.

schema represents a labelpath. As each path schema has at most three children, we use $k = 3$. For example, to assign a label to the path schema `/projects/project/milestone`*, the algorithm receives as input the node schema* `milestone`*, the parent path schema $\mathbf{p}_{par} = 2$ and the sibling path schema $\mathbf{p}_{sib} = 6$. To calculate the new label, procedure* CREATELABEL *increments the sibling label by 1. The newly created path schema has two children with node schemas* `title` *and* `resource`*, for which the algorithm generates labels in the next steps.*

In Section 2.1.2, we defined a schema graph to be acyclic, which impedes recursion. In case of a recursive schema graph, we propose to keep track of the maximum recursion depth in instance documents and extract as many path schemas as there currently exist in the documents.

3.3.2 Document Labeling

When labeling the document tree defined in Section 2.1.1, each node is assigned a unique label, consisting of a schema label and an instance label. The schema label corresponds to the label of the node's path schema (cf. Subsection 3.3.1), whereas the instance label identifies the position of the node in the specific document. The schema and instance label need to support the path operations identified in Subsection 3.1.

Algorithm 3.2 describes how to recursively assign labels to the nodes of a document. To assign a label to a node, procedure CREATELABEL requires the label of (i) the node's path schema to encode its labelpath, (ii) the parent node to encode structural relationships and (iii) the node's previous-sibling to keep document order between labels. In case that the parent label is null, the labeling scheme assigns a new root label. An unknown sibling label indicates that the current node is the first child of the parent.

In Subsection 2.1.1, we have introduced function *pathschema*, which returns the path schema of a node and thus connects a document with its schema. Programmatically, the algorithm can determine the path schema of a node as follows. The node schema of the node must be defined by the type or any supertype of the node's parent. Functions $defines$ and $inherits$ return these node schema candidates. Among these node schema candidates, there is one

Algorithm 3.2 Label the nodes of a document.

Input: Node n of the document; Parent node n_{par}, which is the node's parent or null if the node does not have a parent; Previous-sibling node n_{sib}, which is the previous sibling of the node or null if the node does not have a previous-sibling.

Description: Determine the path schema of node n and generate a label for the node with the help of its path schema, its parent n_{par} and its previous-sibling node n_{sib}. Then recursively generate labels for its descendants.

Result: Node n and all its descendants have assigned a unique label.

```
1: procedure labelDocument($n$, $n_{par}$, $n_{sib}$)
2:     $label(n) = \text{CREATELABEL}(label(pathschema(n)), label(n_{par}), label(n_{sib}))$
3:     $n_{sib} = \perp$                                  ▷ set to null
4:     for all $n_{child} \in children(n)$ do             ▷ in document order
5:         labelDocument($n_{child}, n, n_{sib}$)
6:         $n_{sib} = n_{child}$
7:     end for
8: end procedure
```

node schema which has the same name as the node. The requested path schema is the child of the parent path schema that ends with that node schema. When calling the algorithm with the root node, the parent and the sibling node are still set to null. In this case, the path schema of the processed node has to equal the root path schema.

The label of each node consists of a schema label, which corresponds to the label of the node's path schema, and an instance label. We define function $ilabel : N \rightarrow Z$ to return the instance label of a node, where Z is a set of instance labels. Function $slabel : N \rightarrow \mathbf{Z}$ returns the schema label of a node. Function $label : N \rightarrow (\mathbf{Z} \times Z)$ returns the entire label of a node, consisting of its schema and instance label.

The labels must support the required path operations, efficient encoding and updates without relabeling as documents are usually not static. Various labeling schemes presented in Section 3.2 fulfill these requirements. In this thesis, we use a dynamic prefix labeling scheme to assign instance labels, which is similar to Ordpath [158]. However, we enhance labels with schema labels, which also influences the assignment of instance labels (cf. [39]).

To reduce label size, we propose two properties. Property (i) is based on the fact that sibling nodes respect the order defined between sibling node schemas (cf. Definition 2.8). This allows for assigning self labels in ascending order within nodes with the same parent and with the same node schema. We refer to this property as *node schema property*. Property (ii) further reduces the label size by only adding a self label to a node's label if the node's node schema is multi-valued (*cardinality property*). When using this property, certain path operations require access to the schema (cf. Subsection 3.5).

Example 3.8 describes the sample labeling considering properties (i) and (ii). In subsequent chapters, we use both properties to assign instance labels as we believe that the smaller label size improves readability. Note that these properties cannot be used if the assumption that nodes follow the order defined in the schema does not hold, e.g. in case that mixed content occurs. In this case, it is necessary to assign instance labels in ascending order within nodes with the same parent independent of their node schema.

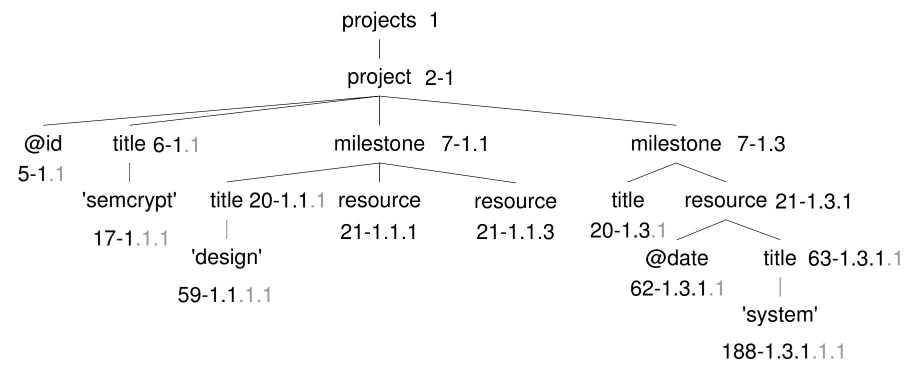


Figure 3.4: Labeling the document of Figure 2.2 with schema labels from Figure 3.3.

Example 3.8 (document labeling) *Figure 3.4 depicts a labeled document tree whose schema labels correspond to the labels of the schema in Figure 3.3. Each label consists of a schema label and an instance label, separated by a hyphen. Instance labels are generated by property (i), whereby the part written in grey can be omitted if additionally using property (ii). Assume that we want to assign a label to the first milestone node. The algorithm takes as input the node, its parent node* `project` *and its previous sibling* `title`*. The path schema of the node has label* 7*. The schema label of the node equals the label of its path schema* 7*. As node schema* `milestone` *is multi-valued, we add a new self label. We take* 1 *as self label because the previous sibling belongs to a different node schema (node schema property). Adding this self label to the parent instance label* 1 *yields instance label* 1.1*. The first milestone node thus receives label* 7–1.1*. We use hyphens to separate the schema label and the instance label, i.e. slabel* = 7 *and ilabel* = 1.1*. The title of this milestone receives label* 20–1.1.1*. When using the cardinality property, the final self label can be omitted as each milestone can only have one title, yielding label* 20–1.1*.*

3.4 Including Type Hierarchies

Existing labeling schemes neglect type hierarchies. In our data model, either each node has explicitly associated a type or it implicitly adopts the type of its node schema, which may be part of a type hierarchy defined in the schema model. The query language used in this thesis supports applying queries to single types as well as to full or partial type hierarchies. To support these queries, we identified the need for indices on type hierarchies (cf. Chapter 2).

The concepts of types, type hierarchies and node schemas can be compared to the concepts of classes, inheritance and attributes in object-oriented databases (OODBs). To support queries on class hierarchies, OODBs distinguish between single-class and class-hierarchy indices [30]. Single-class indices build a separate index for each class. When the query scope refers to a class hierarchy, resolving the query by single-class indices would require to access the index for each class individually. In this case, it is more efficient to build one index for all classes of an inheritance hierarchy. Analogous to single-class and class-hierarchy indices, we can distinguish two labeling approaches:

- Encoding the type directly into the schema label allows for filtering nodes with a specific type and labelpath. However, if type labels depend on labelpaths, types occurring in multiple path schemas will have different labels, which complicates operations on types as well as on labelpaths.
- If labels refer to type hierarchies instead of to single types, the labels only support queries on the type hierarchy, whereas queries on single types depend on additional information, e.g. special index structures.

To support both approaches, we propose to extend the label of a node with an additional type label. This enables the labeling scheme to encode specific types without modifying schema labels. In the following subsections, we extend the labeling approach of Section 3.3 by type hierarchies. Note that in [88], we used a different approach to label types, which makes the type label dependent on the schema label. If a type occurs in various path schemas, the same type has different labels, which complicates handling queries and indices on types. We therefore propose to label types independently in this thesis.

3.4.1 Schema Labeling

Type hierarchy indices need to compare types for index construction and traversal. For this purpose, we assign a unique label to each type defined in the schema model. To support the required operations on types, the labels must allow for (i) evaluating super/subtype relationships between the types of a type hierarchy and (ii) calculating the labels of the supertypes of a type.

Each type hierarchy defined in the schema model forms a tree. If we simply assign a label to each type, the label of a supertype represents both the specific type as well as the partial type hierarchy of which it is the root. For example, when using the label as search key in an index, the label cannot express whether to retrieve the single type or all types of the partial type hierarchy. To solve this problem, we generate a tree of types such that each supertype is abstract (cf. abstract superclass rule [105]). More precisely, we create an additional subtype for each non-abstract type which has subtypes to represent the specific type. When labeling the tree, each leaf label identifies a single type, whereas each inner label refers to a partial hierarchy.

To assign labels, we can use any labeling scheme of Section 3.2 that supports evaluating structural relationships and navigation. Example 3.9 shows a sample labeling, which we use in this thesis. As we assume type hierarchies to be static, we use the k-ary complete-tree labeling scheme [127] to assign labels.

We define function $label : \mathcal{T} \rightarrow \mathcal{Z}$, which returns the label of a type, where $\mathcal{Z}$ is a set of type labels. We further define functions *parent*, *ancestor*, *descendant* and *children* on each type of a type hierarchy, which are analogous to the corresponding functions in the document tree (cf. Subsection 2.1.1).

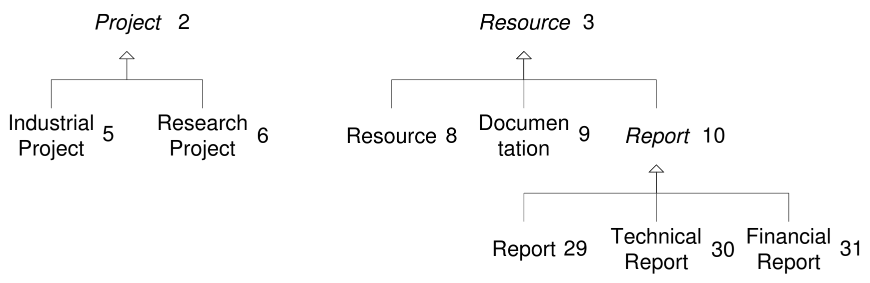


Figure 3.5: Labeling the type hierarchies of the schema in Figure 2.3.

Example 3.9 (labeling type hierarchies) *Figure 3.5 depicts two labeled type hierarchies. As there are two type hierarchies with at most three subtypes per type, we chose $k = 3$. Projects are abstract and are classified into industrial and research projects. Additionally, there are different types of resources and reports. Assume that these supertypes are originally not abstract. We added a subtype to each of these abstract supertypes that represents the specific type. Type label* 10 *thus refers to the partial hierarchy of reports, whereas type label* 29 *refers to all reports, which are neither technical nor financial reports.*

3.4.2 Document Labeling

To encode type hierarchies into labels, we extend each label with a type label. The type label identifies the specific type of the node and therefore must be a leaf of the labeled type hierarchy. The type label does not affect the remaining parts of the label, i.e. the schema and instance label still uniquely identify each node. Further, only nodes whose node schema references a type hierarchy require a type label.

We define partial function $tlabel : N \rightarrow \mathcal{Z}$ to return the type label of a node. Further, we extend function *label* to also incorporate the type label. Function $label : N \rightarrow (\mathbf{Z} \times Z \times \mathcal{Z})$ returns the schema label, instance label and - if existent - type label of a node.

Example 3.10 *In Figure 3.4, assume that the resources of the first milestone have the types `<Report>` and `<Documentation>`, respectively. The first resource has label* 21–1.1.1–29*, where* 29 *identifies the specific type `<Report>`. Note that we use hyphens to separate the individual parts of a label, i.e. the schema, instance and type label. The second resource receives label* 21–1.1.3–9.

The type label enables the comparison of nodes by their type as well as navigation to the supertypes of a type. When navigating to ancestor nodes, the label of a node allows for calculating the schema and instance labels of its ancestor nodes. However, the type labels of ancestors are not encoded into the label. To determine their type labels, access to the document is necessary.

If labels shall also support calculating the types of ancestor nodes, the type label of a node needs to include the type label of all nodes on the node's path. As the schema is defined to be acyclic, it basically allows for determining which type label belongs to which ancestor. To determine the types of ancestors without access to the schema, each type label needs to be associated with its level or schema label. Including type labels of ancestor nodes facilitates navigation, but at the same time increases the label size. Calculating types of ancestors mainly influences query processing. For indexing, it is sufficient to calculate supertypes as well as ancestor labelpaths and paths. In this thesis, we therefore only include the individual type of a node into its label.

Example 3.11 (type labeling) *In Figure 3.4, label* 21–1.1.1–29 *identifies the first resource. When including type labels of ancestor nodes, we require the schema to determine whether the type label* 29 *belongs to the resource or any of its ancestors. If we associate each type label with its level, we get label* 21–1.1.1–4.29 *for this resource. In this case, access to the schema is not required to determine to which node the type belongs. Assume that there are different types of projects and that the sample project is of type* `<ResearchProject>`*. Then the first resource gets the label* 21–1.1.1–2.6, 4.29*. This label allows for determining the labels of all ancestors including their type label without access to the schema or the document.*

3.5 Processing Labels

This section looks at how labels support the required operations on paths, labelpaths and types. After labeling the schema, type-hierarchy and document trees, the vertices of each tree have a unique label. There are different kinds of labels, i.e. schema, type and instance labels. The labeling schemes used to assign the labels directly support the following comparisons and operations on two labels l_1 and l_2 of the same kind (cf. Subsection 3.2.1).

- Document order:
  - $l_1 \prec l_2$ if l_1 precedes l_2 in document order;
- Structural relationships:
  - $l_1 = l_2$ if l_1 and l_2 are equal and reference the same vertex;
  - $l_1 \vdash l_2$ if l_1 is the parent of l_2;
  - $l_1 \Vdash l_2$ if l_1 is an ancestor of l_2;
  - $l_1 \dashv l_2$ if l_1 is a child of l_2;
  - $l_1 \dashv\vert\, l_2$ if l_1 is a descendant of l_2;

  The child and descendant relationships are the inverse of the parent and ancestor relationships, respectively. If l_1 and l_2 are type labels, the structural relationships denote whether a type is a supertype or a subtype of another type. For example, if $l_1 \vdash l_2$, l_1 is the direct supertype of l_2.
- Navigation:
  - Function $parent(l)$ returns the parent label of label l, which enables the recursive calculation of all ancestors of a label;

The labels can also calculate the level of a vertex in the tree. In a prefix label, the level corresponds to the number of individual self labels. While some approaches using the k-ary complete-tree labeling scheme include the level of a vertex into its label (e.g. [180]), it is possible to calculate the level of a k-ary label. Equation 3.1 determines the maximum label n at a certain level. Based on this equation, Equation 3.2 calculates the level of a k-ary label n.

$$n = \sum_{i=0}^{i=level} k^i = \frac{k^{level} - 1}{k - 1} \tag{3.1}$$

$$level(n) = \lceil log_k(n(k-1)+1) \rceil \tag{3.2}$$

Schema and type labels directly support the required operations on label-paths and types, respectively. In contrast, operations on paths need to consider both schema and instance labels due to the node schema property (cf. Subsection 3.3.2), When additionally using the cardinality property, some operations may require access to the schema. In the following, we look at each operation in more detail. Thereby, we describe operations on paths by referring to their last nodes as each node label also uniquely identifies the corresponding path.

3.5.1 Document Order

Determining document order between vertices of the schema or the type tree is directly supported by the labeling scheme. This is illustrated by Equation 3.3, where v_1 and v_2 are vertices of a schema or type tree.

$$v_1 \prec v_2 \Leftrightarrow label(v_1) \prec label(v_2) \tag{3.3}$$

Evaluating document order between nodes or paths requires comparing both the schema and instance label of node labels. When assigning instance labels dependent on node schemas (node schema property), the instance labels do not reflect document order between nodes that have a common ancestor at the intersection path schema of their path schemas. To compare document order, we require functions *ilevel* and *ilabel*.

The intersection path schema between two path schemas $\mathbf{p'} = \langle \mathbf{n'_1} \ldots \mathbf{n'_j} \rangle$ and $\mathbf{p''} = \langle \mathbf{n''_1} \ldots \mathbf{n''_m} \rangle$ is the maximum path schema $\mathbf{p} = \langle \mathbf{n_1} \ldots \mathbf{n_k} \rangle$, where $name(\mathbf{n_i}) = name(\mathbf{n'_i}) = name(\mathbf{n''_i})$ for $i = 1 \ldots k$. The label of the intersection path schema can be calculated by operations on the labels of the two path schemas (cf. Subsection 3.5.3). Function $ilevel(label(\mathbf{p'}), label(\mathbf{p''}))$ returns the level at which the path schemas intersect.

To compare nodes at their intersection path schema, function $ilabel(label(n), ilevel)$ returns the instance label of a node label at a certain level. The returned instance label either equals the instance label of the node or is a prefix of its instance label. When labeling documents based on the node schema property, the function simply needs to return the instance label at the specified level. If additionally using the cardinality property, the level of the instance label need not express the level of its node in the document tree. In this case, access to the schema is required to determine the levels that miss their self labels due to single-valued node schemas.

Determining document order between two nodes n_1 and n_2 works as follows. If both nodes are situated in the same subtree with respect to their intersection path schema, their schema labels express document order, otherwise it is necessary to compare their instance labels. This is shown by Equation 3.4, assuming that $ilevel = ilevel(slabel(n_1), slabel(n_2))$.

$$\begin{aligned} n_1 \prec n_2 \Leftrightarrow (ilabel(label(n_1), ilevel) = ilabel(label(n_2), ilevel) \wedge \\ slabel(n_1) \prec slabel(n_2)) \\ \vee(ilabel(label(n_1), ilevel) \neq ilabel(label(n_2), ilevel) \wedge \\ ilabel(n_1) \prec ilabel(n_2)) \end{aligned} \tag{3.4}$$

Example 3.12 (document order) *Looking at Figure 3.4, assume that we want to compare the document order between nodes* $n_1 = 59\text{–}1.1.1.1$ *and* $n_2 = 21\text{–}1.1.1$*. At their intersection path schema with schema label* 7*, the nodes have the same instance label* 1.1*. We therefore compare the document order of their schema labels. Schema label* 59 *is situated at level 5, whereas schema label* 21 *is situated at level 4. The parent of schema label* 59 *has schema label* 20*. As it is also situated at level 4, we can compare it with schema label* 21*. As* $20 < 21$*, node* n_1 *precedes node* n_2 *in document order. When comparing document order between nodes* $n_1 = 20\text{–}1.1.1$ *and* $n_2 = 21\text{–}1.3.1$*, we get the intersection path schema* 7*. The nodes have different instance labels at this intersection path schema (*$1.1 \neq 1.3$*). Thus, their instance labels express document order, and* $1.1.1 \prec 1.3.1$.

3.5.2 Structural Relationships

Schema and type labels allow for directly evaluating structural relationships between labelpaths (or path schemas) and types, respectively. Comparing nodes or paths depends on the schema and instance labels of the node labels. Equations 3.5-3.7 evaluate structural relationships between nodes n_1 and n_2.

$$n_1 = n_2 \Leftrightarrow slabel(n_1) = slabel(n_2) \wedge ilabel(n_1) = ilabel(n_2) \tag{3.5}$$

$$n_1 \vdash n_2 \Leftrightarrow slabel(n_1) \vdash slabel(n_2) \wedge ilabel(n_1) \vdash ilabel(n_2) \tag{3.6}$$

$$n_1 \Vdash n_2 \Leftrightarrow slabel(n_1) \Vdash slabel(n_2) \wedge ilabel(n_1) \Vdash ilabel(n_2) \tag{3.7}$$

Child and descendant relationships correspond to the inverse of the parent and ancestor relationships, respectively. Note that in case of using the cardinality property to label documents, a child or descendant need not add a new self label. The last part of Equations 3.6 and 3.7 then needs to be adapted to also compare instance labels by equality.

3.5.3 Navigation

Schema and type labels directly support calculating parent labels. Navigating to the parent of a node n or its path requires calculating the parent of both the schema and instance label. Thereby, the following conditions need to be considered. (i) In case of the cardinality property, the instance label only changes if the node schema of n is multi-valued. To determine the cardinality of the node schema, access to the schema is necessary. (ii) If the type label consists of all types of a node's path, calculating the parent of n requires removing the last type label if this type label is situated at the same level as n.

The labels also allow for calculating labels of children in certain cases. Regarding schema and type labels, access to the schema is required to determine whether the calculated labels of children exist. Calculating the child label of a node in the document tree requires the following steps. (i) Access the schema to determine its schema label. (ii) In case that the child node schema is required and single-valued, the child instance label must either add the self label 1 to its parent instance label or remains unchanged when using the cardinality property. Otherwise, access to the document is necessary to determine the instance label. (iii) If the type of the child node schema has subtypes, look up its type in the document.

Example 3.13 (navigation) *The parent of node* 7–1.1 *in Figure 3.4 has schema label* $(7-2)/3+1=2$ *and instance label* $prefix(1.1)=1$*. To calculate the child of node* 2–1 *with node name* `title`*, we first look up the corresponding path schema in the schema, which has label* 6*. The corresponding node schema is single-valued and required. We therefore know that the child instance label is* 1.1 *or - in case of using the cardinality property - it is* 1*.*

3.6 Summary

Labels facilitate processing queries and updates as well as representing and comparing structural properties in indices. The labeling scheme should assign labels to paths, labelpaths and types and support determining document order, evaluating structural relationships and navigating to ancestors via labels. Labels should not change when updating documents. Efficiently encoding labels helps reducing the storage space required for indices.

Existing labeling schemes support path operations, dynamic documents and efficient encoding (e.g. [97, 132]). Schema-aware approaches consider labelpaths [40, 81], but none of them labels type hierarchies to support operations on types.

The labeling scheme presented in this thesis extends existing approaches with schema information. To support the required operations on labelpaths, it constructs a tree of path schemas and labels both the path schemas as well as the type hierarchy trees. To improve query processing, labels assigned to nodes of a document incorporate these schema and type labels. We also discussed reducing the label size, assuming that the schema is small and can be accessed efficiently. The labeling scheme is based on the schema and document model used in this thesis. When assigning labels, it basically abstracts from how to generate the labels as several existing approaches support the identified requirements. As such, the labeling approach used in the examples can easily be replaced.

While the main purpose of the proposed labeling scheme is to support indexing, it also facilitates query processing. Integrating schema information improves query optimization and index selection. The labels enable structural joins and certain navigation without accessing the document when evaluating queries. With regard to indexing, indices can use labels as index keys. To construct and traverse the index structure, they evaluate structural relationships between index keys. Schema information improves determining which updates affect which indices. Labels also accelerate index maintenance by offering support for query processing.

Chapter 4

Index Structures

Contents

This chapter describes the indexing approach SCIENS (Structure and Content Indexing with Extensible, Nestable Structures), which this thesis proposes to provide flexible and selective secondary indices for XML databases. After giving a general introduction to indexing in Section 4.1, Section 4.2 reviews existing index structures for non-XML and XML databases. The main concepts of SCIENS are presented in Section 4.3 and are detailed in the subsequent sections. Section 4.4 addresses extending index structures, whereas Section 4.5 describes the concept of index nesting. Finally, Section 4.6 summarizes the main ideas of SCIENS and contrasts various indexing alternatives. All examples used in this chapter refer to the running example (cf. Figures 2.2 and 2.3 for the document and schema, and Figures 3.3, 3.4 and 3.5 for the corresponding labels). The sample queries are taken from Table 2.2.

4.1 Introduction

XML databases require indices to efficiently evaluate queries on the document content and the document structure. In addition to the primary index structure, which represents entire documents, they need to provide secondary indices adapted to the query workload. These indices must fulfill the requirements presented in Subsection 2.3.

Each index maps index keys to the records (nodes) that are associated with the index keys in the document. For this purpose, the index organizes index keys and records into a file, consisting of a boundary and a data structure. The boundary structure organizes the index keys into pages for efficient retrieval, whereas the data structure contains the records that match the index keys. While traditional secondary indices represent records by pointers to the address of the actual records in the data file, we use node labels instead of physical address pointers. These node labels do not only uniquely identify each node, but also encode structural properties of nodes (cf. Chapter 3). We refer to pages of the boundary structure as index pages and to pages of the data structure as data pages. If a page exceeds a certain size, it needs to be split.

An index can be defined on one or more properties. The number of properties determines the number of index dimensions. When accessing the index, it is possible to specify a search condition for all or partial dimensions. The index then looks for all index keys that match the search conditions and returns associated node labels.

4.2 Related Work

Indices play an essential role in databases. After reviewing index structures in non-XML databases in Subsection 4.2.1, we look at index structures that have been specifically developed for XML in Subsection 4.2.2. Subsection 4.2.3 compares existing indexing approaches with regard to the sample queries of Table 2.2.

4.2.1 Index Structures in Non-XML Databases

This subsection reviews index structures in non-XML databases and looks at which XML indexing requirements they can fulfill. Relational databases use hash tables, B-trees and multidimensional index structures to index values. Object-oriented index structures extend these index structures for representing structural properties, such as aggregation graphs and inheritance hierarchies. Information retrieval techniques use index structures for full-text queries. Object-relational databases propose the concept of extensible indexing to adapt indices to new data types.

One-dimensional Index Structures

The most well-known index structures are hash tables and B-trees, which organize index entries with one index key [83, 164]. Relational databases use hash tables and B-trees to index attribute values. Similarly, XML databases can use hash tables and B-trees to index the document content for exact and range comparisons, respectively.

Hash tables apply a hash function to each index key, which maps the index key to a bucket. The bucket consists of pages holding the index entries. Extendible hashing [75] and linear hashing [139] are dynamic hashing techniques. Hash tables do not support range searches, but only equality comparisons.

B-trees [24] are tree-structured dynamic indices to support range searches. In contrast to hash tables, they sort index keys and organize them into a balanced boundary structure. There exist several variants of the B-tree, such as the B+-tree and the B*-tree [58]. The prefix B-tree [25] is a special kind of B-tree, which uses minimal prefixes as keys to handle characters as index keys more space-efficiently.

Multidimensional Index Structures

Searches often do not only involve one search key. To support queries with several search keys, i.e. queries on multiple dimensions, various approaches exist. They have in common that they index values in each dimension and can therefore be used for indexing node values in XML documents. Multidimensional index structures can be classified according to whether they reuse one-dimensional index structures or develop specific data structures.

When using one index structure for each dimension, a search on multiple dimensions requires to traverse each index structure and to join the results. Multiple-key indexes [83] build an index for one dimension, which recursively points to an index on another dimension, such that there is one (nested) index for each dimension. If the search does not specify a key for the first index, every nested index needs to be visited, which is very time-consuming. Similarly, by simply concatenating keys, only exact match queries are supported efficiently. The UB-tree [23] maps multidimensional points to one dimensional points according to z-order and then stores these points in a B+-tree.

To support exact and range queries on all or partial dimensions more efficiently, several multidimensional index structures have been developed. Gaede and Günther provide an extensive overview of multidimensional index structures in [82].

Tree-based indices. The kd-tree [26] is an unbalanced main-memory data structure generalizing the binary search tree to multidimensional data. Each internal node of the tree splits one dimension into two parts, which are smaller and greater than a certain value, respectively. Several approaches adapt the kd-tree to secondary storage by grouping nodes of the kd-tree into pages.

The KDB-tree [169] is a balanced tree combining ideas from the kd-tree and the B-tree. Each page of the boundary structure consists of regions, which specify ranges for each dimension. For each region, there is a pointer to a page in the next lower lever of the KDB-tree, which further subdivides the region. In the KDB-tree, page splits may result in cascading splits of pages of the data structure, which may result in low storage utilization. Variants of the KDB-tree propose different splitting strategies by representing regions in a kd-tree [203] or by shifting splitting lines [137]. Other examples for tree-based multidimensional index structures are the hB-tree [141] and the Buddy-tree [178].

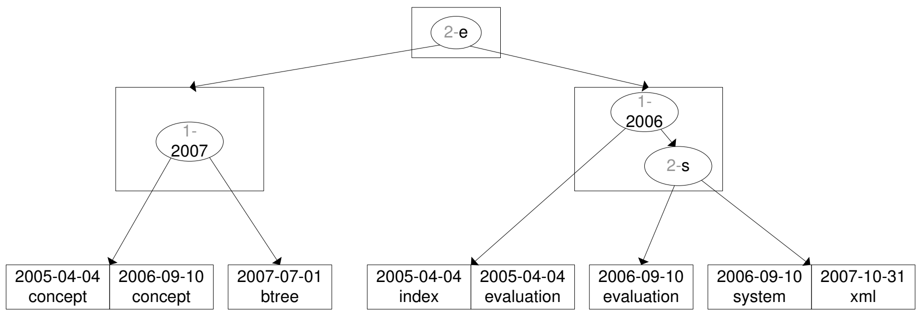


Figure 4.1: KDB-tree on the values of resource dates and titles.

Example 4.1 (KDB-tree) *Figure 4.1 depicts a small KDB-tree on the date and title of resources, which uses a kd-tree to organize regions. Each node in the kd-tree expresses a splitting line, indicating the splitting dimension and the value at which the split occurs. In case of characters, minimal prefixes are used for splitting values. The figure depicts index keys in data pages, but for simplicity it omits data records.*

Hash-based indices. Instead of organizing the boundary structure into a tree, hash-based multidimensional index structures partition dimensions by applying a function on index keys. The grid file [155] partitions each dimension into ranges and maintains a grid directory that provides a mapping between ranges and data pages. The ranges are kept in main memory and enable the index to identify the relevant grids, which in turn point to the relevant data pages. Several variants of the grid file try to reduce the growth of the grid directory and increase its space utilization. Instead of partitioning dimensions, other approaches apply hash functions on dimensions (e.g. [83]). While these hashing techniques are efficient for exact match queries, they are not adequate for range queries.

Bitmap indices. A different multidimensional index structure is the bitmap index [83, 157]. It uses one bit vector for each index key, representing the position of records with that index key. To combine index results, it can take the bitwise AND and OR of bit-vectors without having to access the records themselves.

Object-oriented Index Structures

In the object-oriented data model, objects are instances of classes, which define instance attributes. The value of an attribute can be an object or a set of objects belonging to the same class. Class definitions form a directed graph, which is referred to as *aggregation graph.* A path is a branch in the aggregation graph, whereas a path instantiation is a sequence of nested objects on a path. A class can be a specialization of one or more classes, resulting in an *inheritance hierarchy.* In contrast to relational databases, object-oriented databases require special index structures to support queries on the aggregation and/or inheritance dimension (cf. [30]).

The concepts of classes, class hierarchies and attributes are comparable to types, type hierarchies and node schemas in XML. Objects, paths and path instantiations resemble nodes, labelpaths and paths, respectively. To index these structural properties, we will look at how XML databases can reuse ideas from object-oriented indexing in the following.

Indices on aggregation graphs basically index objects on values of nested objects to find all path instantiations for a given path. They either build one index for each class on a path (multi-index [144], join index [191]) or collect all path instantiations in one index to reduce the number of index accesses (path index and nested index [29]). The path index returns all path instantiations, whereas the nested index provides a direct association between objects at the end and at the beginning of a path. While most indices can evaluate a path in reverse order, some indices also support forward traversal (e.g. join index).

With regard to XML, indices on aggregation graphs can support navigation in XML documents. While forward traversal corresponds to navigation to descendants, evaluating a path in reverse order is comparable to navigating to ancestor nodes. The tree-structure of XML documents allows for navigating to ancestors by operations on labels instead of by accessing index structures (cf. Chapter 3). While it is therefore not necessary to support reverse traversal by returning all nodes on a path, indices on aggregation graphs can be used to index nodes on values and labelpaths and to support navigation to descendants.

index	7-1.1, 7-2.3,...
XML	7-1.3, 7-2.3, 7-5.7,...
...	...

Figure 4.2: Nested index mapping title values to milestone nodes.

Example 4.2 (nested index) *Figure 4.2 depicts a nested index on the partial labelpath `milestone/resource/title/text()`, mapping each distinct resource title to the corresponding milestone nodes.*

Indexing inheritance hierarchies. In object-oriented databases, queries either refer to a single class or to a (partial) class hierarchy. Similarly, queries on the tree-structure of XML documents can be expressed as hierarchical queries. To support such queries, object-oriented indices either reuse one- or multidimensional index structures or develop proprietary index structures.

The first proposals for indexing both attribute values and inheritance hierarchies are based on B+-trees. The CH-index (class-hierarchy index) maintains a unique B+-tree on the attribute value of all classes of the indexed hierarchy, while the SC-index (single-class index) creates a separate B+-tree for each class in the indexed hierarchy [118]. The main difference between these indices is the kind of grouping, which influences their query performance. Key-grouping indices, such as the CH-index, first divide index entries by their attribute value, whereas type-grouping indices first group objects according to their type. For exact match queries, key-grouping is the better choice, while for range queries type-grouping structures have better performance [117]. Several approaches aim at combining the advantages of key and type-grouping by building specialized tree-structured indices (H-tree [160], CG-tree [117], hcC-tree [183]). Instead of building complex tree structures, the CD-Index [166] and the Index Set [11] try to find the optimal sets of classes which to index.

Indexing inheritance hierarchies can also be reduced to a two-dimensional range search problem, whereby one dimension is the class and the other is the value. Instead of maintaining complex tree structures, it is therefore possible to use multidimensional index structures. One difficulty in this approach is that the class domain is small and unordered, while multidimensional index structures have been designed for large, ordered key domains. To address this problem, the MT-Index [153] maps the class hierarchy to a linear order of classes. As the class domain is small in comparison to the attribute domain, the x-tree [46] and the 2D-CHI index [126] propose splitting strategies based on the query workload.

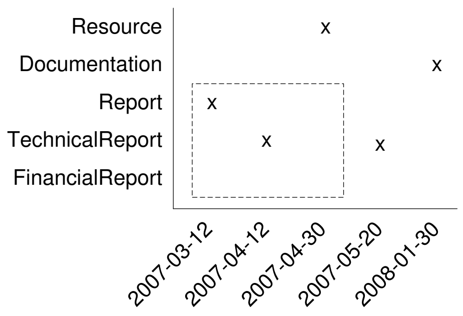


Figure 4.3: Type and value dimension of an index on resource types and date values.

Example 4.3 (type and value dimension) *An index on the date and type of resources has a value dimension and a type dimension, as depicted in Figure 4.3. The dashed rectangle shows a sample search space asking for all reports written between '2007-03-12' and '2007-04-30'.*

While these indices assume that the indexed attribute belongs to the indexed class hierarchy, indices which integrate aggregation graphs and inheritance hierarchies support indexing all class hierarchies on a path (e.g. nested-inherited index [28]).

Information Retrieval

Information retrieval (IR) [20] aims at retrieving documents that are of interest for a user. Query results do not only include documents that exactly contain the keywords in the user query, but rank the documents according to their relevance. To speed up query processing, IR systems index the words of documents. To reduce the number of indexed words, common techniques include elimination of stop words, such as articles, and stemming, which refers to reducing words to their grammatical root. Typical indices used are inverted lists, signature files and suffix arrays. Other data structures, such as the Patricia Trie [152], have been developed for use in main memory.

An inverted list consists of a vocabulary, which contains all the distinct words in a document. Each word points to a list of occurrences, referring to the positions of the word in the document. By searching the vocabulary for a specific keyword, the inverted list can determine all documents containing that keyword. The use of e.g. a B-tree on the vocabulary speeds up scanning the vocabulary. By processing the list of occurrences, inverted lists enable phrase or proximity queries.

Information retrieval indices have also been extended to take into account simple structural information about a document. For example, an inverted list can represent the position of a word in terms of its path [170]. Such indices can be applied to XML indexing by primarily enabling to specify search conditions on the document content, but also taking into account simple structural conditions, such as the name or labelpaths of nodes returned.

Extensible Indexing

New applications, such as geographic information systems, document libraries and multimedia systems, require databases to process new kind of data. Instead of relational data, databases need to support domain-specific data types as well as specific operations on these types. Efficient search in such data requires specialized index structures. For each domain, a large number of specific index structures has been proposed (e.g. spatial and text indices). As a database cannot implement all of them, object-relational, extensible and component databases [69] provide the ability of integrating new index structures. Generally, there are two approaches to extensible indexing:

- Provide an index interface that lets users implement their own index structures by implementing methods for index definition, maintenance and search. For example, this approach is supported by Oracle data cartridges [184], DB2 database extenders [64] and Informix data blades [32]. Their main disadvantage is that they require high implementation effort as each index structure needs to be implemented from scratch.
- Apply existing index structures to new data types and only leave type and operator specific implementations to the user [185]. For example, B+-trees can index any data type with a linear ordering. The generalized search tree (GiST) [101] is such a generic template index that can be extended to support new data types. It can represent balanced trees, such as B+-trees and R-trees, and index arbitrary data types by implementing six methods for each new type.

4.2.2 XML Index Structures

Various index structures have been proposed for XML. Vakali et al. [190] present an overview of these indices. In contrast to relational databases, XML databases need to provide indices on the document structure, in the same way as object-oriented databases. Additionally, they need to support full-text queries similar to IR systems. Thus, we basically distinguish structure-oriented and content-oriented indexing. There also exist so-called hybrid approaches which try to support structure- and content-oriented queries.

Structure-oriented Indexing

Structure-oriented indices summarize the structure of XML documents. They can basically be classified into path, element and sequence indices. While path indices refer to (parts of) labelpaths, element indices are defined on node names and require structural joins to reconstruct the document structure. Sequence indices represent both documents and queries as sequences.

Path indices[1] summarize the labelpaths which exist in a document and associate with each labelpath the nodes which can be reached along the labelpath. They can be classified according to the following characteristics.

- *Total/partial summary indices:* The most well-known representative of these indices is the DataGuide [85], which is a structural summary of all the labelpaths in a document. It can retrieve all nodes belonging to a certain labelpath. While the strong DataGuide is a total summary index, covering all labelpaths starting from the root, partial summary indices only index the most common labelpaths to reduce index size. For example, the T-index [150] only indexes path templates that are frequently queried. The A(k)-index [113] uses k-bsimilarity to store exact answers only to simple paths of length up to k. The M(k)-index [100] and the D(k)-index [162] extend the A(k)-index to be workload aware and build an adaptive structural summary. Apex [56] indexes frequently queried paths to improve query performance.

- *Simple/branching path indices:* The DataGuide covers all simple path queries, but cannot directly support branching path queries. The F&B-index [111] can evaluate branching path queries without having to decompose the query. While the F&B-index is the smallest index that covers all branching path queries, its size may grow rapidly. Similar to simple path indices, the F&B-index can reduce its size by restricting the set of branching path queries to be indexed.

- *Content-aware path indices:* Pure structural indices are not efficient for queries on the document content. In addition to the document structure, several approaches integrate node values into indexing. Basically, they build one value index for each labelpath by associating with each labelpath of the DataGuide an inverted file [121] or a B-tree [128]. The Index Fabric [59] proposes a balanced storage structure for Patricia Tries on labelpaths and values. All these indices require traversing the structural summary before being able to evaluate predicates on the document content. To avoid traversing parts of the structural summary that do not contain the queried value, some approaches propose to associate signatures with each labelpath [52, 195].

Several variants of these path indices exist, which improve processing partial matching queries. Supex [194] builds a so-called element map, which maps each node name to all the nodes in the structural summary with the corresponding name. Barg and Wong [21] construct an inverted list for each node name, which groups nodes according to their labelpath.

[1]The term path actually refers to the labelpath.

Path indices are comparable to indices on aggregation graphs in object-oriented databases. Similar to object-oriented indices, they typically return node identifiers that do not express structural relationships between nodes. To evaluate predicates on internal nodes, they would therefore need to represent all nodes on a node's path, in the same way as object-oriented indices, or use a labeling scheme.

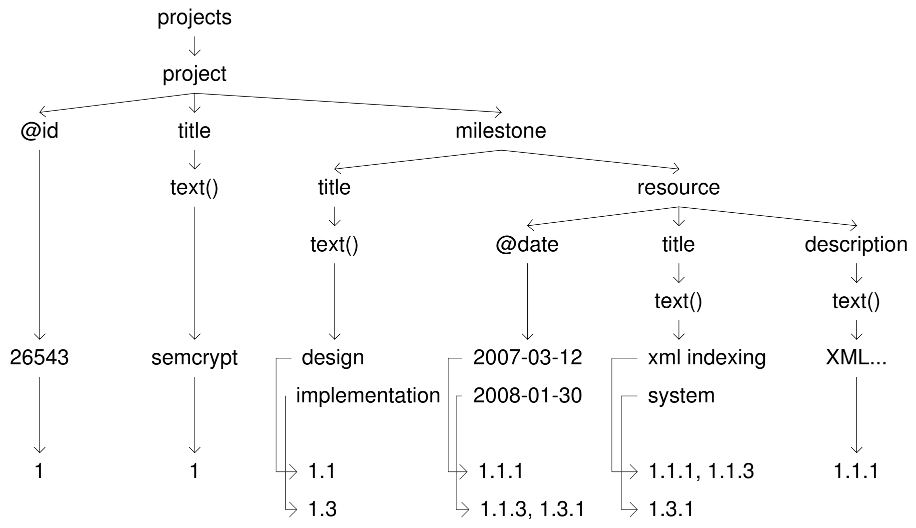


Figure 4.4: Content-aware path index.

Example 4.4 (path index) *Figure 4.4 depicts a sample content-aware path index that is based on our labeling scheme. For each labelpath, there is an inverted list on values, which points to the node labels with the corresponding labelpath and value. The schema labels are not included as they can be inferred from the labelpath. The index is efficient for simple path queries starting at the root node, e.g.* `/projects/project/title` *(Q1). Partial matching queries, such as* `//title` *(Q3), require traversing the whole structural summary. Answering a content-oriented query, such as* `/projects/project[title='semcrypt']` *(Q5), requires traversing the structural summary and the (nested) value index. The query contains a predicate on an internal node. As the sample index returns node labels, it is possible to navigate from the matching text node label (17–1) to the corresponding project node (2–1), which the query returns.*

To summarize, path indices are efficient for simple total matching queries. Variants exist to support branching path queries or partial matching queries. Content-aware path indices can answer queries on the document content, but also favor queries on specific labelpaths. Path indices that are not based on labeling schemes do not directly support predicates on internal nodes.

Element indices[2] build an index on node names. In contrast to path indices, they use labeling schemes to be able to evaluate structural relationships between nodes. To process a query, they access the index to determine candidate nodes and then reconstruct structural relationships between the nodes

[2]Element indices are also referred to as structural join indices.

by structural join algorithms. While XISS [135] requires accessing each node with the queried node names, several enhancements have been proposed to skip nodes that cannot match the query. When building a B+-tree on node names and labels, searching the descendants of a node corresponds to a range search in the B+-tree, which is efficient in skipping descendants that have no matches in a join [55]. Chen et al. [49] propose to enhance the B+-tree with sibling pointers to skip siblings. The XR-tree [109] extends the B+-tree with stab lists to support skipping ancestors and descendants in structural joins. These indices can support content-oriented queries by building an additional index on node values. Processing structural joins can also be improved by labeling schemes (cf. [142, 174]).

@id	5-1
@date	62-1.1.1, 62-1.1.3, 62-1.3.1
milestone	7-1.1, 7-1.3
title	6-1, 20-1.1, 63-1.1.1, 63-1.1.3, 20-1.3, 63-1.3.1
...	...

26543	5-1
2007-03-12	62-1.1.1
2008-01-30	62-1.1.3, 62-1.3.1
design	59-1.1
...	...

Figure 4.5: Index on node names (left) and values (right).

Example 4.5 (element index) *An inverted list on node names and values (cf. Figure 4.5) is efficient for partial matching queries. For example, the index can directly retrieve all titles,* `//title` *(Q3). A simple path query, such as* `/projects/project/title` *(Q1), involves retrieving all titles and then filtering the ones that do not belong to projects. In our example, the schema labels can be used to filter the desired nodes. Without a schema-aware labeling scheme, it would be necessary to retrieve all project nodes and perform structural joins between title and project nodes. Answering a content-oriented query (e.g.* `//*[title='xml']` *(Q6)), involves one index access for each queried node name and each queried value and then joining the results.*

Element indices are efficient for partial matching queries. They do not represent structural relationships between nodes in the index, but use labeling schemes and join algorithms instead. Therefore answering total matching queries may require multiple index accesses and a large number of joins. Element indices support content-oriented queries by handling node values in the same way as node names.

Sequence indices [167, 192] transform the document and the query into sequences and process queries based on subsequence matching. They can answer branching path queries without the need of decomposing the query into subqueries. They do not use labeling schemes and only support equality comparisons.

Content-oriented Indexing

Content-oriented indices extend IR techniques, such as inverted lists, to capture the document structure. These extended inverted lists do not simply return nodes with a specific keyword, but additionally group these nodes according to their node name or labelpath [70]. To process the nodes returned by indices, they either require representing all nodes on a node's path similar to object-oriented indices, or make use of labeling schemes [174]. Additionally, there exist various models for approximate matching and ranking results with regard to both values and structural conditions (e.g. [15, 40, 196]).

Content-oriented indices favor content-oriented queries. They are either more efficient for total or partial matching queries dependent on whether they group nodes according to labelpaths or node names, respectively.

Hybrid Indexing

Hybrid indexing techniques support querying the content and structure of XML documents by either building multidimensional indices or by constructing several indices.

To avoid favoring structure- or content-oriented queries, multidimensional XML indices represent documents by three dimensions - paths, values and document identifiers. They use multidimensional index structures, such as UB-trees [22] or bitmap indices [202], which can process queries without favoring a particular dimension.

Building indices on both the document structure and the document content naturally enables the processing of various kinds of queries. This approach, which has been proposed for the semi-structured database Lore [147], has also been adopted to XML (e.g. [51, 168]). Shimizu and Yoshikawa [179] build a B+-tree on labelpaths and a prefix B+-tree on node values to support structural and full-text queries.

Similarly, various native XML databases integrate indices into query processing, e.g. eXist [148], Berkeley [47], Timber [106] and Tamino [175]. They support simple structural and value indices by specifying either the name of the nodes to be indexed or their labelpath. As index structures, they generally use hash tables to index node names or labelpaths and B-trees to index node values. They do not offer indexing support for more complex queries.

4.2.3 Comparison

Relational databases use hash tables, B-trees and multidimensional index structures, which enable the database to index values for exact and/or range comparisons. In contrast, XML databases require indices on the document structure and the document content. Structure-oriented indices mainly focus on labelpaths and node names. While path indices are efficient for retrieving all nodes with a certain labelpath, element indices are more appropriate for partial matching queries, retrieving nodes with a certain node name. Sequence indices support branching path queries, but require a specific representation of the document. Content-oriented indices extend inverted lists and enable full-text queries. Hybrid approaches try to support both structure- and content-oriented queries with multidimensional index structures or by constructing several indices.

Element indices and most of the hybrid approaches that use several indices are based on labeling schemes and return node labels. These node labels enable the database to evaluate structural relationships and join nodes returned by indices. In contrast, path indices are not based on labels, which makes it difficult to evaluate predicates on internal nodes without accessing the document. Without labels, they would require representing all nodes on a node's path for this purpose, similar to object-oriented indices on aggregation graphs.

	DataGuide [85]	A(k) index [113]	F&B index [111]	Index Fabric [59]	XISS [135]	PRIX [167]	Inverted list [174]	Hybrid indices [179]
Q1	$\oplus$	$+$	$+$	$\oplus$	$+$	$+$	$-$	$\oplus$
Q2	$+$	$\oplus$	$+$	$+$	$+$	$+$	$-$	$+$
Q3	$+$	$+$	$+$	$+$	$\oplus$	$+$	$-$	$+$
Q4	$\ominus$	$\ominus$	$\oplus$	$+^1$	$+$	$\oplus$	$-$	$+$
Q5	$\ominus$	$\ominus$	$\ominus$	$\oplus^1$	$+$	$+$	$\oplus^1$	$+$
Q6	$\ominus$	$\ominus$	$\ominus$	$+^1$	$-$	$+$	$\oplus^1$	$+$
Q7	$\ominus$	$\ominus$	$\ominus$	$+^1$	$+$	$\oplus$	$+^1$	$+$
Q8	$\ominus$	$\ominus$	$\ominus$	$+^1$	$\oplus$	$\ominus$	$+^1$	$+$
Q9	$\ominus$	$\ominus$	$\ominus$	$+^1$	$+$	$\ominus$	$+^1$	$+$
Q10	$\ominus$	$\ominus$	$\ominus$	$+^1$	$-$	$\ominus$	$+^1$	$+$
Q11	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
Q12	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\oplus^1$	$\oplus$
Q13	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$

$\oplus$	specific strength compared to other indices
$+$	supported
$-$	possible but inefficient
$\ominus$	not directly supported
1	only supported if index returns node labels for processing structural joins and/or navigating to ancestors

Table 4.1: Support for queries of Table 2.2 offered by existing index structures.

Example 4.6 (comparison) *Table 4.1 contrasts existing XML indices and the sample queries of Table 2.2. For each index category, one representative is chosen for comparison. Structure-oriented indices are sufficient for path queries, whereas various indices help evaluating simple content-oriented queries. Hybrid approaches can process a large number of queries as they build several indices. Nevertheless, they also miss efficient support for more complex queries. For example, evaluating Q10 (`//project[@id='26543']/milestone[title = 'design']/resource[@date≥ '2008-01-01']`) with existing indices results in a large number of joins, as no index can directly process the query. Current XML indices do not address queries on the type hierarchy (Q11 and Q13).*

Table 4.1 shows that each index is more appropriate for different queries and no index supports all queries. With regard to the requirements of Subsection 2.3, existing XML indices support queries on the labelpath hierarchy as well as on the document content. Element indices can support queries on the path hierarchy, but may require a large number of structural joins for this purpose. Indexing type hierarchies has not been regarded yet. For this purpose, it is possible to adapt object-oriented indices on inheritance hierarchies. Efficient support for complex queries is also missing and can currently be achieved at most by hybrid approaches. Although many approaches extend well-known index structures, such as hash tables and B-trees, adapting the concepts of extensible indexing to XML has not been studied yet. Instead, each approach proposes its own index structure and algorithms.

4.3 Concepts

This section presents the main concepts of our indexing approach SCIENS (Structure and Content Indexing with Extensible, Nestable Structures), whose main ideas we have outlined in [87]. Its primary purpose is to provide secondary indices for XML databases that efficiently support evaluating arbitrary query workloads. To support queries on XML documents, we identified the main requirements for secondary indices in Subsection 2.3. Flexibility refers to the ability of indexing structure- and content-oriented properties and of supporting various operations on these properties. Selectivity denotes the ability of defining indices on document fragments instead of on entire documents only.

Existing XML indexing approaches fail in providing flexibility and selectivity (cf. Subsection 4.2.2). They propose proprietary structures, each of which better supports indexing a different property and operation on that property. Further, each index is defined on the whole document, making the index large in size. Consequently, an XML database has to implement a large number of index structures to support arbitrary query workloads with existing approaches. In case of new requirements, it needs to implement new index structures, which is infeasible in general.

The main objective of SCIENS is to provide the flexibility of selectively indexing frequently queried document fragments with only a small number of index structures. In SCIENS, each index has one or several dimensions and can index one property in each dimension. The indexed property can be the value, name, labelpath, path or type of a node. The domain of a dimension consists of all values of the indexed property. For example, the domain of a type index refers to all types of nodes on which the index is defined.

Each index consists of a set of index entries that map a list of index keys, extracted from the domain values, to node labels. Each index entry specifies one index key for each dimension. An index structure consists of a data structure and algorithms to traverse and update the data structure. It organizes the index keys into pages for quickly retrieving all node labels associated with the search conditions. To evaluate search conditions, the index structure needs to compare search keys with index keys. For index construction and maintenance, it has to compare existing index keys with the index keys to be inserted, deleted or updated.

SCIENS makes use of the following two concepts, which are described in more detail in the subsequent sections:

- *Extensibility:* Each index structure performs operations on index keys and search keys for constructing and traversing its data structure. By binding comparison operators to indexed properties, the same operation can be performed on various properties. An extensible index structure can index those properties on which its operations can be performed and it can evaluate those search conditions that can be expressed with its operations. It need not know which nodes it indexes or how indexed nodes relate to each other.
- *Nestability:* Index nesting refers to placing one index structure beneath another index structure. Dependent on the nesting order, it can favor specific query workloads, e.g. content- or structure-oriented queries. Compared to existing XML indexing approaches, SCIENS provides more flexibility with regard to how to nest index structures to adapt them to the query workload.

To provide *flexibility*, SCIENS abstracts from specific properties indexed and adapts operators to indexed properties. Each index allows for specifying one search condition for each dimension and returns all node labels that match all search conditions. SCIENS distinguishes simple and range search conditions. A simple search condition specifies one search key v, expressing the queried value, and one operator $\circ$. A domain value d fulfills the search condition if $v \circ d$ evaluates to true. A range search condition consists of two search keys v_1 and v_2 that express a value range based on two operators $\circ_1$ and $\circ_2$, where $\circ_1, \circ_2 \in \{<, \leq\}$. The first value represents the lower bound, whereas the second value refers to the upper bound of the search condition. A domain value d fulfills a range search condition if $v_1 \circ_1 d \circ_2 v_2$ returns true.

Dependent on the indexed property, search conditions use different operators for comparing search values with domain values. Table 4.2 depicts supported operators for content- and structure-oriented properties. SCIENS assumes that structural indices serve for grouping index entries according to structural conditions rather than for supporting navigation. Therefore, it currently only supports evaluating whether a search key is part of an indexed hierarchy. It is however possible to extend SCIENS to support more properties and operators.

property	simple search	range search
structure	$=, \Vdash$	
content	$=, \sim, \dot{\sim}$	$<, \leq$

Table 4.2: Simple and range operators on structure- and content-oriented properties.

SCIENS supports *selectivity* by abstracting from what an index is defined on or how indexed nodes relate to each other. This is only relevant when selecting indices for queries or generating index entries for index maintenance. An extensible index simply compares search conditions with domain values and can therefore index arbitrary document fragments. It need not know which nodes it indexes as it only operates on index entries.

In the following, we abstract from what an index returns and define indexed fragments only textually. Defining and processing indices will be detailed in

Chapter 5. We further do not specify whether node labels are stored together with index keys or in separate pages, which is implementation-dependent.

4.4 Extending Index Structures

In this thesis, we have selected three index structures, namely hash table, B+-tree and KDB-tree. Each of these index structures can be extended to support various properties and operations on these properties. The concepts of this thesis are however not restricted to these specific index structures, but can be conferred to other index structures as well (e.g. other multidimensional indices, GiST). Concerning the hash table, any hashing technique can be used (cf. Subsection 4.2.1). As B+-tree and KDB-tree, we use the prefix B+-tree [25] and the KDB-tree that represents regions as kd-trees [169, 203], respectively.

Extending index structures does not require changing their algorithms. We only need to ensure that their operations, which they require to compare index keys and search keys, can be performed on arbitrary properties (cf. Table 4.3). While the hash table only supports the equality operator, the B+-tree and the KDB-tree additionally compare the order of index and search keys.

operation	hash table	B+-tree	KDB-tree
$v = k$	×	×	×
$v < k$		×	×

Table 4.3: Operators required by index structures.

The hash table is a one-dimensional index structure, which represents each distinct domain value as index key. Applying a hash function to an index or search key yields a hash value that identifies the bucket of relevant pages. It then only needs to find the index key k that equals search value v in the page. As the hash table only supports exact comparisons, it only requires the $=$ operator.

The B+-tree is a one-dimensional index structure that orders index keys and constructs a boundary structure to search index keys. To traverse the boundary structure and identify relevant index keys, it requires to compare values by their order. The prefix B+-tree uses minimal prefixes in index pages for index keys that are strings.

The KDB-tree is a multidimensional index structure. In contrast to the B+-tree, it splits pages according to different dimensions. As there is no global ordering between the index entries, it cannot link data pages. There is only a local ordering within the pages, which determines the pointers to be followed from the root of the tree to the data pages when traversing the boundary structure. Thereby, it compares values by their order similar to the B+-tree.

While hash tables only support simple search conditions, the B+-tree and the KDB-tree also support range search. Evaluating range search conditions requires comparing ranges with index keys, as shown in Equation 4.1. A range $(v_1, \circ_1, \circ_2, v_2)$ equals an index key k if k is within the range. A range is smaller than k if its lower bound is smaller than k. If a range is smaller than k, it contains index keys that are smaller than k.

$$\begin{aligned}(v_1, \circ_1, \circ_2, v_2) = k &\Leftrightarrow v_1 \circ_1 k \circ_2 v_2 \\ (v_1, \circ_1, \circ_2, v_2) < k &\Leftrightarrow v_1 < k\end{aligned} \tag{4.1}$$

Each index structure can index the properties that support the required operators. It supports the search operations that can be mapped to exact or range search conditions. In the following, we look at how to extend the index structures to index and query content-oriented properties (Subsection 4.4.1), structure-oriented properties (Subsection 4.4.2) and both kinds of properties (Subsection 4.4.3).

4.4.1 Content Indexing

Content indices need to support exact match, range and text queries on node values. The most appropriate index for exact match queries is the hash table, whereas the B+-tree is efficient for range queries. The hash table needs to compare node values by equality only. Dependent on the data type, the B+-tree needs to compare the numeric or lexicographic order of node values. In case that a range search condition does not specify a lower or upper bound, we use an infinite value ∞ instead.

Example 4.7 (exact match and range index) *To support query Q5 (`/projects/project[title='semcrypt']`), we can create a hash table on the values of project titles. A B+-tree on the date of resources supports Q8 (`//resource[@date≥'2007-01-01' and @date<'2008-01-01']`) as it allows for retrieving all resources within a specific date range.*

These index structures can also support text queries by splitting each indexed node value into its words and indexing the individual words, in the same way as an inverted list. Thereby, applying IR indexing techniques, such as stemming and eliminating stop words, reduces the index size. Each word represents an index key. A hash table on the individual words can retrieve all node labels associated with values in which a specific word appears.

A prefix B+-tree additionally supports searching values with a specific word prefix, as each word prefix search corresponds to a range query. More precisely, searching a specific word v in a B+-tree corresponds to the range search $(v, \leq, \leq, v)$. Let ∞ be larger than any letter. Then the range search $(v, \leq, \leq, v{+}\infty)$ expresses a word prefix search, where v is the word prefix and $v{+}\infty$ concatenates the largest letter ∞ to the word prefix.

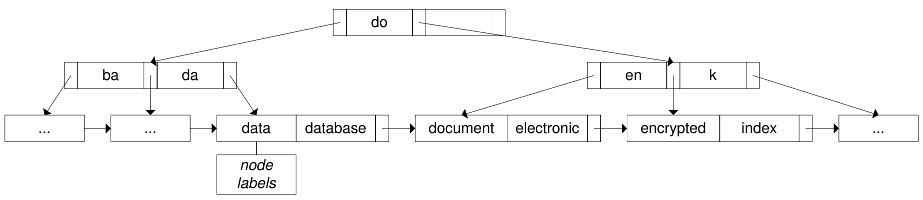


Figure 4.6: Prefix B+-tree on the values of resource descriptions.

Example 4.8 (text index) *To support Q12 (`//resource[description` $\sim$ `'database']`), we can create a prefix B+-tree on the values of resource descriptions. Figure 4.6 depicts a corresponding prefix B+-tree. Assume that we want to index the node value '`The database stores data with an index`'. Splitting the node value into words and applying stemming and elimination of stop words yields the words '`database`', '`store`', '`data`' and '`index`'. These words represent the index keys of the prefix B+-tree. To guide the search through the boundary structure, the prefix B+-tree uses minimal prefixes. For example, the minimal prefix between '`electronic`' and '`encrypted`' corresponds to '`en`'. Retrieving all node labels associated with the word '`database`' corresponds to a standard B+-tree search. Evaluating a word prefix search on '`data`' corresponds to the range search condition ('`data`', $\leq$, $\leq$, '`data`∞'). Starting from the root of the B+-tree, we first follow the first pointer ('`data`' $<$ '`do`') and then the third pointer ('`da`' $\leq$ '`data`'). Both the index keys '`data`' and '`database`' fulfill the search condition ('`data`' $\leq$ '`database`' $\leq$ '`data`∞') and their node labels are returned. As the index key '`document`' does not fulfill '`document`' $\leq$ '`data`∞', the remaining index keys do not match the search condition.*

This thesis only regards word and word prefix search. However, we can easily extend our approach to also support more complex text queries, such as phrase search and approximate queries. For this purpose, the index needs to take into account the positions at which words occur in node values and associate the positions with the node labels. Evaluating a phrase query requires accessing the index for each word of the queried phrase and then determining the index entries in which the words appear in the required order.

Example 4.9 (phrase search) *Assume that we want to evaluate the phrase search '`encrypted document`' with the index of Figure 4.6 and that the index maps the index key '`encrypted`' to the node label* 191–3.5.1 *with positions* 3 *and* 5 *and the index key '`document`' to the node labels* 191–3.3.5 *with position* 2 *and* 191–3.5.1 *with position* 6*. The node label* 191–3.5.1 *has associated a node value in which the word '`encrypted`' appears directly before the word '`document`'. It therefore matches the search condition.*

A multidimensional index, such as the KDB-tree, can index several value domains in one index. It supports multiple search conditions on different domains without the need of first evaluating each search condition and then joining the results. By using minimal prefixes as index keys for strings in index pages, the KDB-tree can also support word prefix search in the same way as the prefix B+-tree. In case of using the KDB-tree for phrase search, it needs to consider that each index key occurs at different positions.

Example 4.10 (multidimensional index) *We can build a KDB-tree on the date and title of resources to support query Q7 (`//resource[@date='2007-10-31'][title='xml']`) (cf. Figure 4.1). The sample KDB-tree can evaluate the range search condition ('`2007-10-31`', $\leq$, $\leq$, '`2007-10-31`') on the date dimension and the range search condition ('`xml`', $\leq$, $\leq$, '`xml`') on the title dimension. Starting from the root of the KDB-tree, we first follow the right pointer, as '`e`' $\leq$ '`xml`'. The corresponding index page first divides the index entries according to their date, whereby we follow the right pointer ('`2006`' $\leq$ '`2007-10-31`'), and secondly according to the title. As '`s`' $\leq$ '`xml`', search* *again follows the right pointer, which yields the requested index entry with keys '`2007-10-31`' and '`xml`'.*

4.4.2 Structure Indexing

Indices on the document structure need to support queries on node names, labelpaths, paths and types. Similar to existing XML indexing approaches, SCIENS uses hash tables to retrieve all nodes with a specific node name or labelpath. As each property supports the equal operator, the extensible hash table[3] can index any property of the XML data model.

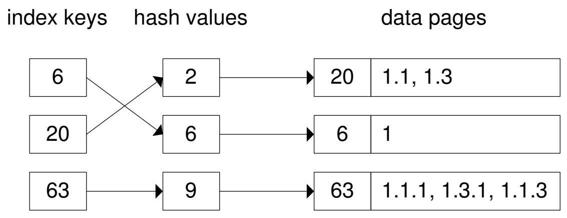


Figure 4.7: Hash table on the labelpaths of titles.

Example 4.11 (labelpath index) *Figure 4.7 depicts a hash table on the labelpaths of nodes with the node name* `title`*. Each labelpath is represented by its schema label. E.g. schema label* 6 *identifies labelpath* `/projects/project/title` *and schema label* 20 *identifies labelpath* `/projects/project/milestone/title`*. The labelpaths represent the index keys and are mapped to hash values. In this example, the hash function simply calculates the checksum of the schema label representing the labelpath. Each data page consists of the index entries whose labelpaths have the same schema label. The index can be used for evaluating queries Q1 (*`/projects/project/title`*), Q2 (*`//milestone//title`*) and Q3 (*`//title`*). As it only supports exact comparisons, the labelpaths need to be extracted from the schema before accessing the index. Evaluating queries Q2 and Q3 requires accessing the index for several labelpaths. Note that in the rest of this thesis, we omit hash values when graphically representing hash tables to keep figures simple.*

A hash table does not consider any relationships between index keys and it can only answer queries which can determine all index keys in advance. This is efficient for retrieving e.g. all nodes with a specific node name, labelpath or type. However, it does not allow for evaluating queries that return all nodes of a subhierarchy. This subhierarchy can refer to a specific part of the schema, a subtree of the document or to a partial type hierarchy.

To support queries within hierarchies, indices have to evaluate structural relationships between search and index keys. Labelpaths, paths and types form trees in XML. The preorder traversal of such a tree establishes a linear order between its vertices, which can be used for comparing these properties in an

[3]In this thesis, the term 'extensible' does not refer to the hashing technique, but to the ability of indexing various properties with a hash table.

index. Each index key can be represented by a label that supports evaluating structural relationships.

SCIENS assumes that index keys represent leaves of the indexed hierarchy, i.e. there are no ancestor-descendant relationships between the index keys of an index. A search key can either be an index key or an ancestor of the index keys, whereby the latter case expresses a hierarchical query. A hierarchical query retrieves all descendants of a search key. As SCIENS assigns labels in document order, a hierarchical query corresponds to a range query on labels. It can be evaluated by comparing the labels of the search key with the index keys. Therefore, it is possible to use standard range indices, such as the B+-tree or the KDB-tree, to index structural properties.

Range search conditions contain operators $<$ and $\leq$. To compare structural properties, range indices therefore have to adapt the operators for equality and order. More precisely, they need to (i) compare equality between search and index keys by evaluating structural relationships between labels and (ii) compare order between structural properties via the document order of labels. An index key matches a search key if their labels equal of if the index key is a descendant of the search key. A search key is smaller than an index key if it precedes the index key in document order. Equation 4.2 shows how to compare a search key v with an index key k with respect to their labels v_l and k_l.

$$\begin{aligned} v = k \Leftrightarrow v_l = k_l \vee v_l \Vdash k_l \\ v < k \Leftrightarrow v_l \prec k_l \end{aligned} \tag{4.2}$$

Evaluating search conditions works as follows. The simple search condition $(v, =)$, looking for a specific value, corresponds to the range search condition $(v, \leq, \leq, v)$. A hierarchical search, expressed by $(v, \Vdash)$, is equally translated to the range search condition $(v, \leq, \leq, v)$. Equation 4.2 guarantees that each index key k, where $v_l \Vdash k_l$, is part of the query result.

For example, when indexing a type hierarchy, each type label which does not have subtypes represents an index key. Due to the abstract superclass rule (cf. Subsection 3.4.1), there is one type label for each concrete type. A search condition either retrieves all nodes with a concrete type or with any supertype. In a hierarchical query, all subtypes equal the queried supertype because they are descendants of the supertype (cf. Equation 4.2).

Note that if the assumption that ancestors of index keys only appear as search keys does not hold, SCIENS has to take the largest possible child label as upper bound in a hierarchical query.

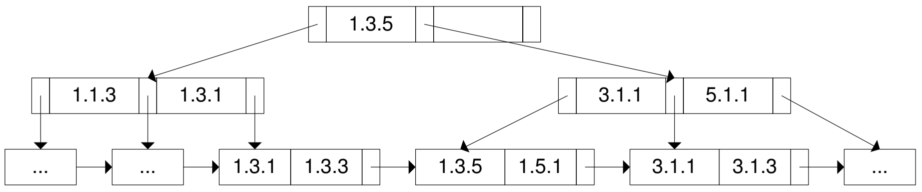


Figure 4.8: B+-tree on the path hierarchy of resources.

Example 4.12 (path index) *Figure 4.8 depicts a B+-tree on the paths of resource nodes, whereby the corresponding node labels represent the index keys. The example abstracts from what the index returns (e.g. it could return the date of resources). The B+-tree can retrieve all index entries associated with a specific resource, e.g. (*1.3.3, =*), by comparing document order between labels. Additionally, it supports hierarchical queries, such as searching index keys belonging to the second milestone (*1.3, $\Vdash$*). Starting from the root of the B+-tree, we first follow the first pointer (*1.3 < 1.3.5*) and then the second pointer (*1.1.3 < 1.3 < 1.3.1*). The first index key which fulfills the search condition in the figure has label* 1.3.1*. According to Equation 4.2,* 1.3 = 1.3.1 *because* 1.3 $\Vdash$ 1.3.1*. Search stops at label* 1.5.1 *because it is not part of the queried hierarchy. Similarly, the B+-tree can also look for all index keys belonging to the first project with the search condition (*1, $\Vdash$*).*

With regard to the sample queries, the B+-tree can be used to locate the resources requested by queries Q8-Q10. Query Q8 (`//resource[@date≥ '2007-01-01' and @date<'2008-01-01']`) requires a full scan of the B+-tree to retrieve all resources. In contrast, Q9 (`//project[title='semcrypt']// resource[@date≥ '2008-01-01']`) and Q10 (`//project[@id='26543']/ milestone[title ='design']/resource[@date≥ '2008-01-01']`) only look at a subtree of resources by selecting a certain project or milestone. As the index accepts paths as input, it avoids the need for structural joins between the selected project/milestone labels and (all) resource labels. The index does not depend on how the queried project or milestone is selected as it expects the corresponding paths (labels) as input. For example, the project could be selected by its id, title or position.

Support for hierarchical queries on structural properties offers the following advantages. By indexing labelpaths, an index can evaluate queries that refer to a specific labelpath or any subhierarchy of the indexed structural summary. An index on paths allows for evaluating search conditions within subtrees without performing structural joins. As the search keys of a path index correspond to intermediate query results, it allows for limiting the scope of index traversal to queried subtrees. Indices on types support queries on specific types or type hierarchies.

Example 4.13 (structure indexing) *Instead of indexing the labelpaths of title nodes with a hash table (cf. Figure 4.7), we can use a B+-tree on the title labelpaths. With regard to the sample queries, Q1 retrieves a specific title labelpath, whereas Q2 and Q3 are hierarchical queries, asking for all titles of milestones and of the whole document, respectively. To support queries Q11-Q13, we can build a B+-tree on the type of resource nodes, which uses type labels as index keys. It allows for retrieving all resources belonging to a particular type (e.g. `<TechnicalReport>`) or to a partial type hierarchy (e.g. `<Report>`). If we want to index both the path and the type of resources in one index, we can use a KDB-tree.*

By adapting the comparison operator to the property being indexed, the B+-tree can index structural properties, such as labelpaths, paths and types. As the KDB-tree uses the same operators as the B+-tree, it can index several structural properties without any modification of its algorithms.

4.4.3 Content and Structure Indexing

In the previous subsections, we have shown how to index content- or structure-oriented properties. Indexing multiple properties in one index is more compact and avoids joining intermediate results. To index both kinds of properties, existing XML indexing approaches either build proprietary index structures or use multidimensional indices (cf. Subsection 4.2.2). While the original KDB-tree only considers value domains, we have shown how to express structural queries as range queries in Subsection 4.4.2. The concept of extensibility therefore enables the KDB-tree to support all indexing requirements.

Indexing values and structural properties in one index structure raises the question of how to split index entries. Some OODB indices propose query-based splitting when indexing class hierarchies and values in a multidimensional index structure (e.g. [46, 159]). However, Lepouchard et al. [129] argue that the improvements of query-based splitting do not outweigh the required effort. Based on their results, the KDB-tree of SCIENS only adapts comparison operators to indexed properties, but is independent of the splitting strategy used.

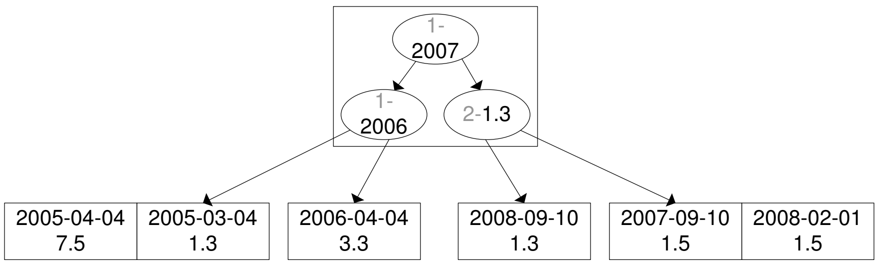


Figure 4.9: KDB-tree on values of resource dates and paths of milestones.

Example 4.14 (content and structure indexing) *Figure 4.9 shows a simplified KDB-tree on the date of resources and the path of milestones. Assume that each index entry maps index keys to resource nodes. The index supports evaluating Q8-Q10. Evaluating Q8 (`//resource[@date≥ '2007-01-01' and @date<'2008-01-01']`) only restricts the value dimension of the index. In contrast, Q9 (`//project[title='semcrypt']//resource[@date≥ '2008-01-01']`) and Q10 (`//project[@id='26543']/milestone[title = 'design']/resource[@date≥ '2008-01-01']`) contain search conditions for both dimensions. For example, to evaluate Q10, we first determine the desired milestone node(s). Let this node have label* 1.5. *We then evaluate the search conditions (*1.5, $\Vdash$*) and (*'2008-01-01'*, $\leq$, $\leq$, ∞) on the KDB-tree, resulting in the index entry with keys* '2008-02-01' *and* 1.5. *The index search does not access and return the index entry with index keys* '2008-09-10' *and* 1.3 *because the corresponding resource belongs to a different milestone. Indexing the path and the date therefore enables the KDB-tree to restrict search on resource dates to the requested milestone. Otherwise, if the index only referred to dates, the index would return all resources within the queried date range. Afterwards, it would be necessary to discard those resources that do not belong to the queried milestone by processing structural joins between the queried milestones and the resources returned by the date index. Similarly, we can use a KDB-tree on the* *labelpath and value of titles to support queries Q5-Q7 and a KDB-tree on the resource types and description values for queries Q11-Q13.*

Existing XML indexing approaches can mainly be classified into structure- and content-oriented approaches, dependent on whether they first group index entries according to structural properties or to values. Grouping index entries according to certain dimensions can be useful to adapt indices to specific queries. Multidimensional index structures treat each dimension equally. However, it is sometimes useful to favor queries on the structure or the content. We therefore introduce the concept of index nesting in the following section.

4.5 Nesting Index Structures

Index nesting refers to placing one index structure beneath another index structure. It allows for combining different kinds of index structures and indexing various properties without implementing proprietary structures. In contrast to multidimensional index structures, index nesting favors queries on specific dimensions. Dependent on which dimension represents the highest level of the nesting hierarchy, different queries are better supported. Generally, the most selective index should be placed first in the nesting hierarchy to quickly narrow down the result and reduce the size of the nested index structures to be traversed.

Existing XML and object-oriented index structures use the idea of index nesting to combine index structures on structure- and content-oriented properties. They create one value index for each labelpath (cf. content-aware path indices in Subsection 4.2.2) or for each type (e.g. CH-index [118]) or they first group index entries according to their value and then according to their labelpath (cf. Subsection 4.2.2). While each of these approaches uses proprietary structures and only supports a specific kind of nesting, SCIENS generalizes the idea of index nesting. Basically, SCIENS can nest arbitrary index structures. Thereby, the index keys of the superior index structure do not point to the node labels to be returned, but to a nested index structure. There is one nested index structure for each distinct index key of the superior index structure.

Nesting index structures only makes sense if the nesting reflects the hierarchical structure. The kind of nesting influences query performance (cf. key- and type-grouping in object-oriented indices in Subsection 4.2.1). If a structure index is nested beneath a value index, queries on the entire hierarchy are better supported. Nesting the index structures the other way round favors queries on a specific part of the hierarchy. As such, nesting always favors specific queries. If each dimension shall be treated equally, it is better to index the dimensions with a multidimensional index structure. SCIENS provides the flexibility to choose the alternative that best matches the query workload.

Example 4.15 (index nesting) *To support queries Q11-Q13, we can index the resource type and the description value by nesting index structures. Figure 4.10 depicts a B+-tree on the resource type with a nested B+-tree on the text of description values. This kind of nesting results in one B+-tree on the description value for each type and favors queries on a specific type. For example, to evaluate Q13 (`//resource<TechnicalReport> [description` $\tilde{\sim}$ `'data']`),*

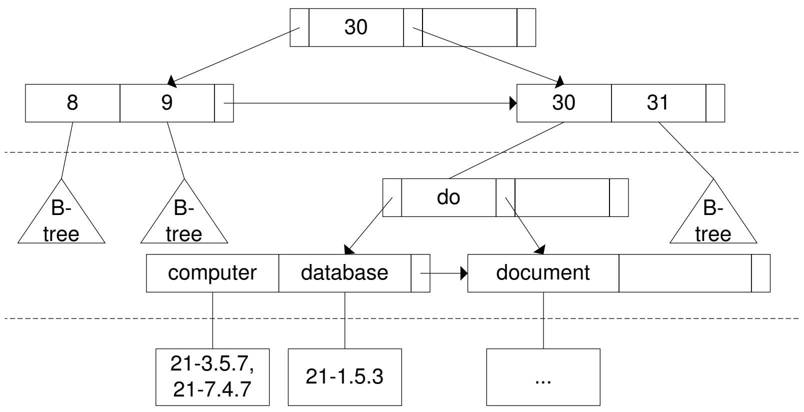


Figure 4.10: Nesting a text index on descriptions beneath an index on resource types.

only one nested index structure needs to be traversed, which is the one associated with type label 30*. Figure 4.11 nests the index structures the other way round. There is one B+-tree on the type hierarchy for each word of indexed description values. This kind of nesting favors queries on the entire or a large part of the type hierarchy. Q12 (`//resource[description ∼ 'database']`), for example, refers to all reports. If Q12 was evaluated with the index of Figure 4.10 instead, it would require traversing multiple B+-trees on the description value. This example shows that the most selective index structure should be placed first in the nesting hierarchy. Further, each kind of nesting is only appropriate for specific queries. If both queries should be supported equally, it would be better to build a KDB-tree on both dimensions.*

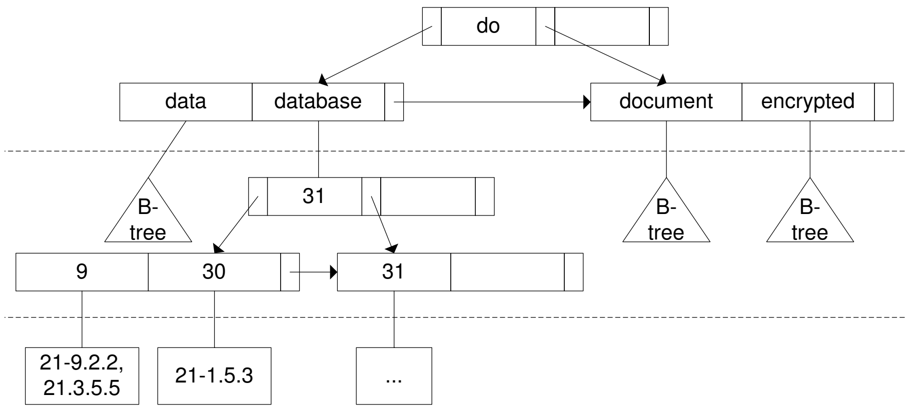


Figure 4.11: Nesting an index on resource types beneath a text index on descriptions.

It is also possible to nest several structure- or content-oriented indices. However, nesting content-oriented indices only makes sense if the nested index structures have a hierarchical relationship. Otherwise, if the first index is not very selective, it is necessary to traverse a large number of index structures at the nested levels (cf. multiple-key indexes [83]). In this case, a multidimensional index structure is the appropriate choice.

Example 4.16 (nesting alternatives) *To support queries Q8-Q10, we can nest a B+-tree on date values beneath a B+-tree on milestone paths, returning resource nodes. This results in one B+-tree on the date for each milestone. When evaluating Q10 (`//project [@id = '26543']/milestone [title = 'design']/resource [@date ≥ '2008-01-01']`), the index allows for retrieving the resources with the requested date after determining the queried milestone node. If we nest the B+-trees vice versa, the index favors queries which are mostly not limited to certain milestones or projects, such as Q8 (`//resource [@date ≥ '2007-01-01' and @date < '2008-01-01']`). To equally support Q8-Q10, it is better to build a KDB-tree on both dimensions (cf. Figure 4.9). Instead of indexing the paths of milestone nodes, it is also possible to index the values of project and milestone titles. By nesting these index structures, we can still reflect the hierarchy. However, in this case, the index does not support restricting the project if it is selected by its id and not by its title. Such an index therefore would not be appropriate for Q10 (`//project [@id = '26543']/milestone [title = 'design']/resource [@date ≥ '2008-01-01']`). Nesting index structures on the date and title of resources, e.g. to support query Q7 (`//resource [@date = '2007-10-31'] [title = 'xml']`), would not be appropriate as these node values do not hierarchically relate to each other. As there can be a large number of index keys for each dimension in this case, the nesting would result in a large number of index structures.*

Object-oriented and XML index structures use proprietary structures to nest structural and value indices (e.g. CH and SC index [118], CADG [195]). Each of these approaches only supports a specific kind of nesting. By generalizing the idea that one index key either points to a list of node labels or to another index structure, SCIENS can nest arbitrary one- or multi-dimensional index structures. Note that index nesting differs from simply concatenating index keys. E.g. in Figure 4.10, it would not be possible to evaluate hierarchical queries on types when concatenating index keys. Concatenation would only support exact comparisons, which can be reflected by using a hash table instead of a B-tree.

As a hash table only allows for exact comparisons and thus is usually the most selective index structure, it should always be placed first in the nesting hierarchy. Nesting multiple hash tables corresponds to a concatenation of the index keys to be hashed and only makes sense if the queries to be supported restrict each dimension with an exact comparison.

Example 4.17 (nesting with hash tables) *Assume that we want to support Q5 (`/projects/project [title = 'semcrypt']`) and Q6 (`//* [title = 'xml']`) with an index. A hash table on the title with a nested B+-tree on the labelpath of title nodes is an appropriate choice as both queries specify an exact comparison on the title.*

When nesting tree-shaped index structures, such as the B+-tree or the KDB-tree, each index key of the superior index structure points to the root of a nested index structure. This implies that at each nesting level, there are as many index structures as there are distinct index keys in the superior index structure. When traversing the superior index structure for performing an update or retrieval operation, the superior index structure retrieves the root of the nested index structure and forwards the update or retrieval operation to the nested index structure. After completing the operation, the nested index structure returns its result to the superior index structure. As such, each index structure can operate in two modes: as a nested index structure, the superior index structure accesses the root page, whereas in the independent mode, the index structure loads the root page itself.

Example 4.18 (nesting operations) *To evaluate query Q13 (`//resource <TechnicalReport> [description` $\dot{\sim}$ `'data']`) with the index of Figure 4.10, search starts at the B+-tree on the type. The queried index key with label* 30 *points to the root of a nested index structure. The superior index structure retrieves this root and forwards it together with the retrieval operation to the B+-tree on the description. The nested B+-tree evaluates the search condition (`'data'`, $\leq$, $\leq$, `'data`∞`'`) and returns the associated node labels (*21–1.5.3*) to the superior index structure.*

When nesting index structures, it also has to be considered that the total index size increases as there is one nested boundary structure for each index key of the superior index structure. Putting the index structure which has fewer distinct index keys first reduces the number of nested boundary structures and therefore the total index size. To further reduce index size, SCIENS proposes to nest index pages. This implies that a page can contain a page of a nested index structure as long as it is not full.

Example 4.19 (index size) *Considering the nesting alternatives of Figure 4.10 and Figure 4.11, the second alternative is expected to be larger in size because its superior index structure has many distinct index keys. In both figures, we depict pointers from index keys to the nested index structure and to the node labels as simple lines. This indicates that the corresponding pages could be nested within the pages of the superior index structure.*

4.6 Summary

SCIENS supports indexing the structure and content of XML documents with extensible, nestable structures. Extensibility refers to using one index structure for various properties and operations on these properties. Nestability allows for adapting index structures to hierarchical queries to favor queries on the entire hierarchy or queries on a specific part of a hierarchy.

In this thesis, we have selected a hash table, a prefix B+-tree and a KDB-tree. While the hash table only supports exact comparison, the prefix B+-tree and the KDB-tree are efficient for range queries. By adapting operators to indexed properties, the index structures do not only support comparing values, but also indexing structural properties, such as labelpaths, paths and types. Each index structure can index every property that supports its operators. The KDB-tree can index an arbitrary number of content- and structure-oriented properties.

Nesting index structures enables the grouping of index entries according to certain properties without the need of changing their algorithms or implementing proprietary index structures.

Note that the B+-tree can substitute the hash table as well as the KDB-tree can also be used to index a single dimension. However, the hash table is more efficient for exact queries than the B+-tree, and the KDB-tree with one dimension is less efficient than the B+-tree. Further, the concepts of SCIENS are not restricted to the specific index structures selected, but can be transferred to other index structures as well.

To summarize the flexibility of SCIENS, Table 4.4 defines sample indices and Table 4.5 contrasts these indices with the sample queries, as explained in Examples 4.20 and 4.21.

Example 4.20 (sample indices) *Table 4.4 defines sample indices for the queries of Table 2.2. I1-I5 focus on the labelpath and value of title nodes (cf. sample queries Q1-Q7), whereas I6-I10 index date values and milestone paths (Q8-Q10). I11-I15 support queries Q11-Q13 by indexing resource types and description texts. Indexing the text refers to indexing the individual words of each description value to create a full-text index. Assume that I1-I4 return labels of title nodes and I5-I15 return labels of resource nodes. All indices are selective, i.e. they are defined on some parts of the document. For example, I2 does not index all values, but only the values of title nodes.* SCIENS *offers a wide flexibility of which indexing alternative to choose. To index two properties, the table defines one index for each property, one index for each possible nesting and one multidimensional index. With the help of* SCIENS*, it is possible to choose the alternative that best matches the query workload.*

I1	B+-tree on title labelpaths
I2	hash table on title values
I3	hash table on title values, nested B+-tree on title labelpaths
I4	KDB-tree on title values and title labelpaths
I5	KDB-tree on resource title values and date values
I6	B+-tree on date values
I7	B+-tree on milestone paths
I8	B+-tree on date values, nested B+-tree on milestone paths
I9	B+-tree on milestone paths, nested B+-tree on date values
I10	KDB-tree on date values and milestone paths
I11	B+-tree on resource types
I12	B+-tree on description texts
I13	B+-tree on description texts, nested B+-tree on resource types
I14	B+-tree on resource types, nested B+-tree on description texts
I15	KDB-tree on description texts and resource types

Table 4.4: Indexing alternatives for queries of Table 2.2.

Example 4.21 (queries and indices) *Table 4.5 contrasts the sample queries of Table 2.2 with the sample indices of Table 4.4. As each index is selective, it can only support specific queries. For each query, we mark the best suited index with a star ($\star$). Further, we distinguish between well supported ($\oplus$) and*

Q·I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	$\star$			$\sim$											
2	$\star$			$\sim$											
3	$\star$			$\sim$											
4	$+$			$\sim$							$\sim$				
5	$\sim$	$\oplus$	$\star$	$\oplus$											
6	$\sim$	$\star$	$\oplus$	$\oplus$											
7	$\sim$	$+$	$+$	$+$	$\star$	$+$	$\sim$	$+$	$+$	$+$					
8					$+$	$\star$	$\sim$	$+$	$\oplus$	$\oplus$					
9	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$+$	$\sim$	$+$	$+$	$\star$					
10	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$+$	$\sim$	$+$	$\star$	$\oplus$					
11											$\star$			$\sim$	$+$
12											$\sim$	$\star$	$\oplus$	$+$	$\oplus$
13											$\sim$	$+$	$+$	$\star$	$\oplus$

$\star$	best index
$\oplus$	well supported
$+$	supported
$\sim$	index offers minor support

Table 4.5: Support for queries of Table 2.2 offered by indices of Table 4.4.

supported (+). The latter denotes that either the index only supports part of the query or that the query needs to traverse large parts of the index. Some indices only offer minor support for a query (∼). For example, the index on milestone paths (I7) only offers minor support for Q9 and Q10. However, it is efficient in combination with an index on date values (I8-I10). I11 offers minor support for Q4 because only reports contain descriptions.

To show the flexibility of SCIENS, Table 4.6 contrasts existing XML indexing approaches with SCIENS. While existing approaches propose various proprietary structures, SCIENS can represent the same indices by extending and nesting existing structures. As such, it is not necessary to implement a large number of different index structures to create arbitrary indices on the structure and content of XML documents.

As can be seen in the table, there is no equivalent for indices supporting branching path queries (branching path and sequence index) in SCIENS. Further, SCIENS only supports total summary indices and not partial summary indices. Also other specific indexing improvements have no counterpart in SCIENS. For example, the structure-oriented CADG [195] proposes to store signatures in the structure index to prune subtrees whose content does not match the search condition. However, we believe that selective indices compensate for such specific improvements because they can be adapted to specific queries and are smaller in size.

In case that an application requires indices that are currently not supported by SCIENS, it is possible to add new index structures. For example, if queries frequently need to traverse the same index structure multiple times and then join the results (e.g. to search keywords), a bitmap index will be more efficient.

Index	Example	SCIENS
Structure-oriented indexing		
summary index	[85]	B+-tree on labelpaths
branching path index	[111]	
content-aware path index	[59, 195]	B+-tree on labelpaths, nested index on values
path index variants	[21]	hash table on node names, nested B+-tree on labelpaths
basic element index	[135]	hash table on node names, hash table on values
element index variants	[55]	hash table on node names, nested B+-tree on paths
sequence index	[167]	
Content-oriented indexing		
content-oriented index	[70, 195]	B+-tree on values, nested B+-tree on labelpaths or node names
full-text index	[179]	B+-tree on text
Hybrid indexing		
multidimensional index	[22, 202]	KDB-tree on paths and values

Table 4.6: Representing existing indexing approaches with SCIENS.

SCIENS does not concentrate on specific index structures, but on adapting them to arbitrary query workloads.

With the concepts of extensibility and nestability, SCIENS can combine various index structures defined on arbitrary properties, which offers more flexibility than all existing approaches. Examples include indices on type hierarchies as well as path indices that can restrict index search to subtrees and avoid structural joins.

Part II

Processing Secondary Indices

Chapter 5

Index Framework

Contents

To integrate indices into query processing, an XML database requires an index framework that enables the database to select, access and maintain indices. Section 5.1 introduces the main concepts of such an index framework and Section 5.2 reviews related work. To process arbitrary indices, an index framework requires a generic index model, which is presented in Section 5.3. Section 5.4 describes the components of the index framework, which allow for processing arbitrary indices. Finally, Section 5.5 summarizes the main ideas of the framework.

5.1 Introduction

XML databases require indices to support frequently issued queries on specific document fragments. In Chapter 4, we have presented the indexing approach SCIENS, which enables the extension and nesting of index structures to reflect arbitrary query workloads. As there does not exist one index which is optimal for all queries, an XML database needs to process several indices, which index different parts of documents using various index structures. Processing indices comprises selecting indices during query optimization, accessing indices during query execution and maintaining indices when updating documents. Note that the term index selection is sometimes also used to denote the process of suggesting possible indices for query workloads during database design, which is out of scope of this thesis (cf. e.g. [94]).

Example 5.1 (index processing) *Q9 (`//project[title='semcrypt']//resource[@date` ≥ `'2008-01-01']`) selects all resources that belong to project 'semcrypt' and that have been created in '2008' or later. To answer this query without looking at each project and each resource, the query optimizer looks for appropriate indices. Assume that there exists an index on date values (I6 of Table 4.4). The query optimizer has to detect that this index answers part of the query and it has to rewrite the query plan such that it uses the index.*

Basically, two possibilities exist for index processing: (i) define proprietary query and update algorithms for each index or (ii) provide a uniform way to process various indices based on an index model. Alternative (i) only works for a small set of predefined indices. Each time when adding a new index, the query and update algorithms need to be adapted as well. Alternative (ii) provides a generic approach, which is not restricted to currently existing indices. Adding new indices does not require changing the query and update algorithms as these can handle arbitrary indices.

As alternative (i) is inappropriate for flexible, selective indexing, an XML database requires an index framework that enables the database to process arbitrary indices. For this purpose, it requires an index model that represents what an index is defined on to select and maintain indices.

5.2 Related Work

This section reviews related work on index frameworks. It first looks at index interfaces of object-relational databases that enable extensible indexing. Current XML databases support querying and updating a limited set of index structures, as will be shown by looking at the eXist XML database [148]. Finally, two index processing approaches, KeyX [94] and XAMs [16], will be described in more detail.

5.2.1 Interfaces for Extensible Indexing

Extensible indexing refers to integrating new index structures into databases (cf. Subsection 4.2.1). For this purpose, object-relational databases provide index interfaces that let users implement their own index structures. For example, Oracle data cartridges [184] provide an SQL based framework for index definition, index maintenance and index scan. Each index structure has to implement corresponding methods and is responsible for defining the index structure, maintaining the index content and searching the index during query processing.

An XML index framework needs to provide similar interfaces. As XML indices are not defined on table columns, it additionally requires generic query and update algorithms.

5.2.2 XML Databases

Current XML databases only offer limited support for indexing, which facilitates index processing. For example, the native XML database eXist [1, 148] creates element indices on node names and supports full-text and range indices on node values. Thereby, the nodes to be indexed can be selected by a total or a partial matching query. The main drawbacks are that it does not support multidimensional index structures nor predicates in index definitions. As a consequence, each index entry has exactly one index key and no navigation is supported between indexed and returned nodes. This restriction greatly facilitates index processing as it does not require to consider relationships between indexed nodes. Offering more complex indices requires a more advanced index framework.

5.2.3 KeyX

KeyX [94] is an XML indexing approach that provides a representation of indices and algorithms to select, suggest and maintain indices. Each index is represented by a set of path expressions, one for each key and each qualifier and one expressing the index return. Keys express index keys on node values, whereas qualifiers represent required nodes.

Example 5.2 (KeyX) *KeyX represents a value index on the title and date of all resources that have a description as follows:*

keys={`//resource/@date`, `//resource/title`}
qualifiers={`//resource/description`}
return={`//resource`}

The main drawbacks of KeyX are that the approach does not consider more complex indices, e.g. on paths or type hierarchies. The provided algorithms for index processing are still limited as well. For example, the index selection algorithm can find at most one index for a query. If several indices match a query, it selects the best index based on simple statistics about the document. It does not consider executing a query by accessing several indices. Further, KeyX does not make use of schema information to improve index processing.

5.2.4 XML Access Modules

XML Access Modules (XAMs) [16, 17] are generic descriptions of what is stored in the primary storage structure, in a view or in a secondary index structure.

Their primary purpose is to achieve physical data independence by processing queries on XAMs without requiring knowledge of underlying data structures.

Tree patterns are frequently used to represent queries on XML documents [42, 54]. XAMs use annotated tree patterns to represent what is stored in a storage structure. Each node in a XAM may be annotated with an identifier (ID), a tag, a value (Val) or a content specification. These specifications indicate which properties are stored in a XAM. Specifications that are marked as required (R) express keys in an index. Nodes in the pattern are connected via parent-child (/) or ancestor-descendant (//) edges. Edges have associated a join semantic and may be marked as optional.

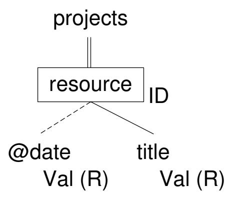


Figure 5.1: XAM defining a value index on resource dates and titles.

Example 5.3 (XAM) *Figure 5.1 shows a sample XAM representing a value index on the date and title of resources. The required value specifications (Val (R)) express index keys on node values. The* `resource` *node represents the return of the index, which is indicated by its border. The ID specification associated with the* `resource` *node denotes that the index returns the id (label) of resource nodes. The dotted edge between the* `resource` *and* `@date` *node expresses that the date is optional, i.e. the index also indexes resources that do not have a date.*

Besides providing a representation for storage structures, the authors of the XAM approach also describe an algorithm for selecting XAMs for queries. The first step is to extract query patterns from XQuery based on a logical query algebra [19]. It then tests whether there exists a XAM which is equivalent to a query pattern [18]. Equivalence is determined by checking containment between patterns based on a structural summary. If there does not exist a single XAM which is equivalent to the query pattern, the algorithm joins XAMs and compares their equivalence to the query pattern. For each query pattern that can be answered with the available storage structures, there finally exists a query plan consisting of a set of XAMs connected via joins. The selection algorithm does not consider costs for accessing XAMs, but selects the query plan which accesses the lowest number of XAMs.

The XAM approach, which has been developed in parallel to SCIENS, provides the necessary abstraction to process indices regardless of what they index. While it provides an algorithm to select indices for query processing, it does not consider maintaining XAMs when updating documents. The index model, which we present in the following, has some similarities to XAMs, but can express all indices of the SCIENS approach.

5.3 Index Model

The index model provides a generic description of indices, which is expressive enough to select and maintain arbitrary indices without depending on their underlying data structures. It represents each index on a logical and a physical level. The index definition specifies what the index is defined on. It is represented as an annotated tree pattern, which enables the processing of index definitions in a language-independent way. Each tree pattern is referred to as index pattern and logically defines an index. An index structure determines the physical data organization of an index. When an index has been selected for a query, the search configuration contains the search keys to be looked up in the index. Index patterns and search configurations are completely independent of the physical data organization. The index configuration associates logical index patterns with physical index structures. In the following, we look at the various parts of the index model in more detail.

5.3.1 Index vs. Index Structure

An *index* is a search function that maps index keys to nodes in a document. An index key can be any property of the XML data model, i.e. the value, name, type, path or labelpath of a node.

Definition 5.1 (index) An *index* I for a document D with nodes N consists of a set of index entries, $\mathsf{I} = (\mathsf{E})$, which contain index keys K and nodes from the document. Each entry in an index $\mathsf{e} \in \mathsf{E}$ maps a list of index keys, specifying the search condition, to a number of labels, representing the nodes returned by the search function, $\mathsf{e} = ((\mathsf{k}_1, \ldots, \mathsf{k}_\mathsf{j}) \in \mathsf{K}) \rightarrow (label(n_1), \ldots, label(n_m))$, where $n_1 \ldots n_m \in N$. The number of index keys j is identical for each entry in an index and determines the dimensionality of the index. The index keys of one index entry may refer to different properties, however, all index entries index the same property at k_i, where $\mathsf{i} \in \{1 \ldots \mathsf{j}\}$. In case that an index key does not exist, it is represented by a null value ($\perp$).

For ease of presentation, we assume that each index is defined on one document. The presented approach can however equally support indices on collections of documents. In this case, the index will contain index keys and nodes from different documents. It is therefore necessary to add a document identifier to each node label. The index itself is not affected by the number of documents it indexes. This is only relevant when generating index entries (cf. Chapter 6).

The node labels that an index returns can belong to the same nodes as the indexed properties or represent any other nodes that structurally relate to these nodes, e.g. ancestor nodes (cf. object-oriented indices on aggregation graphs in Subsection 4.2.1). Returning labels of ancestor nodes reduces index size because the number of ancestors corresponds at most to the number of their descendants. However, returning ancestor labels is only appropriate for queries which do not need the descendants for further processing. Otherwise additional queries would be necessary to retrieve the descendants.

Example 5.4 (index entry) *Consider an index that selects resources according to their type and description text (I15 of Table 4.4). This index is two-*

dimensional and each index entry has two index keys. Sample index entries of this index are $\mathsf{e}_1 = ((\texttt{<Report>}, \text{‘XML’}) \rightarrow (21\text{–}1.1.1))$ *and* $\mathsf{e}_2 = ((\texttt{<Documentation>}, \perp) \rightarrow (21\text{–}1.9.7, 21\text{–}3.5.5))$.

Physically, SCIENS represents each index with *index structures*. While in the general case one index has one underlying index structure, the concept of index nesting (cf. Section 4.5) entails that several index structures can be used to represent one index.

Definition 5.2 (index structure) An *index structure* is a data structure that organizes index entries into pages, such as a hash table, a prefix B+-tree or a KDB-tree. It provides algorithms to traverse its data structure for retrieving and updating index entries.

5.3.2 Index Variables and Index Definition

Index variables define the properties which an index uses as index keys and supported operations. Each index entry has one index key for each variable. The number of variables of an index determines the dimensionality of the index.

Table 5.1 depicts the properties and operators that are provided by SCIENS. The operator $\Vdash$ denotes that hierarchical queries are supported on the property. Concerning values, the operator $\leq$ expresses range queries, whereas operator $\sim$ stands for full-text queries. Value variables index node values, i.e. the values of attribute or text nodes. Defining variables on the content of element nodes is currently not supported as queries typically refer to the values of attribute or text nodes instead of to the concatenation of the values of several text nodes.

property	operator
name	$=$
labelpath	$=, \Vdash$
path	$=, \Vdash$
type	$=, \Vdash$
value	$=, \leq, \sim$

Table 5.1: Index variables defining the property being indexed and supported operators.

Definition 5.3 (index variable) An index variable v consists of one operator and one property, $\mathsf{v} = (\mathsf{o}\ \mathsf{p})$, where $\mathsf{o} \in \{=, \Vdash, \leq, \sim\}$ and $\mathsf{p} \in \{\mathsf{p_n}, \mathsf{p_l}, \mathsf{p_p}, \mathsf{p_t}, \mathsf{p_v}\}$. The subscript of a property denotes whether the property is the <u>n</u>ame, <u>l</u>abelpath, <u>p</u>ath, <u>t</u>ype or <u>v</u>alue of a node. Supported combinations of operators and properties are defined in Table 5.1. Variables can be optional or required. Required variables express that each search condition must provide a search key for that variable. Required variables are enclosed in brackets $(\mathsf{o}\ \mathsf{p})$, optional variables in square brackets $[\mathsf{o}\ \mathsf{p}]$.

Example 5.5 (index variable) *An index on resource types and description texts contains two index variables,* $\mathsf{v}_1 = [\Vdash\ \mathsf{p_t}]$ *and* $\mathsf{v}_2 = [\sim\ \mathsf{p_v}]$. *The first variable indicates that the index supports hierarchical comparisons on types, the second that it supports full-text queries on values.*

An *index definition* selects the nodes to be indexed by specifying which properties to use as index keys and which nodes to return by the search function. It corresponds to a query which contains index variables to define the index keys and the operations to be supported on these keys.

Definition 5.4 (index definition) An *index definition* specifies the nodes to be indexed and corresponds to a query with index variables. The nodes returned by the query represent the nodes returned by the index. The index variables define the index keys and supported operations.

A database provides an index definition language, which allows users to create indices. An index definition language can extend the query language with index variables. We use index patterns to represent and process index definitions (cf. Subsection 5.3.3). A syntax for defining indices is however out of scope of this thesis. Example 5.6 shows a possible index definition using an enhanced XPath syntax. The index definition numbers the index variables. The numbers can then be used to specify an order of the index variables in case of index nesting.

Example 5.6 (index definition) *An index definition language can use the enhanced XPath expression* `//resource[sc:type` $\Vdash$ `$1][description/text()/sc:value` $\sim$ `$2]` *to define an index on the type and description of resources (cf. I15 of Table 4.4). The index definition contains two index variables $1 and $2. The names with namespace prefix* `sc` *denote which properties to use as index keys and the comparison operators define the operations to be supported on the properties. The first variable defines to index types and to support hierarchical comparisons on types. The second defines to support full-text queries on node values. By default, the database can use a KDB-tree as underlying index structure. If we want to use two nested index structures, we can add* `nest by $1, $2` *to the index definition. In this case, the database can decide to nest two prefix B+-trees as the prefix B+-tree supports each property and operation. In this case, the index will be physically represented as a B+-tree on resource types with nested B+-trees on description texts.*

5.3.3 Index Pattern

While an XPath like syntax can be used to create an index, a tree pattern is adequate to represent and efficiently process an index definition in a language-independent way. Tree patterns are frequently used to evaluate queries by tree pattern matching [42, 50, 54, 200]. The XAM approach [16] uses tree patterns to provide a generic description of XML storage structures. Similarly, we use tree patterns to represent index definitions, which we refer to as *index patterns.* These patterns do not only provide a language-independent description of what is indexed. They also allow for comparing index and query patterns for index selection (cf. Subsection 5.4.1) as well as for comparing index patterns with update fragments for index maintenance (cf. Chapter 6). Translating index definitions in XPath or XQuery into index patterns resembles extracting query patterns from XQuery (cf. [19, 54]) and is out of scope of this thesis.

Definition 5.5 (index pattern) An *index pattern* P is an unordered, directed tree, $\mathsf{P} = (\mathsf{N}, \mathsf{F}, name, var, req, root, return, doc, \mathsf{L}, \mathsf{V}, D)$, where

- N is a finite set of pattern nodes.
- $\mathsf{F} \subseteq \mathsf{N} \times \mathsf{N}$ is a finite set of directed edges. There are two kinds of edges, namely parent-child ($\mathsf{F}_{/}$) and ancestor-descendant ($\mathsf{F}_{//}$) edges.
- Pattern nodes have a name, represented by function $name : \mathsf{N} \rightarrow \mathsf{L}$, where L is a set of pattern node names.
- Pattern nodes may have associated variables, represented by function $var : \mathsf{N} \rightarrow 2^{\mathsf{V}}$, where V is a set of index variables.
- A pattern node is either marked as required or optional, represented by function $req : \mathsf{N} \rightarrow \{0, 1\}$, where 1 denotes a required pattern node.
- Each index pattern has one pattern node which has no incoming edge and which is returned by function $root : \mathsf{P} \rightarrow \mathsf{N}$.
- Each index pattern has one distinguished pattern node which marks the return of the index. This pattern node is always required and it is returned by function $return : \mathsf{P} \rightarrow \mathsf{N}$.
- Each index pattern is defined on one document D, which is returned by function $doc : \mathsf{P} \rightarrow D$.

□

We define functions *parent*, *children*, *ancestor* and *descendant* in the same way as the corresponding functions in a document (cf. Subsection 2.1.1). Index variables that are associated with optional pattern nodes define index keys as optional. An optional pattern node may not have required pattern nodes as descendants. We refer to an index pattern that consists of one pattern node as a *simple index pattern*. If any pattern node has more than one child, the pattern is called a *twig index pattern*, otherwise it is referred to as *path index pattern*.

An index pattern is associated with one document (which could be extended to a collection of documents). Each index consists of nodes of the document that structurally correspond to the pattern. More precisely, nodes of the document match an index pattern if their names comply with the names of the pattern nodes and their structural relationships correspond to the edges defined between the pattern nodes. The index variables associated with pattern nodes specify which properties of the nodes represent index keys. Chapter 6 provides details on how to extract index entries from documents based on index patterns.

Each index entry specifies one index key for each index variable of the associated index pattern. The preorder traversal of index variables in the index pattern determines the order of index keys in an index entry. An index key of an index entry is null if there does not exist a node in the document that matches the pattern node of the index variable. An index key may only be null if the corresponding index variable is associated with an optional pattern node.

Example 5.7 (index patterns) *Figure 5.2 shows three sample index patterns. Parent-child edges are depicted as single lines, ancestor-descendant edges as double lines and optional pattern nodes are connected via dotted lines. The return pattern node is bordered and index variables are written down to the right of the pattern node with which they are associated. The left-most index pattern*

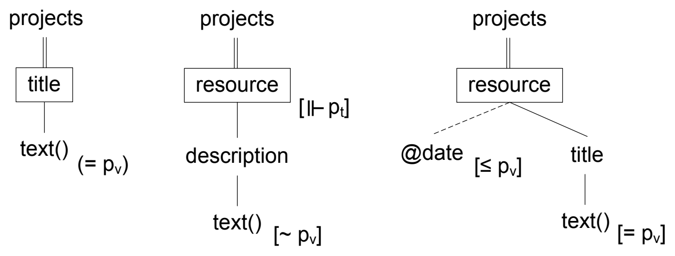


Figure 5.2: Index patterns defining a value index on titles (left), a multidimensional index on description values and resource types (middle) and a multidimensional value index on resource dates and titles (right).

is a path index pattern defining a value index on `title` *nodes that supports exact comparisons. Note that the variable is defined on the text node and not on the element node* `title` *because indexing the element content is not supported. In the document of Figure 3.4, for example, nodes* 1, 6–1 *and* 17–1 *match pattern nodes* `projects`, `title` *and* `text()`*, respectively. As the pattern node* `text()` *has associated a value index variable and the pattern node* `title` *represents the return of the index, a sample index entry maps the index key '*`semcrypt`*' to the node label* 6–1*. The index pattern in the middle specifies to index the value of* `description` *nodes and the type of* `resource` *nodes for full-text and hierarchical queries, respectively. The right-most index pattern is a twig index pattern. It specifies an index that can select* `resource` *nodes according to their* `@date` *and* `title`*. The index supports range comparisons on dates and exact comparisons on titles. As* `@date` *nodes are optional, the index also contains index entries whose date is null. This entails that the index can also be used to look for resources that do not have a date. The index patterns correspond to the indices I2, I15 and I5 of Table 4.4, respectively.*

5.3.4 Index Configuration

Each index can physically use one or - alternatively - several nested index structures as underlying data structure. An *index configuration* defines the index structures for an index by providing a mapping between the index variables of the associated index pattern and the index structures.

Definition 5.6 (index configuration) Given an index pattern P with index variables V, an *index configuration* $C_I = (\mathsf{T}, config)$ consists of a list of index structures T and function $config : \mathsf{V} \rightarrow \mathsf{T}$, which maps index variables to index structures. In case that an index configuration consists of several index structures, the order of the index structures defines the nesting order between the index structures. The first index structure is first in the nesting hierarchy.

Function $config$ is total because each variable requires a mapping to an index structure. It may map several variables to the same index structure if this index structure is multidimensional. In case of index nesting, the index configuration consists of several index structures. Note that it defines one index structure for each nesting level, independent of how many physical index structures there actually exist. For example, when nesting two B+-trees, there is one nested (physical) B+-tree for each index key of the superior index structure. However, the index configuration simply defines one B+-tree for each nesting level (i.e. each index variable) as the actual number of nested index structures dynamically changes.

Dependent on the types of index structures supported, function $config$ may have to consider further restrictions. SCIENS currently supports hash tables, prefix B+-trees and KDB-trees (cf. Chapter 4). As each index structure supports different operators, the mapping is only valid if the index structure can handle the operator defined in the index variable. While hash tables only accept variables with exact comparisons, the prefix B+-tree can handle any variable. Note that an optional index variable cannot be mapped to a hash table (cf. Subsection 5.3.5). Multiple variables can only be mapped to the KDB-tree.

Example 5.8 (index configuration) *Consider the second index pattern of Figure 5.2. A sample index configuration maps the type variable to a B+-tree and the variable on descriptions to a (nested) prefix B+-tree. Alternatively, the index configuration can also map both variables to a KDB-tree.*

5.3.5 Search Configuration

Queries access indices to evaluate different search conditions. A *search configuration* specifies a set of search conditions to access an index. It consists of one search condition for each index variable. There are simple and range search conditions. A simple search condition consists of one search key and one comparison operator. A range search condition comprises two search keys and two comparison operators, whereby one search key and comparison operator represent the lower, the others the upper bound. Simple search conditions can also be expressed as range search conditions (cf. Subsection 4.4).

Definition 5.7 (search configuration) Given an index pattern P with index variables V, a *search configuration* $C_S = (\mathsf{O}, search)$ consists of a set of search conditions O and a bijective function $search : \mathsf{V} \rightarrow \mathsf{O}$, which maps index variables to search conditions. The function must provide a mapping for at least every required index variable.

In case that a search configuration does not contain any search condition, the index performs an entire index lookup. Note that certain index structures, such as the hash table, cannot perform lookups without search conditions. This is insofar reflected as the index configuration may not map an optional index variable to a hash table. Therefore, a valid search configuration must provide a search condition for the hash table.

variable operator	search operator
$=$	$=$
$\Vdash$	$=, \Vdash$
$\sim$	$\sim, \dot{\sim}$
$\leq$	$=, <, \leq$

Table 5.2: Search operators supported by operators of index variables.

Each search key must match the property of its variable. For example, in case of a variable on types, the search key has to be a type as well. The search keys of a range search condition can have the special value infinite ($\pm\infty$), which matches any index key. This is necessary to specify range searches with an infinite lower or upper bound. Which search operator is allowed in a search condition depends on the operator defined in the index variable (cf. Tables 5.2, 4.2). For example, when an index defines a type variable with operator $\Vdash$, the index supports exact queries on types as well as queries on type hierarchies.

Example 5.9 (search configuration) *Consider the third index pattern of Figure 5.2. A sample search configuration specifies the range search condition* $\mathsf{o}_1 = (\text{'}\mathtt{2007\text{-}01\text{-}01}\text{'}, \leq, \leq, \infty)$ *for the date variable and the simple search condition* $\mathsf{o}_2 = (=, \text{'}\mathtt{XML}\text{'})$ *for the title variable. Evaluating this search condition on the index returns all resources written in '2007' or later and having the title 'XML'.*

5.4 Components

To process queries and updates, databases provide a query optimizer and an execution engine. The query optimizer rewrites query plans based on heuristics and/or statistics and selects the plan with minimal costs. The execution engine then processes the execution plan, which exactly defines the necessary steps to process a query or update operation. Integrating indices into databases requires to extend these components. The query optimizer needs to select appropriate indices for queries and the execution engine has to access and maintain indices when processing queries and updates, respectively.

To provide flexibility and extensibility, performing these tasks must be independent of specific indices. In Section 5.3, we have presented an index model which provides a generic description of indices. Each index has associated an index pattern, specifying what the index is defined on, and an index configuration, providing a mapping between the logical index pattern and its physical index structures. With the help of index patterns, the query optimizer can select indices and the execution engine is able to access indices and to determine relevant updates for indices. The index configuration then enables indices to perform the necessary query or update operations on their index structures. The main advantage of this approach is that the components do not require knowledge about specific index structures to perform their tasks. They can handle any existing (or future) index that can be expressed by the index model. When integrating a new index structure, it is possible to change the physical organization of an index (e.g. replace the prefix B+-tree of an index with a new full-text index structure). As long as the index model does not change, processing the index remains unaffected by the new index structure.

In the following, we describe how to extend the query optimizer and the execution engine to process indices. Subsection 5.4.1 shows how the query optimizer can select indices using the XAM approach. The index engine adopts the tasks of accessing and maintaining indices from the execution engine and is introduced in Subsection 5.4.2. As there still does not exist a generic approach for maintaining indices, Chapter 6 provides details on index maintenance.

5.4.1 Index Selection

The query optimizer has to select indices for queries and generate query plans that use indices. The main challenges are (i) to determine which indices equal which parts of a query and (ii) to rewrite the query such that it uses the index and still produces the same result. To provide extensibility, the index selection must not assume the existence of specific indices, but needs to operate on a generic description of indices, such as index patterns.

The XAM approach of Arion et al. [16, 18] describes how to select storage modules for queries (cf. Subsection 5.2). It uses schema information in the form of a structural summary to ease the selection process. As indices are part of storage modules and our schema contains all information of a structural summary, the query optimizer can use the XAM approach for index selection. Its prerequisite is to represent every storage structure (primary and secondary indices, views) as a XAM. In [17], Arion et al. have shown how to express various XML index structures as XAMs. In the following, we first look at differences between index patterns of SCIENS and XAMs, describe the XAM selection algorithm and then look at necessary extensions to the XAM approach to support the full flexibility of SCIENS.

Both index patterns and XAMs are annotated tree patterns. XAMs annotate pattern nodes with identifier, tag, value and content specifications, whereas index patterns use index variables. When expressing an index pattern as a XAM, it is necessary to translate the index variables into required specifications. Path index variables correspond to identifier specifications, index variables on node names are equal to tag specifications and value index variables correspond to value specifications. XAMs do not consider indices on types and labelpaths and therefore need to be extended with corresponding specifications to be able to express all index variables.

To a certain extent, XAMs are more expressive than index patterns as they describe storage structures in general. For example, they can specify multiple return nodes as well as multiple properties to be returned. It is further possible to add join semantics to XAM edges and to add value ranges to value specifications, which enables XAMs to restrict the nodes that are contained in a storage structure based on their value. To describe indices, one return node and default join semantics are sufficient. Adding value ranges to index pattern nodes would enable index patterns to further restrict the set of nodes indexed. Such enhancements to index patterns are possible, but are not further regarded in this thesis.

In Subsection 5.2, we have shortly outlined the main steps of how to select XAMs for queries. To select XAMs for queries, the XAM approach tests equivalence between query patterns and index patterns. For this purpose, it first generates canonical models from query and index patterns by replacing wildcards and ancestor-descendant edges with the help of the structural summary. Canonical models facilitate determining equivalence between patterns. Roughly speaking, a query pattern p is equivalent to a pattern p' if for each canonical model $\mathsf{p_m}$ of p, there exists a canonical model $\mathsf{p'_m}$ of p' that has the same structure and the same return nodes. If no single XAM is equivalent to a query, the algorithm combines XAMs via structural joins and tests their equivalence to the query. Different XAMs may return different properties of nodes or even use different labeling schemes. As each XAM defines what exactly it returns and even distinguishes various kinds of labels, the query optimizer can decide how to combine several XAMs, e.g. which join algorithm to use.

If no single XAM or no combination of XAMs equals the query pattern, the query cannot be answered with the available storage structures. In many cases, there will be several alternative query plans that access a different combination of XAMs. The XAM approach chooses the query plan which accesses a minimum number of XAMs. The alternative use of a cost model is not discussed. When using a cost model, calculating index access and update costs is straightforward in SCIENS as it uses well-known index structures. For example, accessing a hash table is always more efficient than accessing a B+-tree for exact comparisons. Cost models discussed in literature are still limited (cf. e.g. [93, 122, 205]). A detailed cost model is out of scope of this thesis as it is orthogonal to our indexing approach.

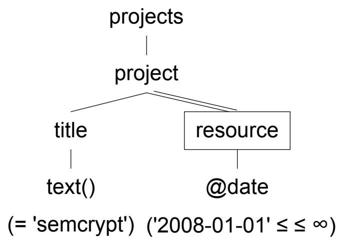


Figure 5.3: Query pattern on title and date values.

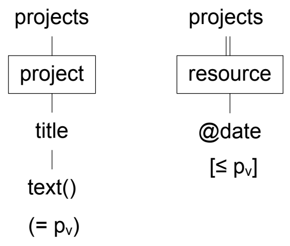


Figure 5.4: Index patterns for the query of Figure 5.3.

Example 5.10 (index selection) *Figure 5.3 depicts a sample query pattern and Figure 5.4 visualizes two index patterns. The query corresponds to sample query Q9 (`//project [title = 'semcrypt']//resource [@date` ≥ `'2008-01-01']`). To generate a query plan for this query, the XAM approach proceeds as follows:*

1. *Generate canonical patterns. In this example, each pattern has exactly one canonical representation because for each ancestor-descendant edge, there is only one matching labelpath in the schema. For example, considering the query pattern, we can use the schema to extend the path `/projects/project//resource` to `/projects/project/milestone/resource`.*

2. *Determine equivalence between each individual index pattern and the query pattern. In this case, no single index pattern is equivalent to the query pattern.*

3. *Combine the index patterns via joins. As each index returns node labels, we can combine the index patterns via a structural join between projects and resources, $\mathsf{p}_1 \bowtie_{project \Vdash resource} \mathsf{p}_2$. The combined index pattern has the same structure as the query pattern and the index variables support the operations of the search condition. A possible query plan thus is to access each index and then structurally join the results.*

Some enhancements to the XAM approach are required to select indices based on index patterns. The XAM approach tests structural equivalence between XAMs and query patterns based on their canonical representation. We further need to ensure that an index pattern supports the search conditions of a query based on its index variables. More precisely, the index variable and the search key must refer to the same property. Additionally, the search operator must be supported by the operator of the index variable (cf. Table 5.2). For example, assume that there is an index with a variable that only supports exact comparisons, which may physically be represented by a hash table. In case of a range search, the operator of the index variable does not support the search condition and thus the index must not be selected.

Further, we need to extend the XAM approach to enable the selection of path, labelpath and type indices. Selecting type hierarchy indices corresponds to selecting value indices. In the following, we outline how to select path and labelpath hierarchy indices, respectively.

Path hierarchy indices. XAMs provide ID specifications, which are similar to paths, but they do not consider hierarchical queries on paths. Path hierarchy indices in SCIENS accept node labels as index keys. This enables the database to take the result of one index access as input into another index instead of accessing each index individually and joining their results. Instead of combining two index patterns by structural joins only, it should therefore also be possible to access indices consecutively. The result of one index access dynamically becomes the input of another index.

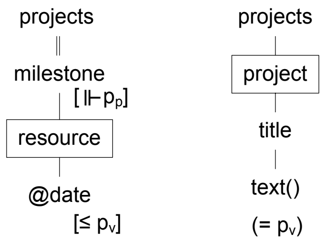


Figure 5.5: Index patterns to retrieve resources by their milestone path and date value (left) and projects by the value of their title (right).

Example 5.11 (index selection with path index) *Consider the index patterns of Figure 5.5 and assume that we want to evaluate sample query Q9 (`//project[title='semcrypt']//resource[@date ≥ '2008-01-01']`). With the original XAM approach, we can evaluate the query by accessing the two index patterns and performing a structural join between the results (cf. Example 5.10). The left index pattern specifies a path hierarchy variable on milestones. We can therefore replace the structural join by using the project labels returned by the right index as search keys for this variable.*

Labelpath hierarchy indices. To find indices on labelpath hierarchies, XAMs have to be extended by labelpath specifications. Testing equivalence between a query and an index pattern with a labelpath variable works as follows. If the query is equivalent to some canonical index pattern, check if search in the index can be restricted to these patterns by adding a search condition on the labelpath. While a search condition with an exact comparison restricts search to one canonical pattern, a hierarchical comparison selects a set of canonical patterns.

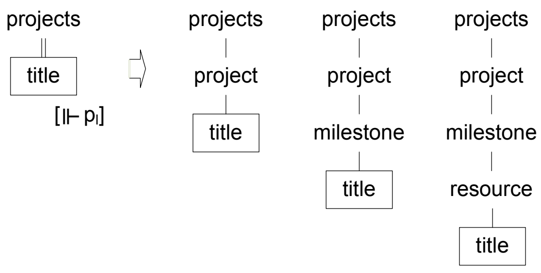


Figure 5.6: Index pattern on labelpaths of titles and its canonical patterns.

Example 5.12 (index selection with labelpath index) *The index pattern in Figure 5.6 depicts an index on title labelpaths (I1 of Table 4.4) and its canonical patterns. Figure 5.7 represents queries Q3, Q1 and Q2 of Table 2.2. Q3 exactly matches the index, whereas Q1 only matches the first canonical pattern. To access the index for Q1, it is therefore necessary to restrict search to the labelpath of project titles. As schema label* 6 *represents the desired labelpath, evaluating search condition* (=,6) *on the index answers Q1. Q2 matches the second and third canonical pattern of the index as it comprises milestone and resource titles. The search condition* (⊩,7) *restricts search to these patterns, where schema label* 7 *identifies the* `milestone` *hierarchy.*

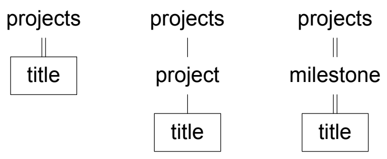


Figure 5.7: Query patterns Q3, Q1, Q2, retrieving titles by their labelpath.

5.4.2 Index Engine

The index engine is the part of the execution engine that is responsible for initializing, accessing and maintaining indices. Figure 5.8 depicts the overall structure of the index engine. All index structures provide the same interface, which allows the index engine to perform its tasks independently of specific index structures and to extend and add index structures. To store and retrieve pages from disk, each index structure interacts with the storage layer.

The index engine processes indices via the generic interface of their index structures, each of which provides the following functionality to the index engine:

- *Initialization - create/remove:* The index structure initializes its data structure (e.g. root page) or completely eliminates its data structure.

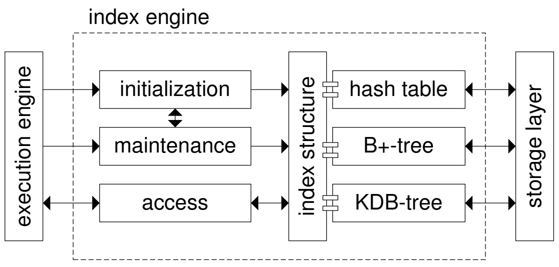


Figure 5.8: Architecture of the index engine.

- *Maintenance - insert/delete/update:* Each maintenance operation receives as input the index entries to be inserted, deleted or updated. The index structure performs the required maintenance operation on its structure using its proprietary algorithms.
- *Access - retrieve:* The index structure receives as input a search configuration, traverses its structure to evaluate the search conditions and returns matching nodes.

The index engine initializes, maintains and accesses indices. With the help of the index and search configurations of indices, it interacts with their index structures. The concept of index nesting entails that one index can have one or several underlying index structures. The index engine always interacts with the first index structure of the nesting hierarchy, which it determines with the help of the index configuration. It then performs each operation on this index structure as follows:

Index initialization. The index engine is responsible for creating and removing indices. When creating indices, it first forwards the creation operation to the index structure. In case that the index is created on an existing document, it needs to determine the index entries to be inserted into the index structure. For this purpose, it interacts with the maintenance module, which determines relevant index entries. In case of removing an index, the index engine needs to ensure that the index structure is completely removed as well. For this purpose, it is necessary to delete all index entries from the index structure. While some index structures can remove their structure themselves (e.g. trees), others may require as input all index entries (e.g. hash tables). To determine these index entries, the initialization module again interacts with the maintenance module.

Index maintenance. The index engine needs to keep indices consistent with document updates. For this purpose, the execution engine forwards the nodes to be inserted, deleted or modified to the index engine. Based on index patterns, the index engine can determine relevant index entries for an index without requiring knowledge about how the index is internally organized. It then forwards the maintenance operation along with relevant index entries to affected index structures. Chapter 6 provides details about how to extract relevant index entries from document updates.

Index access. To access an index, the index engine receives as input the search conditions to be evaluated on the index. A search condition can be dynamic if an index takes the result of another index as input (cf. path hierarchy indices). In this case, the search key is not known at query compile time, but after executing part of the query. The index engine forwards the search configuration to the index structure and returns matching nodes to the execution engine.

Index structures insert, update and delete index entries and process search configurations. The index engine passes entire index entries and search configurations to index structures. In case of index nesting, these index entries and search configurations refer to several index structures. With the help of the index configuration, each index structure can extract the index keys which it needs to index as well as the search conditions which it has to evaluate. To perform the required operations on all index structures, each index structure has to interact with its nested index structure, which it can determine with the help of the index configuration. After completing a maintenance or retrieval operation, it forwards the operation to the nested index structure.

5.5 Summary

To provide flexibility in indexing, an XML database requires an index framework that allows for processing arbitrary indices. Index processing comprises the tasks of selecting appropriate indices for queries, accessing and maintaining indices. With the help of a generic index model, the index framework presented in this chapter can perform these tasks without depending on specific index structures. It therefore offers the flexibility to adapt indices to the query workload as well as to easily extend and add index structures.

The index model logically represents each index as an annotated tree pattern. The tree pattern uses index variables to specify the properties to use as index keys and defines the nodes to be returned. Search configurations contain search conditions on these variables. Index patterns and search configurations do not depend on underlying index structures. Only the index configuration knows about index structures and maps each index to its physical index structures.

To process indices, a database needs to extend the query optimizer and the execution engine. The query optimizer selects appropriate indices for queries based on index patterns. The execution engine accesses and maintains indices. For this purpose, it uses index patterns to determine index entries that are relevant for maintenance operations. Based on index configurations, it forwards retrieval and maintenance operations to affected index structures.

Based on the index model, the index framework can select, access and maintain indices. When extending or adding index structures, no change in the query optimizer or the execution engine is required as long as the new indices can be represented by the index model.

Chapter 6

Index Maintenance

Contents

This chapter presents an index maintenance algorithm to propagate relevant updates to affected index structures when updating documents. After giving a general introduction in Section 6.1, Section 6.2 reviews related work on index maintenance. Section 6.3 presents the main concepts of the proposed algorithm, which is described in detail in Section 6.4. Possible extensions of the algorithm are discussed in Section 6.5. Finally, Section 6.6 summarizes the main ideas of the algorithm.

6.1 Introduction

To keep indices consistent with updates on documents, databases need to propagate relevant updates to affected index structures. In XML, updating comprises inserting, deleting or modifying document fragments. We refer to such fragments as update fragments and to their nodes as update nodes. Index maintenance is the process of extracting index entries from these update fragments based on index definitions. It also comprises determining relevant index entries when creating or deleting indices on existing documents.

SCIENS provides selective indices, which can be defined on arbitrary parts of a document. Selective indices imply that update fragments need not correspond to indexed fragments. They either contain more nodes than are required for index maintenance, or miss required nodes. Therefore, the database needs to determine which nodes of the update fragment affect an index and determine all nodes which are part of index entries.

Example 6.1 (index update) *Consider a multidimensional value index on resource titles and resource dates, returning resource labels. When adding a new resource, the maintenance algorithm needs to extract the title, date and resource label from the new resource and insert the corresponding index entry into the index structure. If the title of the resource changes, the index also needs to be updated. As in this case only the title changes, the maintenance algorithm has to determine the date value and the resource label to modify the index entry.*

The index definition specifies the fragment to be indexed and is represented as a tree pattern in SCIENS (cf. Subsection 5.3.3). Assume that an index maps the value of a node to the label of the node. In this case, the index pattern only consists of one pattern node. Each node which matches the pattern node affects the index and constitutes an index entry. In case of path patterns, the nodes of an index entry are in an ancestor-descendant relationship. For example, an index can map the value of a node to an ancestor of the node. As soon as one of the nodes changes, all nodes on the indexed path are required to perform the maintenance operation. The most difficult case arises when the index definition has the form of a tree pattern. An example is an index on two different values, e.g. on the date and title of resources. In this case, the nodes of an index entry have different paths. To retrieve all structurally related nodes of an index entry, several navigations are necessary.

To be applicable to SCIENS, the index maintenance algorithm should support arbitrary patterns and index structures. Many indexing approaches propose proprietary maintenance algorithms. These algorithms take into account what the index is defined on and thus can only handle specific index patterns. SCIENS bases on the idea of decoupling index structures and index patterns. One index structure can index various patterns and one pattern can be indexed by various index structures. To ensure this extensibility, the maintenance algorithm must neither depend on specific index structures nor on specific index patterns.

As update fragments need not correspond to indexed fragments, the maintenance algorithm has to determine the nodes that are required for performing the maintenance operation. For this purpose, it can (i) maintain relevant nodes in an auxiliary data structure or (ii) retrieve relevant nodes from the corresponding document. Alternative (i) needs to maintain one auxiliary data structure for each index, which increases size and maintenance overhead. As each index requires a different auxiliary data structure, it also impedes flexibility in indexing. Alternative (ii) is referred to as issuing source queries on base data. Which queries are necessary can be inferred from index patterns. This alternative may require a large number of queries, but can handle arbitrary indices. The number of queries can be reduced by extracting required nodes from update fragments instead of querying them from documents.

SCIENS requires a generic index maintenance algorithm that can handle arbitrary index patterns and index structures. It therefore cannot use auxiliary data structures to determine relevant nodes, but has to query relevant nodes. In XML, updates usually do not comprise single nodes, but document fragments. To improve update performance, the maintenance algorithm should extract relevant nodes from update fragments instead of querying them from base data. Further, integrating schema information improves update performance. As SCIENS is based on a schema-aware labeling scheme, the algorithm should exploit the schema to be more efficient.

6.2 Related Work

This section reviews related work on index maintenance. Indices are well-established in databases and each database needs to maintain its indices. Thereby, the data model influences which indices are supported as well as how they can be maintained. Subsection 6.2.1 looks at maintaining indices in (object-) relational databases and Subsection 6.2.2 considers object-oriented databases. The focus of Subsection 6.2.3 is on existing approaches for XML index maintenance. As views are related to indices and need to be updated as well, Subsection 6.2.4 reviews related work on XML view maintenance. Finally, Subsection 6.2.5 compares current approaches and relates them to SCIENS.

6.2.1 Maintaining Indices in Object-Relational Databases

Relational databases support indices on values defined on columns of relations. Each value of an indexed column represents an index key. Updates in relational databases are investigated e.g. by Schkolnick and Tiberio [172]. Denuit et al. [63] consider updates on secondary indices and views defined on complex types. Basically, an update affects an index if the index references one or more columns being updated. In case that an index may only refer to columns of one relation, the index simply needs to be updated when the values of these columns change due to the insertion, deletion or modification of a tuple. The more difficult case arises when an index may be defined on columns of different relations according to a join condition. If the attribute of one relation changes, in the worst case the other relation needs to be scanned entirely to find the attributes that match the join condition. Relations do not support direct navigation from the tuples of one relation to structurally related tuples of another relation. Such navigation can at most be supported by join indices [151]. Updating indices on multiple relations is especially considered in the context of data warehouses (e.g. [124]).

Example 6.2 (relational index update) *Consider a milestone relation, which assigns a unique id to each milestone, and a resource relation with a* *foreign key on the milestone id. We define (i) an index on the resource date and (ii) a multidimensional index on the resource date and the milestone title. Inserting a new resource triggers the insertion of a new attribute into the date column of the resource relation. This attribute is required for maintaining index (i). With regard to index (ii), a query on the milestone relation is necessary to find the corresponding milestone title before inserting the date and title into the index.*

Extensible databases (cf. Subsection 4.2.1) support adding index structures to databases, but leave the task of updating these indices to the developer. In DB2 database extenders [64], the developer needs to specify a function (called key transform) which takes values of indexed columns as input and generates index keys from these values. With regard to Oracle data cartridges [184], the developer defines methods for insertion, deletion and update that are specific to each index structure. Whenever a base table changes, these methods are invoked with the old and new values of the indexed columns and their tuple identifiers.

6.2.2 Maintaining Object-Oriented Indices

Object-oriented databases incorporate the nested structure between objects into index structures (cf. Subsection 4.2.1). As long as indices only refer to object attributes, updates can be processed similarly to relational databases. However, object-oriented databases typically support indices on aggregation graphs, which index objects on values of nested objects. When one of the objects changes, all objects on the indexed path are required to perform the maintenance operation. For this purpose, specific index structures have been developed [28], which can be classified into two categories. (i) The multi-index and the join index maintain all relevant objects in the index structure, which are accessed when processing updates. (ii) The nested index and the path index need to perform queries, i.e. traverse nested objects, to determine relevant objects. In each case, the maintenance process depends on the underlying data structure.

Example 6.3 (object-oriented index update) *Consider a nested index on the title of resource nodes, which returns for each title value the labels of related milestone nodes (cf. Figure 4.2). When adding a title to a resource, a query is necessary to determine the corresponding milestone label. This query can be performed by following backward references from the resource object to the milestone object.*

To maintain object-oriented indices regardless of underlying data structures, Henrich [102] proposes index update definitions. There is one index update definition for each class and each index structure which may be affected when updating an object belonging to this class. The index definition consists of (i) a description of the event, which causes the update, (ii) a reference to the index structure, for which one or more entries have to be updated, (iii) a query determining relevant objects to generate index entries, (iv) the necessary index actualization. When processing an update, the index definitions associated with the class of the updated object enable the database to generate index entries for affected index structures.

6.2.3 XML Index Maintenance

Several XML index structures incorporate the hierarchical structure of XML documents (cf. Subsection 4.2.2) and thus need to consider relationships between nodes when propagating updates to index structures. Various algorithms have been proposed to update structural indices, e.g. [112, 201]. However, these algorithms are tailored to specific index structures and do not support the maintenance of arbitrary indices.

Various native XML databases [47] provide simple structural and value indices. To determine whether an update affects an index, they compare the labelpath of the update node with the path of the index definition via query containment techniques (e.g. [149, 154, 177]) or path subsetting [188]. However, they do not support more complex indices whose index definitions have the form of path or tree patterns.

Example 6.4 (eXist index update) *In the native XML database eXist [2], one can define a secondary index on the value of date nodes, which returns the labels of these nodes. When processing an update, the database simply needs to compare the name of the update node with the node name of the index definition, which both refer to `@date`. As long as no navigation between indexed nodes is supported, index maintenance remains trivial.*

The KeyX approach of Hammerschmidt et al. [94, 95] supports selective XML indexing. It represents each index definition as a set of linearized path expressions, which are comparable to index patterns (cf. Subsection 5.2). To maintain indices, KeyX compares the nodes that are inserted, deleted or modified with the index definitions. More precisely, it compares each update node with each index definition to determine whether the node affects the index. If this is the case, it executes queries to navigate to all nodes which are part of the index entry. More precisely, for each update node and each index definition, KeyX performs the following steps. (i) Check whether the update node is either a key, a qualifier or the return of an index by comparing the labelpath of the update node with each linearization of the index definition through path intersection. In case that the path intersection is not empty, i.e. both paths select common nodes, (ii) calculate relative path expressions to the remaining keys, qualifiers or the return of the index definition. Then execute queries on these paths to retrieve all nodes that are part of the index entry. (iii) Each combination of qualifier, return and key constitutes an index entry and is forwarded to the corresponding index structure for maintenance. There also exists an index creation algorithm for KeyX. When creating an index on an existing document, it queries the document to determine associated index entries.

Example 6.5 (KeyX index update) *Consider a multidimensional value index on the date and title of resources, returning resource labels. KeyX represents the corresponding index definition with one path expression for each key, namely `//resource/title/text()` and `//resource/@date`, and one for the return, which is `//resource`. Assume that we add a new resource with a date and a title. The KeyX approach performs the following steps for each node of the update fragment, starting with the resource node. (i) It first determines whether the labelpath of the resource node intersects with any path of the index definition. As the intersection with the return of the index definition is not empty, the resource* *node is part of an index entry. (ii) To retrieve the corresponding date and title, it executes queries on the keys of the index definition. (iii) It then inserts the index entry, which maps the title and date value to the resource label, into the index. The same steps are performed for the remaining nodes of the update fragment, i.e. for the date and title node, yielding the same index entry. The approach needs to execute two queries for each affected update node (resource, date and title). As it considers each node individually, it needs to execute six queries in total and generates the same index entry three times. However, the update fragment contains all nodes that are relevant for the maintenance operation and therefore no query would be necessary.*

Sanders [171] also considers updates on multidimensional indices, referred to as compound indices. This approach associates with each node name the index definitions that include the node name. When updating a node, the following steps are performed for each associated index definition. The sets of index keys that exist in the document prior to and after the update are calculated. These sets are referred to as before keys and after keys, respectively. Then the difference between these sets is calculated and maintained. Calculating before and after keys consists of (i) collecting the set of all relevant nodes in the document and (ii) combining them into index entries based on structural relationships. To reduce the number of nodes considered, only nodes that relate to the update node according to the index definition are collected.

6.2.4 XML View Maintenance

Maintaining XML views is closely related to maintaining XML indices since views need to be updated as well when base data changes. Several approaches have been proposed to incrementally maintain XML views instead of recomputing the view. Thereby, the main assumption is that incremental maintenance is more efficient than recomputation. To incrementally maintain a view, the following steps are necessary: (i) determine whether an update affects a view, (ii) select the nodes required for maintenance, and (iii) generate and execute the maintenance statement. The main difficulty in performing these steps is to reduce the number of accesses to base data. For this purpose, incremental view maintenance approaches either keep relevant nodes in auxiliary data structures [9, 48, 68, 74] or only support updating views for which required nodes are known [136, 156].

Abiteboul et al. [9] propose to maintain an auxiliary data structure for each view. The auxiliary data structure consists of the object identifiers which are required for determining whether an update is relevant. Chen et al. [48] extend this approach and use as auxiliary data structure an aggregate path index (APIX). As this index also encodes structural information, it helps reducing the number of accesses to base data. To keep all relevant nodes in the auxiliary data structure, El-Sayed et al. [68, 74] materialize intermediate query results of the algebraic operator tree, which defines the view, in an auxiliary data structure. This data structure can then process updates with the help of propagation rules. All of these approaches have the disadvantage that auxiliary data structures grow rapidly and need to be maintained as well when base data changes.

Liefke and Davidson [136] and Nilekar [156] propose maintenance approaches which only depend on the view and the update, but neither require auxiliary data structures nor access to base data. However, these approaches are limited with regard to which updates they support over views. If an update requires nodes which are not part of the update or the view, they cannot update the view.

6.2.5 Comparison

XML index maintenance approaches cannot directly adopt relational maintenance approaches as they need to consider structural relationships between indexed nodes. Extensible object-relational databases leave the task of index maintenance to the developer. The maintenance process needs to be reimplemented for each specific index structure, which naturally causes redundancy. As maintenance depends on the specific index definitions, this approach impedes extending index structures and is not applicable to SCIENS.

Object-oriented and XML indices consider structural relationships between objects and nodes, respectively, and thus require more complex maintenance algorithms. Object-oriented maintenance approaches traverse object references to generate index entries. They only support path patterns, but not tree patterns. While the first approaches for XML index maintenance only considered path patterns as well, they have recently been extended to tree patterns. Basically, they determine for each update node whether it affects an index and then execute queries to retrieve all nodes which are required for the maintenance operation. XML view maintenance approaches equally query relevant nodes or maintain them in auxiliary data structures.

As SCIENS cannot maintain auxiliary data structures and requires a generic maintenance algorithm, it could adopt existing XML index maintenance approaches that query relevant nodes. The main drawback of these approaches is that they consider each update node individually and thus may need to execute a large number of queries. An important observation is that by extracting relevant nodes from update fragments instead of querying them, the number of queries can be greatly reduced. Existing maintenance approaches do not exploit schema information. However, integrating schema information does not only improve processing queries (cf. Subsection 2.1.2), but can also accelerate updates. In contrast to existing approaches, the maintenance algorithm of SCIENS should therefore exploit update fragments and schema information.

6.3 Concepts

This section presents the main concepts of the index maintenance approach of SCIENS, which we have published in [89]. The main idea of our approach is to extract index entries from update fragments based on index patterns. Index patterns uncouple the algorithm from specific index structures and guarantee support for indices defined on arbitrary document fragments. By exploiting the structure of update fragments, the proposed algorithm minimizes the number of additional queries without relying on auxiliary data structures. It only needs to retrieve those nodes from the database that are required for indexing but are not contained in the update fragment. If all index entries can be inferred from the update fragment, the algorithm is completely self-maintainable. The algorithm also supports determining relevant index entries when creating or deleting indices on existing documents. The developed techniques are not restricted to index maintenance, but can be carried over to the maintenance of caches, views or related problems.

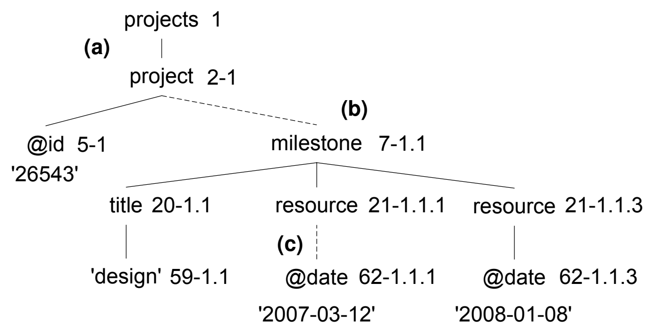


Figure 6.1: Update fragments to update the entire document (a), a milestone (b) and a date (c).

Example 6.6 (index updates) *Assume that we successively insert fragments (a)-(c) of Figure 6.1 and propagate updates to the indices whose patterns are shown in Figure 6.2. After executing the update on the document, each node of the update fragment has assigned a unique label, which encodes the position of the node in the document. The insertion of fragment (a) only affects the first index, which needs to be updated with index entry* $((\text{'26543'}) \rightarrow (5\text{–}1))$. *When inserting fragment (b), we need to insert index entry* $((\text{'2008-01-08'}) \rightarrow (21\text{–}1.1.3))$ *into the second index. The update further affects the third index, in which we need to insert the index entries* $((\text{'design'}, \perp) \rightarrow (21\text{–}1.1.1))$ *and* $((\text{'design'}, \text{'2008-01-08'}) \rightarrow (21\text{–}1.1.3))$. *Note that the date is required in the second index and is optional in the third index. As the first resource still does not have a date, it is not part of the second index. Now let us add a date to the first resource by inserting fragment (c). The insertion affects both the second and the third index. In each case, the insertion is not self-maintainable. We first need to determine the resource label belonging to the date node. We can then insert a new index entry into the second index, namely* $((\text{'2007-03-12'}) \rightarrow (21\text{–}1.1.1))$. *With regard to the third index, the update also requires the title of the milestone. In this case, we do not need to insert a new but update an existing index entry. More precisely, we have to update index entry* $((\text{'design'}, \perp) \rightarrow (21\text{–}1.1.1))$ *by replacing the null value with the value of the new date node. Note that only the update of fragment (c) depends on executing queries on base data to determine all nodes required for maintenance. In the other cases, relevant index entries can be extracted from update fragments.*

To extract index entries from update fragments, the algorithm adopts ideas from XML query processing techniques. XML pattern matching [42, 50, 142, 200] and tree inclusion [31, 53] algorithms represent queries as tree patterns and find the nodes in the document that match the pattern. Each subtree that structurally corresponds to a pattern is referred to as embedding. Pattern matching

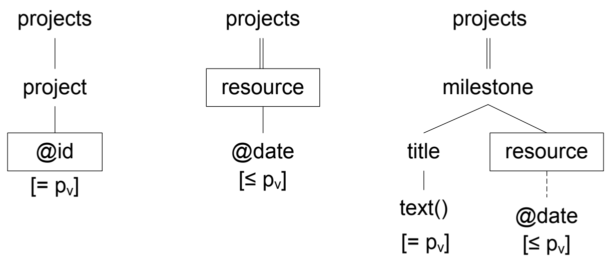


Figure 6.2: Simple index pattern on project ids (left), path index pattern on resource dates (middle) and twig index pattern on milestone titles and resource dates (right).

algorithms are based on index structures and labeling schemes. Tree inclusion algorithms traverse whole documents to find embeddings. XML filtering techniques [45, 66] have been developed for publish/subscribe systems to evaluate several queries, represented as patterns, by traversing the document only once. They do not return embeddings, but only determine the documents that match the queries.

The index maintenance approach of SCIENS finds embeddings of index patterns in update fragments and then generates index entries from these embeddings. Each embedding consists of nodes in the document that structurally correspond to the index pattern and represents an index entry. While it reuses some ideas of query processing techniques to find embeddings, it has to consider several differences: (i) It needs to find embeddings in update fragments, but should not traverse whole documents or access auxiliary data structures for this purpose. (ii) If the update fragment misses nodes that are required for indexing, it needs to issue source queries. (iii) It needs to associate the index keys with the nodes to be returned and generate index entries from embeddings. Dependent on the update operation, the index structures can then insert, delete or modify the index entries.

To extract index entries from update fragments based on index patterns, the proposed algorithm performs the following steps:

1. Find embeddings of index patterns in update fragments.
2. Execute queries to retrieve the nodes that are not contained in the update fragment but are required for generating index entries.
3. Generate index entries from embeddings by extracting the index keys and the nodes to be returned. Forward the generated index entries to the specific index structures, which insert, delete or modify the generated index entries.

In the following, we present the main concepts of the maintenance algorithm. We start with update fragments in Subsection 6.3.1, define embeddings in Subsection 6.3.2 and present a stack encoding for embeddings in Subsection 6.3.3. Subsection 6.3.4 describes how to improve finding embeddings and reduce the size of the stack encoding by integrating schema information into index patterns. Subsection 6.3.5 presents the main concepts for querying required nodes and Subsection 6.3.6 for generating index entries.

6.3.1 Update Fragments

Each update (insertion, deletion, modification) consists of an update operation that specifies the location of the update in the document. The update operation corresponds to a query that returns the nodes relative to which the update is performed. In case of a deletion, each fragment returned by the update operation represents an update fragment that needs to be deleted. In case of an insertion, the update additionally contains the fragment to be inserted. This fragment constitutes the update fragment that is inserted relative to the nodes returned by the update operation. Each modification also specifies an update fragment that replaces the nodes returned by the update operation. In the following, we assume that a modification is executed as a deletion followed by an insertion. Although modifying index entries instead of deleting and reinserting them may be more efficient, we omit modifications for ease of presentation. The algorithm could however easily be extended to handle modifications as well.

Each update operation specifies the path from the root of the document to the root of each update fragment. The maintenance algorithm receives as input the update fragments after executing the update operation. Each node of the update fragments therefore has assigned a unique label, which implies that the algorithm knows the path of each node in the update fragment. In the following, each update fragment is a tree of labeled nodes that are inserted into or deleted from a document.

6.3.2 Embeddings

To determine relevant index entries, the maintenance algorithm finds embeddings of index patterns in update fragments. Nodes of a document that structurally correspond to the pattern nodes of an index pattern are referred to as embedding. This means that their names, parent-child and ancestor-descendant relationships must comply. In case that a pattern node is required, there must exist a corresponding node in the document. If it is optional, it has associated a corresponding node in the document only if this node exists.

Definition 6.1 (embedding) An *embedding* of an index pattern $\mathsf{P} = (\mathsf{N}, \mathsf{F})$ in a document $D = (N, F)$ is a mapping from N to N. The edges of an index pattern can be classified into parent-child and ancestor-descendant edges, $\mathsf{F} = \mathsf{F}_{/} \cup \mathsf{F}_{//}$ (cf. Definition 5.5). Functions *parent* and *ancestor* determine whether nodes N in the document are connected via edges F (cf. Subsection 2.1.1). An embedding is defined by partial function $emb : \mathsf{N} \rightarrow N$, such that $\forall \mathsf{x}, \mathsf{y} \in \mathsf{N}$:

(a) if $req(\mathsf{x}) = 1 \rightarrow emb(\mathsf{x})$ is defined

(b) if $(\mathsf{x}, \mathsf{y}) \in \mathsf{F}_{/} \rightarrow emb(\mathsf{x}) = parent(emb(\mathsf{y}))$

(c) if $(\mathsf{x}, \mathsf{y}) \in \mathsf{F}_{//} \rightarrow emb(\mathsf{x}) \in ancestor(emb(\mathsf{y}))$

□

Nodes of a document that represent an embedding are called *matching nodes*. The pattern nodes that match a node are referred to as *matching pattern nodes*. Note that an embedding need not be an injective function as a node of the document may match several pattern nodes.

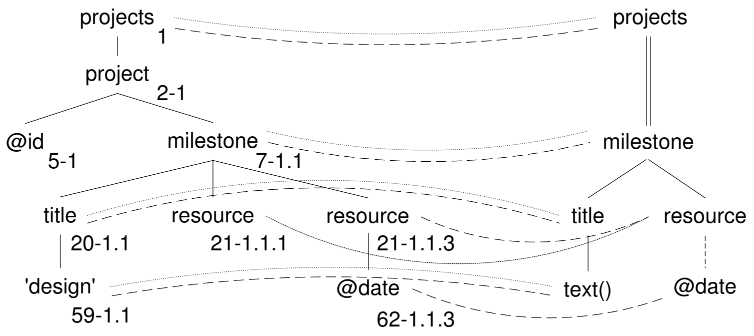


Figure 6.3: Embeddings of an index pattern (right) in a document (left).

Example 6.7 (embedding) *Figure 6.3 depicts two embeddings of an index pattern in a document. Each embedding associates each required pattern node with one node of the document. The first embedding, depicted with dotted lines, consists of the nodes with labels (*1, 7–1.1, 20–1.1, 59–1.1, 21–1.1.1*). Note that pattern node* `@date` *is optional. Thus the embedding need not contain a matching node for this pattern node. The second embedding, depicted with dashed lines, consists of the nodes with labels (*1, 7–1.1, 20–1.1, 59–1.1, 21–1.1.3, 62–1.1.3*).*

We refer to an embedding which still misses matching nodes as a *partial embedding* and to its matching nodes as *candidate nodes*. A partial embedding is *complete* as soon as there exists a matching node for each required pattern node of the corresponding index pattern. If there does not exist a matching node for each required pattern node, we say that the embedding and its candidate matching nodes are *false positives*.

Example 6.8 (false positive) *Considering the second index pattern of Figure 6.2 and update fragment (b) of Figure 6.1, node* 21–1.1.1 *is a candidate node as it matches pattern node* `resource`*. However, before inserting fragment (c), this partial embedding misses a matching node for the required* `@date` *pattern node and thus represents a false positive.*

Note that we associated each index pattern with one document in Subsection 5.3.3. When defining an index on a collection of documents, the index pattern has embeddings in each document of the collection.

6.3.3 Stack Encoding

When traversing a document to extract embeddings, embeddings do not appear in a linear order. Instead, successive nodes may belong to different embeddings, or some nodes may be part of several embeddings. It is therefore necessary to keep matching nodes in an intermediate data structure that encodes embeddings. For this purpose, it is possible to reuse stack encodings proposed by pattern matching algorithms (cf. e.g. [200]). This data structure associates with each pattern node of an index pattern a stack. Each entry in a stack consists of a matching node from the document and a pointer to its ancestor node in the stack of the parent pattern node, encoding the structural relationships between the nodes.

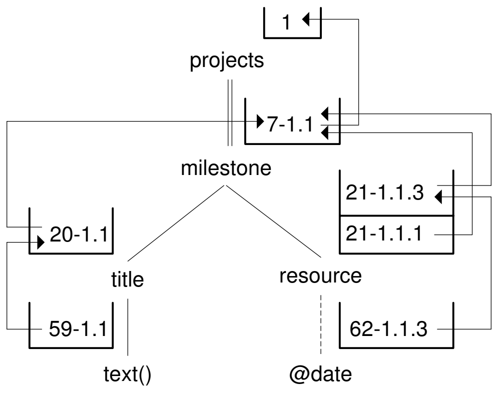


Figure 6.4: Stack encoding of the embeddings in Figure 6.3.

Example 6.9 *Figure 6.4 depicts the stack encoding of the embeddings of Figure 6.3. Some nodes belong to both embeddings, e.g. node* 20–1.1*. It is therefore not possible to remove this node from the stack as long as not all embeddings that contain this node have been found. The two embeddings differ in their nodes matching the* `resource` *and* `@date` *pattern nodes. We know that date* 62–1.1.3 *belongs to resource* 21–1.1.3 *because of the pointer between these nodes. Note that in this case, the first resource could be removed from the stack when matching the second resource because its embedding is already complete. It might therefore seem sufficient if one stack only contains one matching node at the same time and that keeping references is not necessary. However, imagine that each milestone can have several titles. In this case, all titles of a milestone are found before any resources of the milestone. As each resource forms one embedding with each title, it is indispensable to keep all titles in the stack.*

6.3.4 Schema-aware Index Patterns

All embeddings have the same structure as the index pattern. Integrating structural information from the schema facilitates finding embeddings. For this purpose, this subsection presents schema-aware and minimal index patterns.

Each document has associated a schema in SCIENS. A valid index pattern only refers to paths that exist in the document, i.e. that have been defined in the schema. Therefore each path in the index pattern matches one or several path schemas of the schema tree. By associating matching path schemas with pattern nodes, schema information can be exploited when finding embeddings.

Matching path schemas of pattern nodes can be determined by finding embeddings of the index pattern in the schema tree. For this purpose, existing query pattern matching algorithms can be used [42, 50, 142, 200]. Each matching path schema of a pattern node has one ancestor path schema in the corresponding parent pattern node. One path schema could have several ancestor path schemas in the parent pattern node only if we allowed recursion in the schema or wildcards in the index pattern.

A schema-aware index pattern associates matching path schemas with each pattern node and uses one stack for each matching path schema. This implies that each pattern node has as many stacks as it has matching path schemas. To be valid, each pattern node must have at least one matching path schema.

Definition 6.2 (schema-aware index pattern) A *schema-aware index pattern* $\mathsf{P_S} = (\mathsf{N}, \mathsf{F}, name, var, req, root, return, doc, paths, stack, pstack, \mathsf{L}, \mathsf{V}, D, \mathsf{S}, \mathbf{P})$ extends an index pattern P, as defined in Definition 5.5, by a set of stacks S and by the set of path schemas $\mathbf{P}$ which are taken from the schema $\mathbf{S}$ of the document D on which the index pattern is defined. It further defines functions *paths* and *stack* and partial function *pstack*.

- Function $paths : \mathsf{N} \rightarrow 2^{\mathbf{P}}$ associates each pattern node with matching path schemas.
- Function $stack : (\mathsf{N}, \mathbf{P}) \rightarrow \mathsf{S}$ maps each matching path schema of each pattern node to a stack such that $\forall \mathsf{n} \in \mathsf{N} : \forall \mathbf{p} \in paths(\mathsf{n}) : stack(\mathsf{n}, \mathbf{p}) \in \mathsf{S}$.
- Partial function $pstack : (\mathsf{N}, \mathbf{P}) \rightarrow \mathsf{S}$ relates each stack with a parent stack, such that the parent stack is associated with a parent pattern node and an ancestor path schema, i.e. $\forall \mathsf{n} \in (\mathsf{N} \setminus root(\mathsf{P_S})), \mathsf{n}' \in \mathsf{N}, \mathbf{p} \in paths(\mathsf{n}), \mathbf{p}' \in paths(\mathsf{n}') : parent(\mathsf{n}) = \mathsf{n}' \wedge \mathbf{p}' \in ancestor(\mathbf{p}) \rightarrow pstack(\mathsf{n}, \mathbf{p}) = stack(\mathsf{n}', \mathbf{p}')$.

□

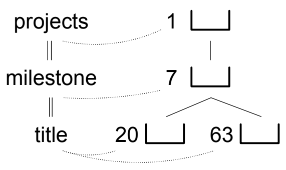


Figure 6.5: Schema-aware index pattern on milestone and resource titles.

Example 6.10 (schema-aware index pattern) *The schema-aware index pattern of Figure 6.5 depicts the labels of embedded path schemas and their stacks. The paths relationships are shown as dotted lines, pstack relationships as simple lines. As both milestones and resources have titles, the* `title` *pattern node has two matching path schemas, namely the path schemas with labels* 20 *and* 63*.*

The number of matching nodes increases with the size of an index pattern. The more pattern nodes an index pattern has, the more matching nodes does the maintenance algorithm find and encode into the stacks. To extract index entries, only those nodes of an embedding are required that are either index keys or return nodes. We therefore propose to omit all pattern nodes of an index pattern that are redundant, i.e. that are not relevant for generating index entries. Using minimal index patterns improves the space complexity of the maintenance algorithm. Various approaches have been proposed to minimize tree patterns by finding smaller, equivalent patterns (cf. [14, 78, 165]). By exploiting schema information, it is similarly possible to reduce the size of index patterns.

Each index pattern must contain the return pattern node and the pattern nodes that have associated index variables. To preserve structural relationships, it requires branching pattern nodes. As pattern nodes can be required or optional, it must further contain those pattern nodes that separate required from optional fragments. All other pattern nodes are redundant and can be removed because they are not relevant for defining the index.

Definition 6.3 (redundant pattern node) A pattern node $\mathsf{n_r}$ of a schema-aware index pattern $\mathsf{P_S}$ with pattern nodes N, $\mathsf{n_r} \in \mathsf{N}$, is *redundant* if it fulfills each one of the following conditions:

(a) $\mathsf{n_r} \neq \mathit{return}(\mathsf{P_S})$ (the pattern node is not the return pattern node)

(b) $\mathit{var}(\mathsf{n_r}) = \emptyset$ (the pattern node has not associated index variables)

(c) $(\forall \mathsf{n_1}, \mathsf{n_2} \in \mathit{children}(\mathsf{n_r}) : \mathsf{n_1} = \mathsf{n_2}) \vee (\mathit{children}(\mathsf{n_r}) = \emptyset \wedge \mathit{req}(\mathsf{n_r}) = 0)$ (the pattern node has exactly one child pattern node or it is an optional leaf)

(d) $\forall \mathsf{n_1} \in \mathit{children}(\mathsf{n_r}) : \mathit{req}(\mathsf{n_r}) = \mathit{req}(\mathsf{n_1})$ (the pattern node and its child are either both required or both optional)

□

The size of a schema-aware index pattern can be reduced by removing redundant pattern nodes. A minimal index pattern can be obtained from a schema-aware index pattern by removing each redundant pattern node and its associated edges and inserting an ancestor-descendant edge between its parent and its children. Note that each pattern node of a schema-aware index pattern has associated path schemas. Therefore it is possible to omit redundant pattern nodes and insert ancestor-descendant edges without changing the semantics of the index pattern. Minimization could be further enhanced by also taking into account the cardinality of node schemas. In the following, we assume that each index pattern is schema-aware and minimal.

Definition 6.4 (minimal index pattern) A *minimal index pattern* is a schema-aware index pattern that does not contain redundant pattern nodes.

Example 6.11 (minimal index pattern) *Figure 6.6 depicts a non-minimal index pattern on the left and its minimal version on the right. Each pattern node has associated the labels of matching path schemas. The non-redundant pattern nodes are the branching pattern node `milestone`, the return pattern node `resource` and the pattern nodes `text()` and `@date` that have associated index variables. The minimal index pattern contains an ancestor-descendant edge between pattern nodes `milestone` and `text()`. Nevertheless, only text nodes whose parents are `title` nodes (but e.g. not `description` nodes) will match the `text()` pattern node because of its associated path schema.*

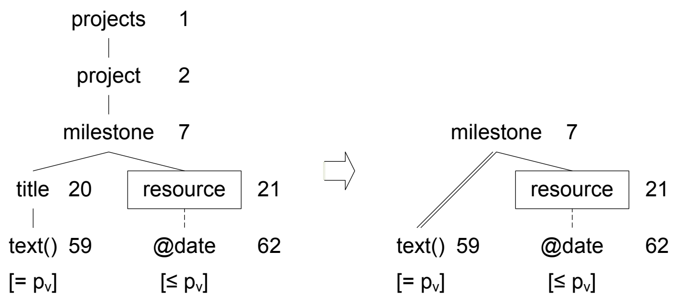


Figure 6.6: Non-minimal (left) and minimal (right) index pattern on milestone titles and resource dates.

6.3.5 Queries

In the previous subsection, we defined embeddings of index patterns in documents. Updates usually do not refer to entire documents, but to fragments only. When finding embeddings of an index pattern in an update fragment, each embedding may miss nodes that have not been contained in the update fragment. However, it is necessary to find all nodes of affected embeddings before generating index entries. To determine missing nodes, it is necessary to execute source queries on the document.

An embedding may miss ancestors of the update fragment or any descendant of these ancestors (except for the descendants that constitute the update fragment itself). If it misses nodes, at least one matching node, which is not associated with the root pattern node, has not associated an ancestor in the stack encoding. In this case, the ancestors and (possibly) the descendants of these ancestors have to be queried. As the index pattern associates path schemas with pattern nodes, it is possible to generate source queries for missing nodes.

Example 6.12 (queries) *Assume that we delete a resource and update the third index of Figure 6.2. The resource node matches pattern node* `resource`*. However, the update fragment misses the milestone title, which needs to be queried. As the resource node has not associated an ancestor, we first need to execute a query to determine its parent, which matches the* `milestone` *pattern node. Departing from this node, we can then query the missing title.*

The efficiency of this step can be improved with labeling schemes that allow for navigating along certain axes without accessing base data (cf. Chapter 3). The update of whole documents is always self-maintainable as no queries are executed. The update of simple index patterns, which only consist of one pattern node, is always self-maintainable as well.

When creating or deleting an index on an existing document, it is necessary to retrieve all embeddings from the document and generate index entries from these embeddings. For this purpose, it is likewise necessary to query all matching nodes based on the index pattern. Therefore, the same algorithm can be used as for querying missing nodes. Note that the step of generating index entries can be omitted when completely deleting tree-structured indices, such as a B+-tree. In contrast to e.g. a hash table, these index structures can retrieve and delete all pages themselves.

6.3.6 Index Entries

Each index consists of a set of index entries, mapping index keys to nodes in the document. These index entries are referred to as complete index entries in the following. When finding embeddings of index patterns in update fragments, each embedding consists of index keys and one return node. We refer to these index keys and the return node as a partial index entry. When inserting a fragment, the index has to insert the partial index entries into the complete index entries. If no index entry with the same index keys exists, it must generate a new complete index entry. Otherwise, it has to insert the return node into the index entry having the same index keys. Handling deletions is analogous.

Definition 6.5 (partial index entry) A partial index entry maps a list of index keys to one label, representing a return node. A complete index entry consists of as many partial index entries as it has return nodes.

Example 6.13 (partial index entry) *Assume that the third index of Figure 6.2 contains the index entry* ((*'*`design`*','2008-03-03'*) → (21–1.3.1, 21–7.3.5, 21–7.3.7)). *When inserting a new resource, we find a new embedding, e.g.* ((*'*`design`*','2008-03-03'*) → (21–5.5.3)). *This embedding represents a partial index entry. When inserting the partial index entry, we need to extend the complete index entry by adding the new return label* 21–5.5.3.

To generate partial index entries from embeddings, the maintenance algorithm needs to extract the embeddings from the stack encoding. The index entries need to be inserted into the index structure in case of an insert operation, and deleted in case of a delete operation. The update of index structures with index entries is not subject of this work, as each index structure provides specific algorithms for this purpose.

The set of return labels of a complete index entry may contain duplicates if several embeddings contain identical index keys. In this case, a list of index keys maps to the same return node multiple times. Duplicates are necessary to handle deletions correctly, as demonstrated below in Example 6.14. Similar ideas are used by counting algorithms for view maintenance [91]. The KeyX approach of Hammerschmidt (cf. Subsection 6.2.3) does not handle duplicates correctly because it may generate the same index entry several times (cf. Example 6.5). Note that we do not make any assumptions on how an index structure stores duplicates. For example, it can associate with a label the number of occurrences instead of storing the same label multiple times.

Example 6.14 (duplicates) *Consider the index on milestone and resource titles of Figure 6.5. Assume that a resource has the same title as its milestone. In this case, we find the same partial index entry twice, yielding e.g. the complete index entry* ((*'*`design`*'*) → (7–5.3, 7–5.3)). *When deleting the corresponding resource, we have to delete one partial index entry. As the milestone title still exists, we may only remove one milestone node from the complete index entry. Without duplicates, a query would be necessary to decide whether the index entry can be deleted, resulting in additional overhead.*

Special attention needs to be paid when an update operation affects an index key that is associated with an optional pattern node (remember that the return is always required). We refer to these index keys as optional index keys. The insertion/deletion of an optional index key does not trigger an insertion/deletion of the corresponding partial index entry. Instead, an existing partial index entry needs to be updated. An insertion of an optional key replaces a null value of one of its index keys, whereas a deletion generates a null value.

Example 6.15 (optional index key) *Considering the third index of Figure 6.2 whose pattern node `@date` is optional, assume that it contains the partial index entry* ((*'`design`'*, $\perp$) $\rightarrow$ (21–5.3.1))*. If we insert a new date with value '`2008-09-09`' into this milestone, we need to update the index entry by replacing the null value with the new date.*

An optional index key always stems from a matching node of an optional pattern node that has been contained in the update fragment and has not been queried. To insert a new optional index key, it is necessary to (i) replace the index key with a null value and delete this partial index entry from the index, (ii) restore the new index key in the index entry and insert it into the index. Handling several optional keys and deletions is analogous.

6.4 Maintenance Algorithm

The maintenance algorithm generates index entries based on index patterns and update fragments by performing the following steps:

1. Find embeddings of index patterns in update fragments and encode them into stacks associated with each index pattern.
2. Execute queries to retrieve nodes that are not contained in the update fragment but are part of embeddings.
3. Generate index entries from the embeddings by extracting index keys and the nodes to be returned. Forward the index entries to the index structures for insertion or deletion.

The algorithm uses an intermediate data structure in the form of a stack encoding to compactly store the nodes of an embedding up to the time of generating index entries. The index entries are then forwarded to the affected index structures, which update their data structure with proprietary algorithms. Insertions and deletions can be handled analogously as they only differ in the update operation executed on the index structure.

In the following, we assume that each update affects one update fragment and one index pattern as extending the algorithm to several fragments and patterns is straightforward. Note that in case of several index patterns, the algorithm has to process each index pattern individually. This could be improved by adopting ideas from XML filtering [45, 66] to share commonalities between patterns.

Procedure MAINTAIN maintains an index when inserting or deleting a document fragment. It receives as input the root node of the update fragment, the index pattern of the index to be maintained and the kind of update operation, which is either an insert or a delete operation. Algorithm 6.1 provides the details of the procedure. It basically executes the three steps which are necessary

Algorithm 6.1 Maintain an index with an update fragment based on its index pattern.

Input: Update node $n \in D$, which constitutes the root of the update fragment; Schema-aware, minimal index pattern P defined on the document D to which the update fragment belongs; Update operation $o \in \{i, d\}$, indicating whether the update is an insertion i or a deletion d.
Description: Find embeddings of the index pattern in the update fragment and encode them into the stacks associated with the index pattern. Then execute queries to retrieve missing nodes and push them onto the stacks of the index pattern. Finally, generate index entries from the embeddings that are encoded into the stacks of the index pattern and forward them to the index structure for insertion or deletion.
Result: The index with index pattern P has been updated with update node n and its descendants.

```
1: procedure MAINTAIN($n$, $\mathsf{P}$, $o$)
2:     FINDEMBEDDINGS($n, \mathsf{P}$)
3:     EXECUTEQUERIES($\mathsf{P}$)
4:     GENERATEINDEXENTRIES($\mathsf{P}, o$)
5: end procedure
```

to maintain an index. First it finds embeddings of the index pattern in the update fragment and encodes them into the stacks associated with the index pattern. It then executes queries to missing nodes and again stores these nodes in the stack encoding of the index pattern. The first two procedures therefore modify the index pattern by pushing matching nodes onto its stacks. Finally, it generates index entries from the embeddings that are encoded into the stacks of the index pattern. The algorithm calls several procedures, as depicted in Figure 6.7.

The following subsections describe each procedure of the algorithm in more detail, whereby focusing on step 1 and 2 of the algorithm. The algorithm need not execute these steps sequentially. Instead of first finding all embeddings and then generating index entries, it could also generate an index entry as soon as an embedding has been found. Although the latter would reduce the size of the intermediate data structure, we follow a sequential order for ease of presentation.

6.4.1 Find Embeddings

To find embeddings of an index pattern in an update fragment, the algorithm traverses the update fragment once in document order and compares its structure with the structure of the index pattern. If a node of the update fragment has the same path schema as is associated with a pattern node, it represents a candidate node and is put onto the corresponding stack. If a subfragment cannot be relevant for the index pattern, the algorithm does not further process its nodes. Discarding non-relevant subfragments improves the runtime complexity of the algorithm. By traversing the update fragment in document order, the algorithm finds candidate nodes of an embedding one after another. As soon as it detects that a partial embedding cannot become complete, it deletes the false

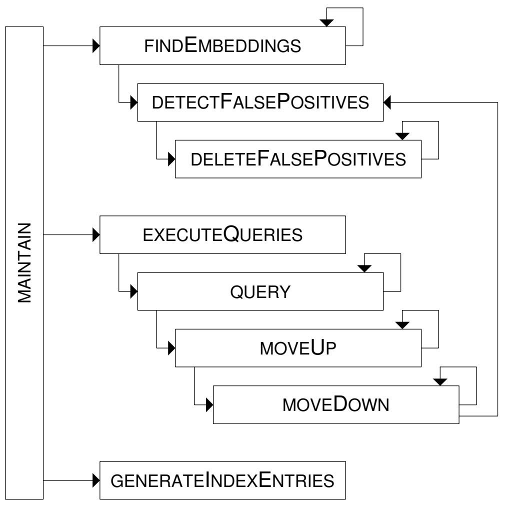


Figure 6.7: Maintenance algorithm.

positives from their stacks to keep space complexity, i.e. the number of nodes encoded into stacks, as small as possible.

The algorithm consists of three main procedures. Procedure FINDEMBEDDINGS matches the update fragment against the index pattern to find partial embeddings. Procedure DETECTFALSEPOSITIVES detects false positives and procedure DELETEFALSEPOSITIVES deletes partial embeddings that represent false positives. Figure 6.8 gives an abstract example to depict the interaction between the various procedures.

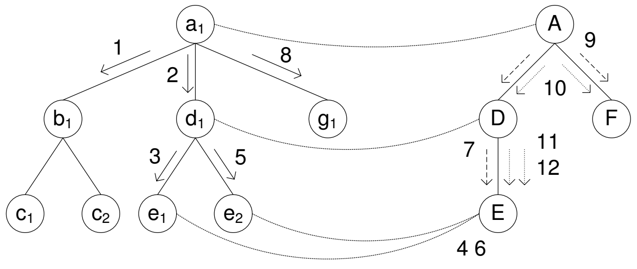


Figure 6.8: Find embeddings of index pattern (right) in update fragment (left).

Example 6.16 (find embeddings) *Figure 6.8 gives an abstract example of how to find embeddings of an index pattern in an update fragment. For simplicity, each node of the update fragment has a unique lower-case letter and each node of the index pattern a unique upper-case letter. Procedure* FINDEMBEDDINGS *traverses the update fragment (solid arrows). Procedures* DETECTFALSEPOSITIVES *and* DELETEFALSEPOSITIVES *traverse the stack encoding associated with the index pattern to find false positives (dashed arrows) and to delete them from the stack encoding (dotted arrows).*

Before describing each of these procedures in more detail, we define some more functions and methods:

- Function *topnode* : $\mathsf{S} \rightarrow N$ returns the top-most node associated with a stack, or null ($\perp$) if the stack is empty.
- Function *toppointer* : $\mathsf{S} \rightarrow N$ returns the node to which the top-most node of a stack points, or null ($\perp$) if such a node does not exist or if the top-most node has not associated a pointer.
- Method PUSH(n, s) puts node n onto stack s.
- Method POP(n, s) removes node n from stack s. Note that in our context, this node need not necessarily be the top-most node of the stack.
- Method CREATEPOINTER($n, \mathsf{s}, n_a, \mathsf{s_p}$) creates a pointer from node n associated with stack s to node n_a associated with stack $\mathsf{s_p}$, whereby stack $\mathsf{s_p}$ is the parent of stack s. If n_a is undefined, no pointer is created.

In the following, we will give a concrete algorithm for each procedure and demonstrate its usage based on the abstract example of Figure 6.8 and on the example of updating an index pattern of Figure 6.2 with update fragment (b) of Figure 6.1.

Find embeddings

Procedure FINDEMBEDDINGS recursively processes the nodes of an update fragment to find matching pattern nodes and encodes partial embeddings into stacks. To reduce runtime complexity, the algorithm only processes relevant fragments, i.e. that possibly contain matching nodes. For each node, it determines matching pattern nodes by comparing their path schemas. If a node has the same path schema as a pattern node, it represents a candidate node and is put onto the corresponding stack.

To determine whether a node is relevant for an index pattern and to find matching pattern nodes for a relevant node, the procedure uses functions *relevant* and *matches*. Let n be an update node of document D and P be an index pattern with pattern nodes N that is associated with document D. Each node in the document and each pattern node in the index pattern has associated path schemas, which are accessible via functions *pathschema* and *paths*, respectively. The set of path schemas $\mathbf{P}$ is taken from the schema to which the document D belongs.

- The fragment rooted at n is relevant for index pattern P, $relevant(n, \mathsf{P})$, iff the path schema of n is an ancestor-or-self of any path schema associated with P. $\exists \mathsf{n} \in \mathsf{N} : \exists \mathbf{p} \in paths(\mathsf{n}) : pathschema(n) = \mathbf{p} \vee pathschema(n) \in ancestor(\mathbf{p}) \leftrightarrow relevant(n, \mathsf{P})$.
- Function *matches* : $N \rightarrow 2^{\mathsf{N}}$ returns matching pattern nodes for a node, $matches(n) = \{\mathsf{n} \in \mathsf{N} \mid pathschema(n) \in paths(\mathsf{n})\}$.

Algorithm 6.2 Find embeddings of an index pattern in an update fragment.

Input: Update node n, which is the root of the update fragment; Schema-aware, minimal index pattern P.

Description: Find embeddings of index pattern P in the update fragment rooted at n and encode matching nodes into stacks associated with P.

Result: Update node n and its descendants that match index pattern P have been encoded into stacks associated with the pattern.

```
procedure findEmbeddings($n$,$\mathsf{P}$)
    if $relevant(n, \mathsf{P})$ then
        for all $\mathsf{n} \in matches(n)$ do
            $\mathsf{s} = stack(\mathsf{n}, pathschema(n))$
            push($n, \mathsf{s}$)
            if $\mathsf{n} \neq root(\mathsf{P})$ then
                $n_a = topnode(pstack(\mathsf{n}, pathschema(n)))$
                createPointer($n, \mathsf{s}, n_a, pstack(\mathsf{n}, patschema(n))$)
            end if
        end for
        for all $n_c \in children(n)$ do
            findEmbeddings($n_c, \mathsf{P}$)
        end for
        for all $\mathsf{n} \in matches(n)$ do
            detectFalsePositives($n, \mathsf{n}$)
        end for
    end if
end procedure
```

By integrating schema information, it is possible to pregenerate a list of relevant path schemas as well as a map from path schemas to matching pattern nodes, which avoids the need for comparing each node with each pattern node.

Algorithm 6.2 shows the details of procedure findEmbeddings. It takes as input a node n, representing the root of the currently processed update fragment, and an index pattern P. To determine whether the node matches the pattern, it invokes various functions on the node and the index pattern. With the help of function *relevant*, it determines whether the fragment rooted at n is relevant for the index pattern P. In case that the fragment is not relevant, it does not further process n or any of its descendants. Otherwise, it determines matching pattern nodes for n via function *matches*. Each matching pattern node n has associated one stack s with the same path schema as n. Procedure push puts node n onto this stack. Procedure createPointer creates a pointer from n to its ancestor in the parent stack to encode structural relationships. As the algorithm processes update fragments in document order, the ancestor must be the top-most node n_a of the parent stack. If the parent stack is still empty, the update fragment did not contain the ancestor and the ancestor is still undefined. After matching node n, the algorithm recursively invokes procedure findEmbeddings for each child n_c of n.

As soon as all descendants of node n have been processed, the detected partial embeddings must contain candidate nodes for required descendants. Otherwise the partial embeddings represent false positives, which is determined by calling procedure DETECTFALSEPOSITIVES for each matching pattern node.

Example 6.17 (find embeddings) *Consider the update fragment and the index pattern of Figure 6.8. Procedure* FINDEMBEDDINGS *recursively traverses the update fragment, starting with the root node a_1. After matching this node with pattern node A, the algorithm reinvokes the procedure with each child of a_1. Node b_1 is not relevant for the index pattern, and the algorithm does not further process its children. Node d_1 matches pattern node D and its children e_1 and e_2 match pattern node E. Every time when the algorithm has processed all children of a node, it invokes procedure* DETECTFALSEPOSITIVES*. Therefore, after having processed nodes e_1 and e_2 respectively, it calls this procedure (steps 4 and 6). After having processed the children of d_1, the algorithm again invokes procedure* DETECTFALSEPOSITIVES *(step 7). The final child of node a_1 is node g_1 which is not relevant for the pattern. Finally, the algorithm invokes procedure* DETECTFALSEPOSITIVES *one more time (step 9).*

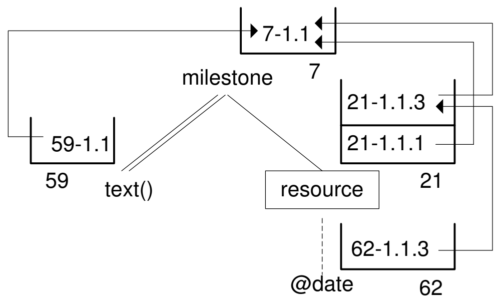


Figure 6.9: Embeddings of fragment (b) of Figure 6.1 encoded into stacks.

Example 6.18 (find milestone titles and resource dates) *Assume that we insert fragment (b) of Figure 6.1 and update the index pattern shown in Figure 6.9. The algorithm starts with processing node* 7–1.1*. The corresponding fragment is relevant, and the algorithm determines that the node matches pattern node* `milestone` *and puts it onto its stack. As pattern node* `milestone` *is the root pattern node, it need not create a pointer. Then it recursively processes the children, starting with node* 20–1.1*. This node is relevant, but it does not match any pattern node. The algorithm proceeds with node* 59–1.1*, which matches pattern node* `text()` *as it has the same path schema. After putting the node onto the corresponding stack, the algorithm creates a pointer to node* 7–1.1*, which is its ancestor in the parent stack. The algorithm then matches nodes* 21–1.1.1 *and* 21–1.1.3 *with pattern node* `resource` *and node* 62–1.1.3 *with pattern node* `@date`*.*

Detect false positives

Procedure DETECTFALSEPOSITIVES determines whether partial embeddings contain candidate nodes for required pattern nodes. It bases on the observation that after processing the descendants of a node, all required descendants of matching pattern nodes must have associated candidate nodes that are part of the same embeddings. If required descendants are missing, they cannot appear later on and the corresponding embeddings represent false positives.

Algorithm 6.3 Detect false positives that miss required nodes of a subfragment.

Input: Node n of document D; Pattern node n of index pattern P; Node n matches pattern node n, $\mathsf{n} \in matches(n)$, and is associated with a stack of the pattern node.

Description: Recursively determine whether each required child of pattern node n has associated a candidate node, which is a descendant of n. Initiate deletion of false positives.

Result: False positives that are descendants of node n and match a descendant of pattern node n have been detected and removed from the stack encoding.

```
procedure DETECTFALSEPOSITIVES($n$,$\mathsf{n}$)
    for all $\mathsf{n_c} \in children(\mathsf{n})$ do
        if $req(\mathsf{n_c}) = 1$ then
            if $n \notin \bigcup_{\mathbf{p} \in paths(\mathsf{n_c})} toppointer(stack(\mathsf{n_c}, \mathbf{p}))$ then
                DELETEFALSEPOSITIVES($n$, $\mathsf{n}$)
            end if
        end if
    end for
end procedure
```

Algorithm 6.3 shows the details of procedure DETECTFALSEPOSITIVES. The procedure takes as input a node n of the update fragment and one of its matching pattern nodes n of the affected index pattern. It recursively processes each child pattern node $\mathsf{n_c}$ of pattern node n. If the child pattern node is required, one of its stacks must have associated a descendant of n. This is true if one of its stacks contains a top node with an ancestor pointer to n. If this is not the case, the candidate nodes of the partial embedding represent false positives and are deleted.

Example 6.19 (detect false positives) *Considering Figure 6.8, Example 6.17 invokes procedure* DETECTFALSEPOSITIVES *four times. We will now look at step 9 in more detail, which invokes the procedure with node a_1 and pattern node A. Assume that both child pattern nodes D and F are required. Pattern node D has associated candidate node d_1, but pattern node F does not have any candidate node. Therefore, the partial embedding represents a false positive and is deleted by calling procedure* DELETEFALSEPOSITIVES *with node a_1 and pattern node A as input.*

Example 6.20 (detect resource without date) *Figure 6.10 depicts partial embeddings encoded into stacks when inserting fragment (b) of Figure 6.1. Node* 21–1.1.1 *matches pattern node* `resource`*. After processing the descendants of this node, the algorithm calls procedure* DETECTFALSEPOSITIVES *with the node and its pattern node as input. The child* `@date` *of pattern node* `resource` *is required. However, its stack does not contain a node that points to* 21–1.1.1. *Note that the date of the resource can only appear within the descendants of node* 21–1.1.1, *but not in a later step. Therefore node* 21–1.1.1 *represents a false*

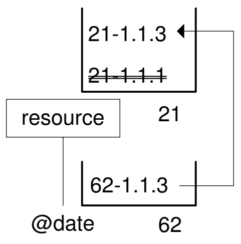


Figure 6.10: Detect and delete false positives when updating fragment (b) of Figure 6.1.

positive. When matching the second resource, node 21–1.1.3 *matches pattern node* `resource` *and node* 62–1.1.3 *matches pattern node* `@date`*. As this resource has a date, the nodes are not false positives and form a complete embedding.*

Delete false positives

Procedure deleteFalsePositives recursively removes a node and its descendants, which represent false positives, from their stacks to delete the corresponding partial embedding.

Algorithm 6.4 Delete false positives from partial embeddings.

Input: Node n of document D; Pattern node n of index pattern P; Node n matches pattern node n, $\mathsf{n} \in matches(n)$, and is associated with a stack of the pattern node.
Description: Recursively remove all descendants of n that match a child pattern node of n. Then remove node n from the stack of pattern node n.
Result: False positives that are descendants of node n and match a descendant of pattern node n have been removed from the stack encoding.

```
procedure deleteFalsePositives($n$,$\mathsf{n}$)
    for all $\mathsf{n_c} \in children(\mathsf{n})$ do
        for all $\mathbf{p} \in paths(\mathsf{n_c})$ do
            while $toppointer(stack(\mathsf{n_c}, \mathbf{p})) = n$ do
                deleteFalsePositives($topnode(stack(\mathsf{n_c}, \mathbf{p})), \mathsf{n_c}$)
            end while
        end for
    end for
    pop($n, stack(\mathsf{n}, pathschema(n))$)
end procedure
```

Algorithm 6.4 shows the details of procedure deleteFalsePositives. The procedure takes as input a node n of the update fragment and one of its matching pattern nodes n of the affected index pattern. Node n matches pattern node n and is associated with a stack of the pattern node. However, it is part of an embedding that represents a false positive and therefore has to be deleted. The algorithm recursively deletes descendant nodes that point to n bottom-up. For this purpose, it determines for each child pattern node $\mathsf{n_c}$ of pattern node n

whether one of its stacks has a corresponding descendant. In case that a child stack has nodes that point to n, it recursively calls procedure DELETEFALSE-POSITIVES to pop them from the stack. Procedure POP removes node n from the stack of the currently processed pattern node `n`, including its pointer to an ancestor in the parent stack.

Example 6.21 (delete false positives) *Considering Figure 6.8, Example 6.19 invokes procedure* DELETEFALSEPOSTIVES *with node a_1 and pattern node A as input (step 10). The child pattern node D has associated node d_1, which is a descendant of a_1. The algorithm therefore reinvokes the procedure with node d_1 and pattern node D as input. Pattern node D again has a child pattern node E. The candidate nodes associated with this pattern node, namely nodes e_1 and e_2 are descendants of d_1. Therefore, the algorithm reinvokes the procedure with node e_1 and pattern node E and node e_2 and pattern node E, respectively (steps 11 and 12). At the end of each recursion, it deletes the processed node from the stack encoding.*

Note that if node n and its descendants represent false positives, they are deleted. The corresponding partial embeddings may also contain ancestors of n or their descendants. These candidate nodes may also be part of other embeddings and are kept at first. As false positives are detected bottom-up, they will be deleted in a later step if necessary.

Example 6.22 (delete resource without date) *Consider Example 6.20 that detects that node* 21–1.1.1 *represents a false positive in Figure 6.10. Procedure* DELETEFALSEPOSITIVES *receives as input this node and its pattern node* `resource`*. It first reinvokes the procedure with child pattern node* `@date`*. As the stack associated with this pattern node has no descendant pointing to node* 21–1.1.1*, the procedure only has to delete node* 21–1.1.1 *from the stack encoding.*

6.4.2 Execute Queries

After the step of finding embeddings, the index pattern encodes candidate nodes into stacks. To retrieve nodes that are part of embeddings but not contained in the update fragment, the algorithm recursively processes the pattern nodes bottom-up to query missing ancestors. When adding an ancestor to a pattern node, the algorithm recursively traverses its children top-down to query missing descendants. The algorithm encodes queried nodes into the stacks of the processed index pattern in the same way as when finding embeddings. Thereby, it marks queried nodes, which is necessary when generating index entries (cf. Subsection 6.4.3). Note that the algorithm currently does not consider query optimization strategies when retrieving missing nodes.

The algorithm to execute queries uses three main procedures. Procedure QUERY traverses the index pattern top-down to find those pattern nodes whose candidate nodes miss ancestor pointers. This procedure then calls procedure MOVEUP to query missing ancestors, which itself calls procedure MOVEDOWN to query missing descendants. Figure 6.11 gives an abstract example to depict the interaction between the various procedures.

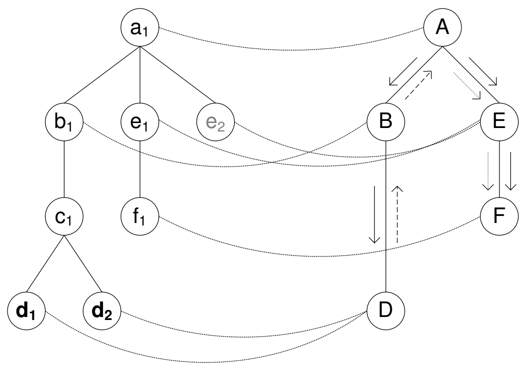


Figure 6.11: Query missing nodes (left) for an index pattern (right).

Example 6.23 (execute queries) *Figure 6.11 gives an abstract example of how to query missing nodes based on an index pattern. Assume that the update fragment contains nodes d_1 and d_2, which are candidate nodes for pattern node D. Procedure* QUERY *traverses the index pattern to determine the pattern nodes whose matching nodes miss ancestors (solid arrows). Procedure* MOVEUP *queries missing ancestors (dashed arrows) and procedure* MOVEDOWN *queries missing descendants (dotted arrows).*

Procedure MAINTAIN initiates the execution of queries by calling procedure EXECUTEQUERIES with the currently processed index pattern as input. Algorithm 6.5 provides the details of the procedure. Each pattern node has associated one or several stacks. Procedure EXECUTEQUERIES simply calls procedure QUERY with each stack associated with the root pattern node as input.

Algorithm 6.5 Initiate the execution of queries.

Input: Index pattern P with candidate nodes of the update fragment encoded into stacks.

Description: Call procedure QUERY with each stack of the root pattern node as input.

Result: All missing nodes that relate to candidate nodes encoded into index pattern P have been queried and encoded into the stacks of P.

```
1: procedure EXECUTEQUERIES(P)
2:     n = $root$(P)
3:     for all s $\in$ $stacks$(n) do
4:         QUERY(s)
5:     end for
6: end procedure
```

Each stack has associated a matching path schema. These path schemas can be used to generate queries to missing nodes (cf. Subsection 6.3.5). The stacks associated with an index pattern form stack hierarchies. As the procedures operate on these stack hierarchies to query missing nodes, we introduce some more functions:

- Function *nodes* : $\mathsf{S} \rightarrow 2^N$ returns all nodes associated with a stack.
- Function *path* : $\mathsf{S} \rightarrow \mathbf{P}$ associates each stack with its path schema: $\forall \mathsf{n} \in \mathsf{N}, \mathbf{p} \in paths(\mathsf{n}) : path(stack(\mathsf{n}, \mathbf{p})) = \mathbf{p}$.
- Function *stacks* : $\mathsf{N} \rightarrow 2^{\mathsf{S}}$ returns all stacks of a pattern node: $\forall \mathsf{n} \in \mathsf{N} : stacks(\mathsf{n}) = \bigcup_{\mathbf{p} \in paths(\mathsf{n})} stack(\mathsf{n}, \mathbf{p})$.
- Function *pnode* : $\mathsf{S} \rightarrow \mathsf{N}$ associates a stack with its pattern node: $\forall \mathsf{n} \in \mathsf{N}, \mathsf{s} \in stacks(\mathsf{n}) : pnode(\mathsf{s}) = \mathsf{n}$.
- Function *children* : $\mathsf{S} \rightarrow 2^{\mathsf{S}}$ returns the children of a stack: $children(\mathsf{s}) = \{\mathsf{s_c} \in \mathsf{S} \mid pstack(pnode(\mathsf{s_c}), path(\mathsf{s_c})) = \mathsf{s}\}$.
- Partial function *parent* : $\mathsf{S} \rightarrow \mathsf{S}$ returns the parent stack of a stack: $parent(\mathsf{s}) = \mathsf{s_p}$ iff $\mathsf{s} \in children(\mathsf{s_p})$.

In the following, we will give a concrete algorithm for each procedure and demonstrate its usage based on the abstract example of Figure 6.11 and on the running example of managing project resources.

Query index pattern

After finding embeddings of the index pattern in the update fragment, several embeddings may only be partial, i.e. miss nodes. Queries to missing nodes have to start from those candidate nodes that do not have ancestor pointers and are not in a stack of the root pattern node. Several stacks of an index pattern may have associated candidate nodes that do not have such ancestor pointers.

Algorithm 6.6 Traverse index pattern to locate stacks from which queries start.

Input: Stack s associated with index pattern P.
Description: Recursively traverse the children of stack s to find those stacks whose candidate nodes do not have ancestor pointers. Start queries from these candidate nodes.
Result: If stack s has associated candidate nodes, missing ancestors of these nodes and their descendants have been queried and encoded into stacks.

```
1: procedure QUERY($\mathsf{s}$)
2:     if $topnode(\mathsf{s}) = \perp$ then
3:         for all $\mathsf{s_c} \in children(\mathsf{s})$ do
4:             QUERY($\mathsf{s_c}$)
5:         end for
6:     else
7:         MOVEUP($\mathsf{s}$)
8:     end if
9: end procedure
```

Algorithm 6.6 shows the details of procedure QUERY. The procedure is first invoked with the stacks of the root pattern node by procedure EXECUTE-QUERIES. It then recursively traverses the stacks top-down as long as the stacks are empty. As soon as it has reached a non-empty stack, the nodes of this stack cannot have ancestor pointers because their parent stack was empty. The algorithm calls procedure moveUp to query missing nodes. In case that the update fragment did not contain any matching nodes, procedure query does not find a non-empty stack and therefore does not initiate queries.

Example 6.24 (query) *In Figure 6.11, the index pattern encodes nodes d_1 and d_2 after having executed step 1 of the maintenance algorithm. Assume that each pattern node has associated exactly one stack. Procedure* executeQuery *calls procedure* query *with the stack of the root pattern node A. It recursively traverses the stack structure to determine that the stack of pattern node D is not empty. Thus, it calls procedure* moveUp *with this stack as input.*

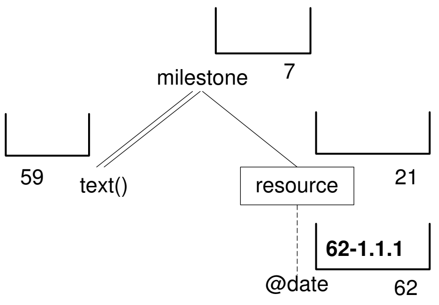


Figure 6.12: Initiate querying missing nodes when updating fragment (c) of Figure 6.1.

Example 6.25 (query from resource date) *Figure 6.12 depicts the stack encoding when updating fragment (c) of Figure 6.1. The first step of the maintenance algorithm matches node* 62–1.1.1 *with pattern node* `@date`*. All other nodes need to be queried. Procedure* query *traverses the stacks of the index pattern until reaching the stack associated with pattern node* `@date`*. As this stack is the upper-most non-empty stack, queries need to start from this stack.*

Query missing ancestors

If a stack is not empty, all nodes in the stack miss the same ancestor because the maintenance algorithm considers each update fragment individually. Procedure moveUp queries the missing ancestors of these nodes. Note that all descendants of these nodes have been contained in the update fragment and need not be retrieved from the document.

Algorithm 6.7 contains the details of procedure moveUp. The procedure receives as input a stack `s` whose candidate nodes miss ancestor pointers. In general, the parent stack is still empty (the opposite case will be discussed below) and the procedure queries the missing ancestor by calling retrieveAncestor. It then adds it to the parent stack and creates a pointer from each node in the stack to the ancestor. By calling procedure moveDown, it queries missing descendants of this ancestor. Finally, it recursively moves up to query further missing ancestors.

Algorithm 6.7 Move up the index pattern to query missing ancestors.

Input: Stack s associated with index pattern P whose nodes miss ancestor pointers.

Description: Query the ancestors of the nodes associated with s and push them onto the parent of stack s. Then initiate queries to missing descendants of these ancestors.

Result: All missing ancestors of candidate nodes associated with stack s and their descendants have been queried and encoded into stacks.

```
procedure moveUp(s)
    if $\exists parent(\mathsf{s}) \land (topnode(\mathsf{s}) \neq \perp)$ then
        if $topnode(parent(\mathsf{s})) \neq \perp$ then
            $n_a = topnode(parent(\mathsf{s}))$
        else
            $n_a =$ retrieveAncestor($topnode(\mathsf{s}), path(parent(\mathsf{s}))$)
            push($n_a, parent(\mathsf{s})$)
        end if
        for all $n \in nodes(\mathsf{s})$ do
            createPointer($n, \mathsf{s}, n_a, parent(\mathsf{s})$)
        end for
        moveDown($n_a, parent(\mathsf{s}), pnode(\mathsf{s})$)
        moveUp($parent(\mathsf{s})$)
    end if
end procedure
```

Procedure retrieveAncestor receives as input a context node, from which the query starts, and the path schema of the ancestor which it queries. Note that the algorithm does not make any assumption on how the database executes this procedure. It may either issue a query on the document or calculate its label with the help of a labeling scheme (cf. Chapter 3).

Example 6.26 (move up) *Considering Example 6.24 and Figure 6.11, procedure* query *calls procedure* moveUp *with the stack of pattern node D as input. This stack contains nodes d_1 and d_2. The procedure queries the missing ancestor belonging to pattern node B, which is node b_1. It pushes this node onto the stack of pattern node B and creates pointers from d_1 and d_2 to b_1. As pattern node B may have further children apart from D, the procedure calls procedure* moveDown*, which queries missing descendants of b_1. It then reinvokes procedure* moveUp *with the stack of pattern node B as input. In the second pass, the procedure queries the ancestor a_1 of node b_1 and pushes it onto the stack of pattern node A.*

Example 6.27 (query ancestors of date) *To continue with Example 6.25, procedure* moveUp *has to retrieve missing ancestors of node* 62–1.1.1 *as depicted in Figure 6.13. It first executes a query to retrieve the ancestor of node* 62–1.1.1 *with path schema* 21*. It adds this ancestor, which has label* 21–1.1.1*, to the parent stack associated with pattern node* `resource` *and adds a pointer from node* 62–1.1.1 *to the queried ancestor. The ancestor does not miss any descendants and thus the procedure is reinvoked to calculate the ancestor of node* 21–1.1.1*,*

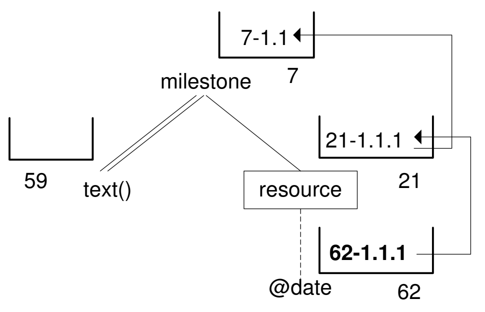


Figure 6.13: Query missing ancestors when updating fragment (c) of Figure 6.1.

which has label 7–1.1. *The algorithm finally calls procedure* MOVEDOWN *to query the missing descendants of node* 7–1.1.

A special case arises when procedure QUERY calls procedure MOVEUP more than one time, i.e. if sibling stacks contain candidate nodes that miss ancestor pointers. Again all candidate nodes miss the same ancestor, which is queried when invoking procedure MOVEUP the first time. In succeeding calls, the ancestor need not be queried again, but corresponds to the top-most node of the parent stack (cf. lines 3 and 4 in the algorithm).

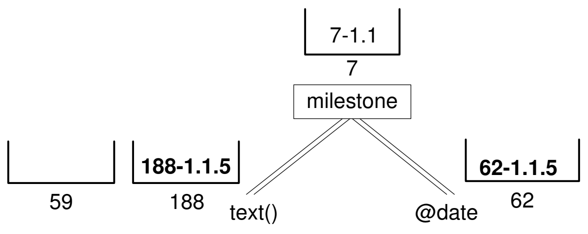


Figure 6.14: Stack encoding when ancestor queries start from several stacks.

Example 6.28 (queries from several stacks) *Consider the index pattern of Figure 6.14, which indexes all titles and dates of milestones without considering resources (ancestor pointers are omitted for better readability). Assume that we insert a new resource with label* 21–1.1.5 *that has a title with label* 188–1.1.5 *and a date with label* 62–1.1.5. *The first step of the maintenance algorithm finds a candidate node for the* `text()` *and the* `@date` *pattern nodes (depicted in bold). Queries start from each of these candidate nodes. Both candidate nodes have the same ancestor with label* 7–1.1, *which only needs to be queried when calling procedure* MOVEUP *the first time.*

Query missing descendants

After having queried a missing ancestor, procedure MOVEUP calls procedure MOVEDOWN to query missing descendants of this ancestor. Thereby, procedure MOVEDOWN has to query all descendants except for those descendants that have been the context nodes for the ancestor queries, i.e. which have been contained in the update fragment. After having queried missing descendants, the procedure has to call procedure DETECTFALSEPOSITIVES to determine if required nodes are missing in the document.

Algorithm 6.8 Move down the index pattern to query missing descendants.

Input: Context node n and stack s to which the node belongs; Pattern node $\mathsf{n_q}$, which is a child of the pattern node with which stack s is associated or null.
Description: Recursively query missing descendants of node n that need to be associated with children of s. Thereby, exclude children stacks that belong to pattern node $\mathsf{n_q}$. Push queried nodes onto the corresponding stacks, which are children of stack s. Check for false positives that need to be deleted from the stack encoding.
Result: All missing descendants of node n have been queried and encoded into descendant stacks of stack s.

```
 1: procedure MOVEDOWN($n$, $\mathsf{s}$, $\mathsf{n_q}$)
 2:     $\mathsf{S_c} = \{\mathsf{s_c} \in children(\mathsf{s}) \mid pnode(\mathsf{s_c}) \neq \mathsf{n_q}\}$
 3:     for all $\mathsf{s_c} \in \mathsf{S_c}$ do
 4:         $N_d$ = RETRIEVEDESCENDANTS($n, path(\mathsf{s_c})$)
 5:         $N_r = \{n_d \in N_d \mid n_d \notin nodes(\mathsf{s_c})\}$
 6:         for all $n_r \in N_r$ do
 7:             PUSH($n_r, \mathsf{s_c}$)
 8:             CREATEPOINTER($n_r, \mathsf{s_c}, n, \mathsf{s}$)
 9:             MOVEDOWN($n_r, \mathsf{s_c}, \perp$)
10:         end for
11:     end for
12:     DETECTFALSEPOSITIVES($n, pnode(\mathsf{s})$)
13: end procedure
```

Algorithm 6.8 provides the details of procedure MOVEDOWN. It receives as input a context node n and a stack s. Node n has been pushed to stack s by procedure MOVEUP. Further, the procedure receives as input a pattern node $\mathsf{n_q}$ which marks the pattern node from which procedure MOVEUP has started the ancestor query. Procedure MOVEDOWN first determines the set of stacks $\mathsf{S_c}$ whose candidate nodes need to be queried. This set consists of the children of stack s that do not belong to pattern node $\mathsf{n_q}$. The descendants that match pattern node $\mathsf{n_q}$ have already been contained in the update fragment. For each stack to be queried, procedure RETRIEVEDESCENDANTS executes a query to retrieve the descendants N_d of node n that have the path schema of the queried stack. In general, these descendants still have not been queried, i.e. they are not part of the child stack (the opposite case will be discussed below). In this case, the set of descendants N_r is equal to N_d. The algorithm then adds each descendant n_r to the child stack $\mathsf{s_c}$ and creates a pointer from the descendant n_r to node n to keep structural relationships. It then recursively queries further descendants. Finally, the procedure calls procedure DETECTFALSEPOSITIVES to detect and eliminate false positives.

For ease of presentation, we chose to look for false positives after having queried all descendants. In case that a child pattern node misses required nodes, the procedure executes queries on remaining children before detecting false positives. However, these queries would not be necessary as its nodes represent false positives anyway.

Example 6.29 (move down) *Considering Example 6.26 and Figure 6.11, procedure* moveUp *calls procedure* moveDown *with node a_1, the stack of pattern node A and pattern node B as input. The latter pattern node represents the pattern node from which the query to node a_1 started. Therefore, this pattern node does not miss further descendants. The procedure only has to query the descendants of node a_1 that belong to stack E. The corresponding query returns nodes e_1 and e_2, which are pushed onto the stack of pattern node E. As both nodes miss further descendants, procedure* moveDown *is reinvoked with each node and the stack of pattern node E. Note that this time the third parameter is null because all descendants of these nodes have to be queried. The query for the descendants of node e_1 returns node f_1, whereas node e_2 does not have any descendants. By calling procedure* detectFalsePositives *with node e_2 and pattern node E as input, the algorithm will detect that this node represents a false positive and will delete it from the stack.*

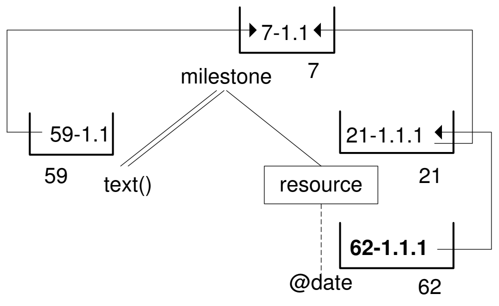


Figure 6.15: Query missing descendants when updating fragment (c) of Figure 6.1.

Example 6.30 (query missing descendants of milestone) *Reconsider Example 6.27 that calls procedure* moveDown *after having queried the missing ancestor with label* 7–1.1 *(cf. Figure 6.15). The procedure receives as input the queried ancestor label* 7–1.1*, its stack and pattern node* `resource`*. Procedure* moveDown *first determines the stacks which it needs to query. As the missing* `resource` *node has already been queried when moving up the index pattern, it only needs to query the missing milestone title, which has label* 59–1.1.

A special case arises when queries start from several stacks (cf. Example 6.28). In this case, procedure moveUp may call procedure moveDown several times with the same ancestor as input. In this case, it is still necessary to query missing descendants because the pattern node, which is excluded from querying, can be different in each invocation of the procedure. As the algorithm may therefore query the same descendants several times, it only adds those descendants to the stack encoding which still have not been queried (cf. line 5 of Algorithm 6.8). To avoid executing the same query multiple times, an alternative solution would be to keep track of the pattern nodes to which queries have already been executed.

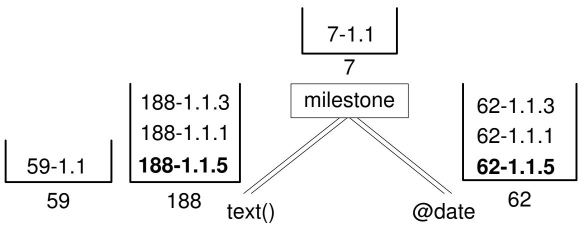


Figure 6.16: Stack encoding when several descendant queries start from the same ancestor.

Example 6.31 (queries from several stacks) *Considering Example 6.28, procedure* moveUp *has been invoked twice to add ancestor pointers from the candidate nodes of the update fragment to the queried ancestor* 7–1.1. *The new title forms an embedding with the date of every resource of the milestone. Similarly, the date relates to the milestone title and every resource title according to the index pattern. Figure 6.16 depicts the stack encoding when querying missing descendants. In this example, procedure* moveDown *is called twice by procedure* moveUp*. It first queries missing dates, which have labels* 62–1.1.1 *and* 62–1.1.3*. In the second call, it queries missing titles, which have labels* 59–1.1*,* 188–1.1.1 *and* 188–1.1.3*. Note that according to the index pattern, each combination of title and date forms an index entry (we do not discuss the meaningfulness of such a pattern here).*

Procedure moveDown can also be used to retrieve all embeddings from a document when creating or deleting an index on an existing document. In this case, it queries all embeddings from the document. Alternatively, it would also be possible to query the entire document and match it with the index pattern when creating an index on an existing document. However, in case of large documents, the latter approach will be very inefficient as each index usually only refers to some nodes of the document.

6.4.3 Generate Index Entries

To generate partial index entries, the maintenance algorithm calls procedure generateIndexEntries. This procedure receives as input an index pattern P which encodes embeddings into its stacks and the kind of update operation. It then extracts the embeddings from the stack encoding. As query pattern matching algorithms use similar procedures (cf. [200]), we do not provide details for this step. Based on the index pattern, the algorithm has to determine for each embedding which nodes form index keys and which node represents the return. Finally, it has to forward the partial index entries to the index structures. The index entries need to be inserted into the index structure in case of an insert operation, deleted in case of a delete operation and updated if the insert or delete operation affects optional keys.

Example 6.32 (generate index entries) *Figure 6.9 encodes two embeddings, resulting from inserting fragment (b) of Figure 6.1. The embedding (*7–1.1*,* 59–1.1*,* 21–1.1.1*,* $\perp$*) is transformed into the partial* *index entry* ((‘`design`’, ⊥) → (21–1.1.1)). *The embedding (*7–1.1, 59–1.1, 21–1.1.3, 62–1.1.3*) is transformed into the partial index entry* ((‘`design`’,‘`2008-01-08`’) → (21–1.1.3)). *When inserting fragment (c), the algorithm finds embedding (*7–1.1, 59–1.1, 21–1.1.1, 62–1.1.1*), as depicted in Figure 6.15. As the update affects an optional key, the corresponding partial index entry* ((‘`design`’, ⊥) → (21–1.1.1)) *is deleted and the new partial index entry* ((‘`design`’,‘`2007-03-12`’) → (21–1.1.1)) *is inserted into the corresponding index structure.*

In Subsection 6.4.2, we have mentioned that the execution algorithm has to mark queried nodes. This is necessary to easily detect optional index keys, which stem from nodes of the update fragment (cf. Subsection 6.3.6). However, there is another situation in which the distinction between queried nodes and nodes of the update fragment is essential. It can arise if candidate nodes of different stacks miss ancestor pointers before executing queries (cf. Examples 6.28, 6.31 and 6.33). In this case, the algorithm may generate index entries which only contain queried nodes. These index entries must not be updated because they are already part of the index. We refer to such index entries as unaffected index entries.

Example 6.33 (unaffected index entries) *Consider the index pattern of Figure 6.16, which depicts matching nodes of the update fragment in bold. Amongst others, the stacks encode the embedding with nodes* (7–1.1, 59–1.1, 62–1.1.1)*, which however only consists of queried nodes. Therefore the maintenance algorithm must not update this index entry. The update only affects those index entries which include at least one node of the update fragment.*

6.5 Evaluation and Extensions

To update all relevant index entries, the maintenance algorithm needs to find all affected embeddings and then generate partial index entries from these embeddings. This section shortly explains why the proposed algorithm finds all affected embeddings and then looks at possible extensions to index patterns.

The maintenance algorithm needs to find all affected index entries. This implies that it must not generate index entries that the index is not defined on, nor must it miss index entries. As the algorithm generates index entries from embeddings, it has to find all affected embeddings, which it guarantees as follows:

- *An embedding does not contain nodes that do not match the index pattern:* Each matching node has the same path schema as its matching pattern node and thus matches the index pattern. The stack encoding ensures that an embedding does not contain nodes that structurally do not belong together via ancestor pointers.

- *An embedding does not have too many nodes:* Each embedding contains at most one candidate node for each pattern node and thus cannot contain more nodes than required.

- *An embedding does not miss nodes:* The algorithm matches all nodes of the update fragment. If partial embeddings miss nodes, it executes source queries. Therefore, each embedding contains all nodes that it requires.
- *The algorithm does not find too many embeddings:* The algorithm does not find embeddings that are already contained in the index because each embedding contains at least one node of the update fragment. It does not produce embeddings that are not relevant by deleting false positives.
- *The algorithm does not miss embeddings:* The algorithm generates an embedding for each matching node of the update fragment and therefore cannot miss embeddings.

Several parts of the algorithm depend on current restrictions of index patterns (cf. Section 5.3). Possible extensions to index patterns include wildcards, required pattern nodes beneath optional pattern nodes or value variables on element nodes. Supporting such extensions also affects the maintenance algorithm as we shortly outline in the following.

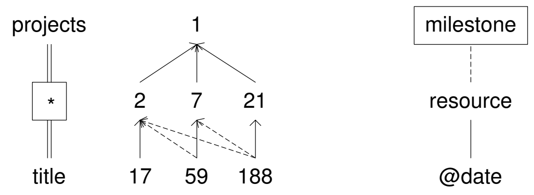


Figure 6.17: Index pattern with wildcard (left) and with required pattern nodes beneath optional pattern nodes (right).

Wildcards entail that one matching node may have several ancestors in the parent pattern node. A similar situation arises when allowing recursive schemas.

Example 6.34 (wildcards) *The left index pattern of Figure 6.17 contains a wildcard that matches pattern nodes `project`, `milestone` and `resource`. Each title does not only form an embedding with its parent, but also with any ancestor matching the wildcard. Assume that we insert a new resource with a title. The insertion affects three embeddings because the title forms an embedding with the resource node as well as with the milestone and project nodes, which are its ancestors.*

In case that an optional pattern node may have a required pattern node as descendant, an insertion of a fragment may trigger a deletion of index entries and vice versa.

Example 6.35 (required and optional pattern nodes) *The right index pattern of Figure 6.17 defines a required pattern node beneath an optional pattern node. It indexes all milestones that either have no resource or that have at least one resource with a date. The insertion of a single milestone node, which does not have any resource, triggers the insertion of a new index entry. Assume that we then add a resource without a title to this milestone. According to the index pattern, if the milestone has a resource, the resource must have a date. Therefore, the milestone may no longer be indexed. Thus, the insertion of a new node triggers the deletion of an index entry in this case.*

Index patterns currently only support value variables on text and attribute nodes. A value variable on an element node expresses that the index key consists of the element content, i.e. the concatenation of its text node descendants. In this case, an update of any of these text nodes affects the index although the corresponding path schema is not directly indexed. Further, the index key is composed of the values of several nodes. To insert/delete single values of this composed index key, the entire index key needs to be determined. A similar situation arises when indexing the text nodes of an element that allows mixed content.

Example 6.36 (content variables) *Assume that we define a value variable on resource nodes, which corresponds to a variable on the element content of resources. The index key of each resource node is composed of its title and description. Updating the title affects the content variable associated with the resource node. Maintaining the index requires the corresponding description to determine the entire index key and perform the update operation.*

6.6 Summary

To keep indices consistent with updates on documents, a database requires an index maintenance algorithm. The algorithm presented in this chapter extracts index entries from update fragments based on index patterns. By matching update fragments against index patterns, the algorithm can update arbitrary index structures defined on arbitrary document fragments.

The algorithm consists of three steps: (1) find embeddings of index patterns in update fragments, (2) execute queries if nodes are required for maintenance that are not part of the update fragment, (3) generate index entries from the embeddings and forward them to the corresponding index structures.

To determine whether an update node matches an index pattern, the algorithm matches the index pattern against the schema and associates each pattern node with affected path schemas. A node of the update fragment is affected by an index if its path schema is part of the index pattern. The algorithm traverses the update fragment, matches the nodes against the index pattern and encodes matching nodes into stacks. By keeping structural relationships between matching nodes via pointers, the stacks encode embeddings of index patterns in update fragments. In case that the update fragment misses nodes that are part of affected embeddings, the algorithm executes source queries. If it detects that an embedding is incomplete because not all required nodes exist in the document, it deletes it from the stacks. Finally, the stacks encode all affected embeddings and the algorithm generates index entries from the embeddings.

The main advantage compared to existing approaches is that the algorithm exploits the structure of the update fragment instead of processing each update node individually. It need not query relevant nodes that are contained in the update fragment. As updates in XML usually refer to fragments instead of to single nodes, the proposed algorithm minimizes the number of source queries, resulting in an improved update performance. In case that the update fragment contains all relevant nodes, the algorithm is completely self-maintainable. The update of simple patterns that only consist of one pattern node is always self-maintainable as well. With the help of labeling schemes, it is possible to update path patterns without querying the document by calculating the labels of ancestors. Integrating schema information allows the algorithm for efficiently determining whether a node is part of an index and affected by an update.

Part III

Evaluation

Chapter 7

Case Study: The XML Database SemCrypt

Contents

This chapter presents the main concepts of SemCrypt in form of a case study. SemCrypt is a native, secure XML database in whose context the concepts of this thesis have been developed and implemented. Section 7.1 gives a general introduction to SemCrypt and Section 7.2 describes its architecture. The concepts of SemCrypt are explained in Section 7.3 based on the running example of managing project resources. Section 7.4 further exemplifies how to process queries with indices in SemCrypt. Finally, Section 7.5 summarizes the main ideas of SemCrypt.

7.1 Introduction

Outsourcing IT services to external service providers is an emerging and growing market which represents a popular alternative to maintaining services in-house. By specializing on particular services, service providers can increase the quality and decrease the costs of their services. An important IT service is providing and administrating a data store. In this "database as a service" model, individuals or companies store documents at an external server and then query and update these documents without having to worry about, e.g., IT infrastructure, availability of data stores or back-ups. When storing sensitive data at an external data store, the so-called storage provider needs to ensure that neither intruders nor its own staff can access the data.

Service level agreements are insufficient to guarantee secure storage (cf. e.g. [197]). Instead, technical solutions are required to provide data owners with the necessary confidence that confidentiality and privacy of their data are maintained as well as to unburden storage providers from costly security measures. Secure processing models typically encrypt data to guarantee data confidentiality. Efficiently querying and updating encrypted data is yet considered a significant open problem [99]. Previous solutions on querying encrypted data either partition data [92, 108, 193], which raises security concerns [79], delegate substantial query processing work to the client [34, 35, 61], or rely on proprietary encryption techniques [41, 182] that cannot be replaced when they are broken or better ones are available.

In contrast to existing approaches on querying outsourced, encrypted data, the native XML database SemCrypt [176] neither relies on partitioning nor on specific encryption techniques. Instead, it uses special storage and index structures. The storage structures guarantee privacy of encrypted data and prevent statistical attacks. The index structures reduce processing overhead on the client by enabling the processing of queries and updates with only transferring a minimum amount of data from the storage provider to the client. The developed processing model grounds on a client-server architecture, where clients process data and servers act as storage providers in an untrustworthy domain. At the external storage provider, SemCrypt exclusively processes encrypted documents. Documents or document fragments are only decrypted at the client-side for authorized users.

As more and more data is semi-structured and consists of both structured and unstructured parts, SemCrypt uses XML as data format. The developed techniques to process queries and updates are based on a schema-aware labeling scheme and utilize the schemas of XML documents and combine them with special index structures. These techniques enable the direct access to encrypted data without having to perform a time-consuming decryption of the whole data store. Schemas, labeling and index structures, as described in this thesis, are thus an integral part of SemCrypt. Although SemCrypt has been designed as an encrypted database, the developed concepts can equally be applied to unencrypted XML databases.

7.2 Architecture

The architecture of SemCrypt consists of two major components, the server, which acts as the *Storage Provider*, and the SemCrypt *Core*, which incorporates specific techniques to process queries and updates.

The *Storage Provider* is responsible for storing, retrieving and updating document fragments as well as for backup, recovery and basic transaction control (cf. [71]). The SemCrypt *Core*, depicted in Figure 7.1, integrates the various components and their data models and bridges the gap between the physical storage of encrypted data and its representation to end users.

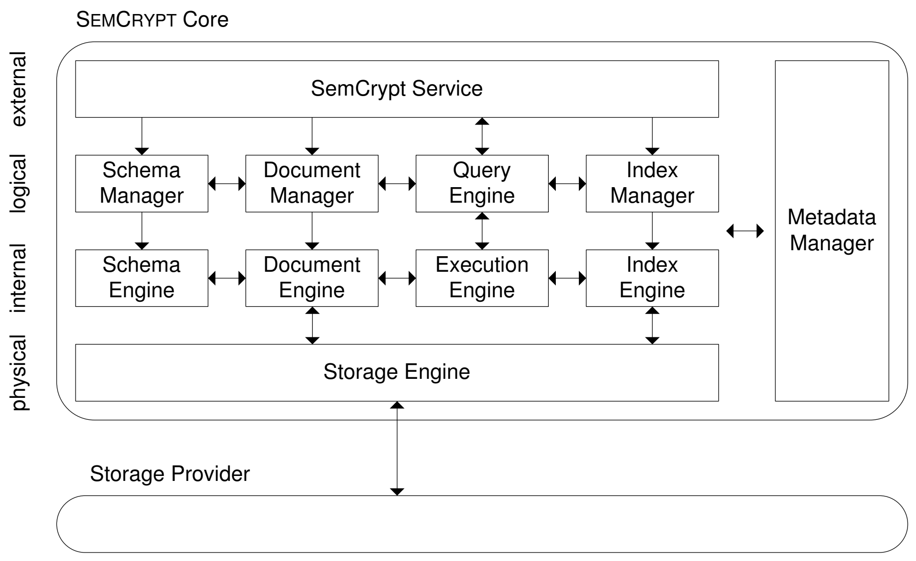


Figure 7.1: Architecture of the SemCrypt Core.

SemCrypt processes four kinds of primary data, namely *schemas*, *documents*, *indices* and *queries*. Following database reference architectures [98, 140], the processing of primary data occurs within different layers. The *external layer* represents data in a way familiar to end users, who e.g. formulate queries in terms of XPath. The *SemCrypt Service* takes user requests and data in their external representation, parses and dispatches them. Various database tasks, such as query optimization, cannot be efficiently performed on external primary data. The *logical layer* abstracts from external representations, while the *internal layer* incorporates specific techniques to improve system performance, such as labeling and indices. Within the logical and the internal layer, there is one component for each kind of primary data. At the *physical layer*, the *Storage Engine* provides access to the physical storage structures and is responsible for data encryption/decryption. All components create metadata, like names of available documents or available indices, and provide this metadata to other components. To manage metadata, SemCrypt follows the approach of using a separate (XML) database, encapsulated by the *Metadata Manager*.

The layered structure of this architecture allows for adapting SemCrypt to diverse application scenarios, all of which rely on a potentially untrustworthy Storage Provider to store encrypted data. Dependent on the scenario, the SemCrypt Core can run on a secure or an insecure client. The project focused on single-user scenarios, but also investigated extensions to multi-user scenarios. Further, it is possible to use SemCrypt as a native, unencrypted XML database by transferring the SemCrypt Core to the Storage Provider. Thus, the developed techniques enable efficient processing of encrypted or unencrypted XML documents.

In the *single-user scenario*, each one of several single users has full access to exactly one encrypted store, where one Storage Provider may host several stores. Typical applications include storing e-mails as searchable, encrypted XML documents at the Storage Provider, or providing an encrypted, searchable private document store where users manage their documents using a special purpose application. This application utilizes the SemCrypt Core for performing query and update processing as well as encryption/decryption within the trusted domain of the client.

The *multi-user scenario* enables several users to access the same documents from different, potentially insecure clients. Access authorization and encryption/decryption is delegated to a trusted security component which encrypts/decrypts document fragments for authorized users. Alternatively, this security check could also be performed by specific secure devices, such as powerful smart cards. Typical application scenarios are to outsource the storage of records in e-government applications, of health records or of financial reports.

7.3 Concepts

Designing a secure XML database system poses several challenges. To ensure data privacy and prevent security risks [62, 79, 138], the system must guarantee both *storage and communication security.* The physical storage structure must neither reveal the document content nor the document structure to the Storage Provider. To be widely applicable, the database should not depend on specific encryption techniques. Repeated encryption of the same plain text fragments or markups needs to result in different cipher texts in order to prevent statistic-based attacks. Regarding the client-server communication, repeated transmission of the same data has to be avoided, again in order to prevent statistic-based attacks. The database needs to support *querying and updating* of both the content and structure of documents, i.e. navigating within documents and constraining values and types. Finally, regarding the system's overall *performance*, the data volume to be transferred from the Storage Provider for query processing has to be minimized.

SemCrypt tackles these challenges as follows. The *physical storage structure* guarantees data privacy and security through encryption. To identify encrypted fragments, query and update processing techniques exploit the schema of XML documents, which is captured by the document structure. Processing *documents* is based on a schema-aware labeling scheme which not only enables SemCrypt to identify each node of a document by a unique node label, but also to execute many operations directly onto node labels without accessing encrypted data. To efficiently process queries, SemCrypt provides extensible *indexing* on the content and structure of arbitrary document fragments. *Query processing* is based on a query algebra enabling query optimization and index selection.

Prior works on XML processing present isolated, incompatible solutions for individual subproblems. Current XML databases widely ignore query optimization and cannot easily be extended to integrate schemas and flexible, selective indices. SemCrypt therefore also extends the state-of-the-art of unencrypted XML processing by integrating schemas, labeling and extensible indexing into a database system.

The following subsections give an overview of developed techniques. They provide various examples to manage project resources, taking the sample schema, document, queries and indices that have been used as running examples throughout the thesis. The examples are graphically represented by screen shots taken from the SemCrypt prototype as far as graphical representations are available. Subsection 7.3.1 starts with the storage structure. Schemas are investigated in Subsection 7.3.2 and documents in Subsection 7.3.3. Subsection 7.3.4 and Subsection 7.3.5 look at processing queries and indices, respectively.

7.3.1 Storage Structure

To guarantee data confidentiality and privacy, the *Storage Engine* encrypts each data fragment with a user defined standard encryption algorithm (E) and encryption key (K), whereby arbitrary encryption techniques can be used. The use of nonrecurring random numbers (nonce) prevents statistical analysis by encrypting same fragments differently in each communication with the Storage Provider. To identify encrypted fragments, each fragment has assigned a unique id, which encodes structural information about the fragment and is encrypted using a cryptographic hash function (H). The cryptographic hash function disperses fragments in the encrypted store, which further complicates statistical analysis. In order to prevent frequency analysis, the Storage Engine can cache frequently requested fragments.

id	fragment
1–18<12–1:1:1:	‘`XML is becoming increasingly important...`’

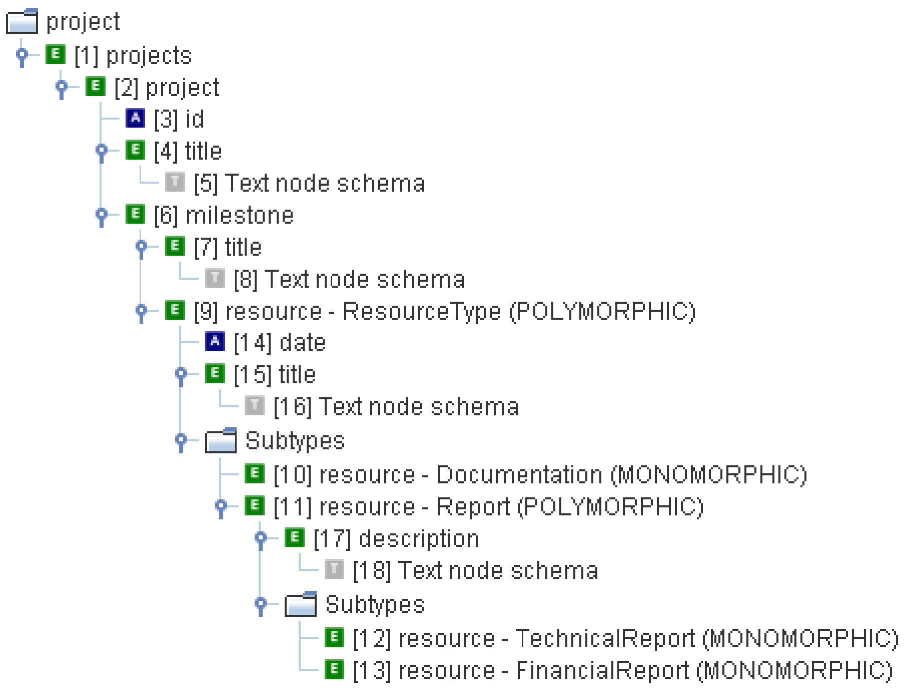
H(id)	E(fragment, K)	nonce
6253115564881552	$kdfu983fadifow03kdkaksdfhdl3i4r3ve...$	12481219

Table 7.1: Unencrypted and encrypted view on the storage structure.

Table 7.1 depicts a sample unencrypted data fragment and its encrypted counterpart. When the SemCrypt Core requires a fragment, it assembles its unique id based on metadata or partial query results and requests the fragment from the Storage Provider.

7.3.2 Schema Processing

SemCrypt uses the schema of XML documents to identify encrypted document fragments and optimize processing queries and updates. Each document stored in SemCrypt must have associated an XML Schema [187]. The *Schema Manager* represents each XML Schema in the logical schema model (cf. Subsection 2.1.2). Internally, the schema is represented as a schema tree to which the *Schema Engine* assigns schema labels (cf. Subsection 3.3.1, [88]). Logical and internal schema models represent metadata and are thus stored and retrieved by the Metadata Manager.



```
project
  E [1] projects
    E [2] project
      A [3] id
      E [4] title
        T [5] Text node schema
      E [6] milestone
        E [7] title
          T [8] Text node schema
        E [9] resource - ResourceType (POLYMORPHIC)
          A [14] date
          E [15] title
            T [16] Text node schema
          Subtypes
            E [10] resource - Documentation (MONOMORPHIC)
            E [11] resource - Report (POLYMORPHIC)
              E [17] description
                T [18] Text node schema
              Subtypes
                E [12] resource - TechnicalReport (MONOMORPHIC)
                E [13] resource - FinancialReport (MONOMORPHIC)
```

Figure 7.2: Labeled schema tree.

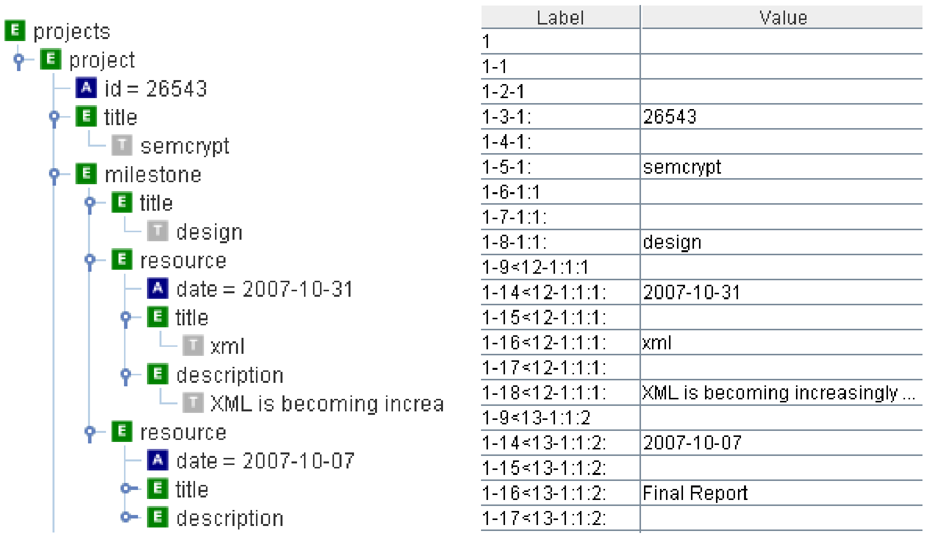
Example 7.1 (schema) *Figure 7.2 depicts the schema of the running example in form of a labeled schema tree. For example, path schema* `/projects/project/title` *has label* 4*. Note that the assignment of labels differs from the labeling proposed in this thesis.* SemCrypt *currently assigns labels in ascending order, which entails that access to the schema is necessary to compare structural relationships between labels. Further,* SemCrypt *currently assigns type labels dependent on path schema labels. For example, type* `<Documentation>` *has label* 10 *within path schema* 9*. The term 'polymorphic' denotes that a type has subtypes, whereas a 'monomorphic' type does not have subtypes.*

Schema labels are part of the internal representations of documents, queries and indices. More precisely, they are used within node labels of documents, within internal query plans and within the internal representation of indices as index patterns. They are exploited for optimizing and processing queries and for maintaining indices. Further, they are used as index keys in index structures.

7.3.3 Document Processing

Processing documents is based on a schema-aware, dynamic prefix labeling scheme (cf. Chapter 3, [88]). Node labels serve as identifiers for data fragments and are used to retrieve those encrypted fragments that are needed for query processing. They further support query processing by enabling to determine structural relationships between nodes. Within index structures, they are used as index keys as well as to represent the nodes returned by indices.

The *Document Manager* logically represents each document as a tree of nodes (cf. Subsection 2.1.1). The *Document Engine* internally assigns node labels to this document tree (cf. Subsection 3.3.2) and provides methods to evaluate structural relationships between labels and to support navigation via labels. The Document Manager identifies each document by a unique document name. The Document Engine assigns a unique id to each document, which is added to node labels to identify the document to which nodes belong. The document name and its id represent metadata and are stored by the Metadata Manager.



```
E projects
  E project
    A id = 26543
    E title
      T semcrypt
    E milestone
      E title
        T design
      E resource
        A date = 2007-10-31
        E title
          T xml
        E description
          T XML is becoming increa
      E resource
        A date = 2007-10-07
        E title
        E description
```

Label	Value
1	
1-1	
1-2-1	
1-3-1:	26543
1-4-1:	
1-5-1:	semcrypt
1-6-1:1	
1-7-1:1:	
1-8-1:1:	design
1-9<12-1:1:1	
1-14<12-1:1:1:	2007-10-31
1-15<12-1:1:1:	
1-16<12-1:1:1:	xml
1-17<12-1:1:1:	
1-18<12-1:1:1:	XML is becoming increasingly ...
1-9<13-1:1:2	
1-14<13-1:1:2:	2007-10-07
1-15<13-1:1:2:	
1-16<13-1:1:2:	Final Report
1-17<13-1:1:2:	

Figure 7.3: Document as a tree of nodes (left) and as a list of labels (right).

Example 7.2 (document) *The screen shots of Figure 7.3, which are taken from the* SemCrypt *prototype, depict a sample document as a tree of nodes and a list of labels. The document is instance of the schema of Figure 7.2. Each label identifies one node of the tree. Label* 1–9 <12–1:1:1*, for example, identifies the first resource. Thereby, the first number* (1) *identifies the document, the second* (9) *the labelpath* `/projects/project/milestone/resource` *and the third* (12) *the type* `<TechnicalReport>`*. The labelpath and type labels are taken from the labeled schema tree of Figure 7.2. The instance label* (1:1:1) *expresses that the node belongs to the first project and the first milestone of this project and that it is the first resource within this milestone. In the figure, an instance label terminates with a double point if the node cannot have siblings with the same name according to the schema.*

The Document Engine is responsible for storing, updating and retrieving the content and structure of documents via the Storage Engine. For this purpose, it uses two data structures as primary index structure, namely a hash table for the document content and a B+-tree for the document structure.

The hash table[1] associates each node label with its node value. Storing each node individually may result in large communication overhead when retrieving document fragments. SemCrypt therefore supports storing document fragments, whereby the fragmentation can be configured within the schema to better reflect query processing needs. When storing a document fragment, the hash table maps a node label of an element node to its descendant text and attribute nodes that are represented as a list of label-value pairs.

8-1:1:	design	
9-1:1:1	14-1:1:1:	2007-10-31
	16-1:1:1:	xml
	18-1:1:1:	XML is beco...
...	...	

Figure 7.4: Hash table on document content.

Example 7.3 (hash table on document content) *Figure 7.4 depicts a hash table on part of the content of the sample document of Figure 7.3. While it stores the title of milestone nodes individually, it represents each resource as a document fragment. The node label of the resource node* (9–1:1:1) *therefore maps to a list of label-value pairs, representing the text and attribute node descendants of the resource node. When outputting a resource, one access to the hash table is sufficient to retrieve the values of its descendants.*

To keep relationships between document fragments and evaluate queries on the document structure, SemCrypt uses a B+-tree on node labels. By indexing node labels in document order, the B+-tree can navigate through the document structure, e.g. to retrieve all descendants or children of a node (cf. [49]). Note that the Document Engine performs navigation with the help of labels as far as possible, e.g. to navigate to parents and ancestors. The B+-tree uses the same storage structure as the hash table by assigning a unique id to each page, representing pointers between pages with the help of these ids and storing each page as a fragment.

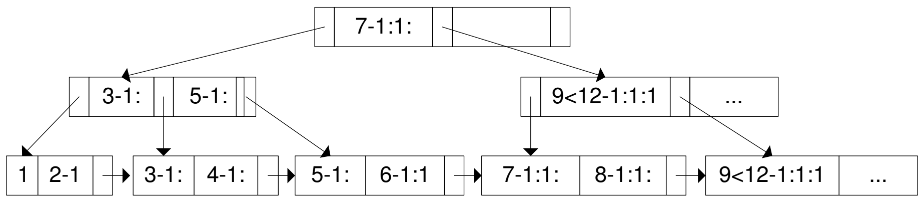


Figure 7.5: B+-tree on document structure.

[1]Alternatively, it would of course be possible to use e.g. a B+-tree.

Example 7.4 (B+-tree on document structure) *Figure 7.5 depicts a B+-tree on part of the document structure of the sample document (cf. Figure 7.3). To retrieve all descendants of resource* 9–1:1:1*, the retrieval algorithm of the B+-tree locates the data page of the resource node and traverses succeeding data pages as long as the labels represent descendants of the requested resource.*

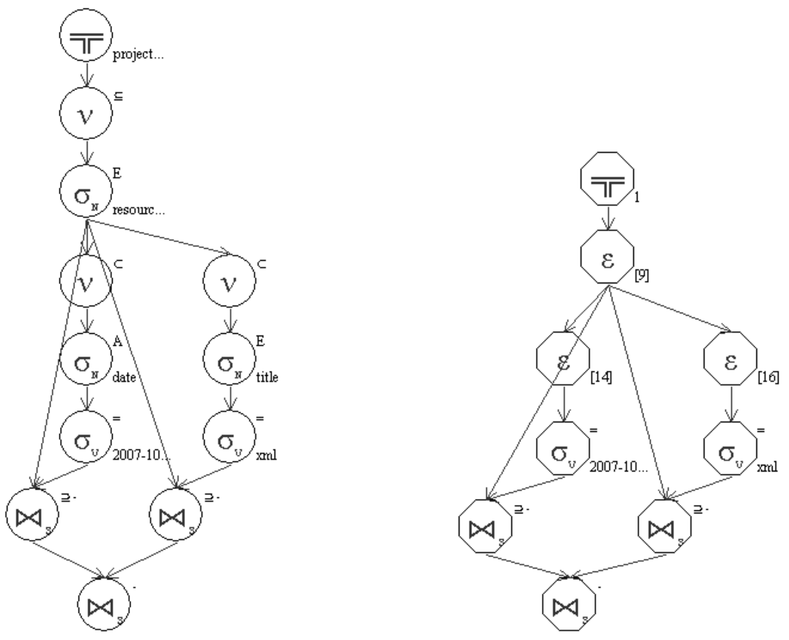
7.3.4 Query Processing

The performance of SemCrypt largely depends on query optimization since the most efficient way to process a query requires fewer data transferred and less communication with the Storage Provider. SemCrypt supports a restricted set of XPath 2.0, which basically corresponds to the subset presented in Section 2.2, but allows some more functions (e.g. count, position, matches).

Query processing occurs on two layers - the logical layer optimizes queries based on a logical query algebra, whereas the internal layer represents queries as execution plans and executes them. The *Query Engine* performs query optimization based on a logical query algebra. Although several query algebras have been proposed for XML (e.g. [37, 77, 80, 107, 145, 206]), there still does not exist a standard XML algebra. SemCrypt uses a proprietary algebra, which represents each query as a graph of logical operators. The algebra adapts existing algebras to support types and type hierarchies and basic function calls. We omit details of the query algebra as it is out of scope of this thesis.

While the logical algebra does not specify how to process queries, the internal algebra determines the necessary steps to execute queries. As the execution of queries is guided by the use of labels and indices, SemCrypt uses a proprietary internal algebra. The *Execution Engine* evaluates query plans compiled into this internal query algebra. It communicates with the Document Engine and the Index Engine to access and update documents and indices, respectively.

Example 7.5 *Assume that we want to retrieve resources of the sample document '`projects.xml`' by their date and title. Figure 7.6 depicts screen shots of the logical and internal query plans when executing the query `evaluate //resource [@date = '2007-10-31'] [title = 'xml'] over projects.xml` in* SemCrypt*. In the logical query plan, the first operator specifies the document to be queried. The second and third operators determine to navigate to all descendants that are elements and have the name `resource`. Then the query plan specifies two predicates, which are represented as branches in the query plan. The left predicate navigates from resulting resources to their dates and selects the dates with the requested value. The right predicate navigates to titles to select titles according to their value. Each predicate terminates with a structural join between resource nodes and the nodes returned by the predicate. It expresses to retain only those resources that fulfill the predicate. The final operator is again a structural join to only keep resources that fulfill both predicates. The internal query plan is similar except for that it represents documents by their internal id and navigational steps via schema labels. For example, the navigation to resources is represented by schema label* 9*. Note that* SemCrypt *currently does not rewrite query plans. For example, it may be more efficient to first determine the resources which have the requested date and then only compare the titles of these resources. Such query optimization techniques are out of scope of this thesis as they are orthogonal to index processing.*



	description	superscript	subscript
╥	document operator		document name
ν	axis navigation	axis	
σ_N	node selection	node kind	node name
σ_V	value selection	comparison operator	node value
$\bowtie_S$	structural join	join axis	
ε	axis navigation and node selection		labels

Figure 7.6: Query on the date and title of resources in the logical (left) and internal (right) query algebra.

7.3.5 Index Processing

SemCrypt provides flexible and selective indexing by integrating index structures (cf. Chapter 4), the index framework (cf. Chapter 5) and the index maintenance algorithm (cf. Chapter 6) as presented in this thesis. The master thesis [125] also provides details on index structures and the index framework in SemCrypt and their implementation.

To define indices, SemCrypt uses a proprietary index definition language, which extends XPath by index variables. The *Index Manager* represents index definitions as logical query plans, whereas the *Index Engine* internally uses index patterns. These patterns represent metadata and are stored by the Metadata Manager. The Index Engine is responsible for maintaining and accessing index structures. These index structures communicate with the Storage Engine to store and retrieve index pages, using the same storage structure as documents.

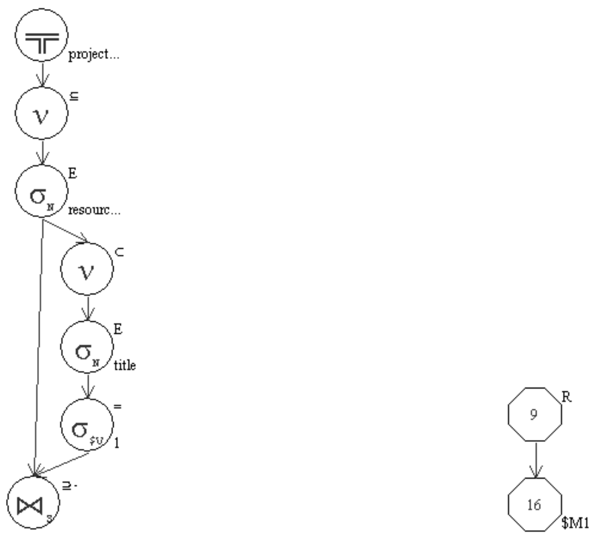


Figure 7.7: Value index on resource titles in the logical algebra (left) and as an index pattern (right).

Example 7.6 *In* SemCrypt, *a value index on resource titles can be defined by* `create index titleIndex on projects.xml for /descendant::resource [child::title = $V1]`. *Operator* = *denotes that the index only supports exact comparisons.* $\$V1$ *represents an index variable on values with id* 1. *The left side of Figure 7.7 depicts a screen shot of the logical index. It corresponds to a logical query which uses an index variable instead of the value selection. The right side of Figure 7.7 represents the index as a tree pattern that uses schema labels taken from the labeled schema tree of Figure 7.2. The index pattern consists of two pattern nodes. The root pattern node denotes that the index returns (R) nodes with schema label* 9. *Its child pattern node defines an index variable ($M1) on nodes with label* 16. *Thereby, the letter M stands for indexing the value for exact comparisons. Based on this index variable,* SemCrypt *implicitly configures the index to physically use a hash table.*

To select indices for queries, the index framework extends the Query Engine. Currently, SemCrypt uses a proprietary index selection algorithm which compares query plans with index patterns. If an index pattern matches part of a query, it replaces the corresponding operator chain with an index access operator. The current selection algorithm does not find all possible indices for a query. It cannot rewrite the query plan, but only replaces operator chains by index access operators. Alternatively, it would be possible to use the index selection algorithm of Arion et al. [16], which is more powerful. To access selected indices, the internal query algebra of the Execution Engine defines a corresponding index access operator. When updating documents, the Execution Engine forwards update fragments to the Index Engine for maintenance.

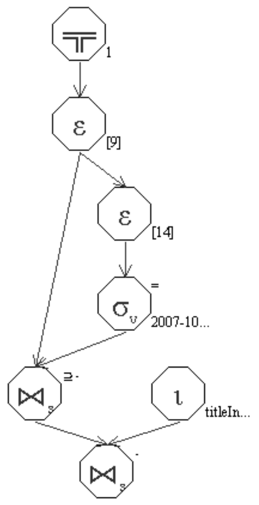


Figure 7.8: Querying the date and title of resources with an index on resource titles.

Example 7.7 (index selection) *Figure 7.8 depicts the internal query plan when executing the query of Figure 7.6 with the index of Figure 7.7. Instead of navigating to resource titles and selecting the resources with the requested title, the query plan specifies to access the index on resource titles. Note that the query plan could be optimized by specifying to first access the index and then only navigate to the date of resources returned by the index.*

7.4 Indices in Action

This section demonstrates how indices simplify the execution of queries by comparing query plans without and with indices.

Table 7.2 defines sample queries and indices using the SemCrypt syntax. The queries correspond to queries Q7, Q8, Q13 and Q10 of Table 2.2. The indices correspond to indices I2, I6, I15 and I9 of Table 4.4. The only exception is that I1 only refers to resource titles instead of to all titles because SemCrypt currently does not support wildcards before predicates. Implicitly, SemCrypt

Q1	evaluate /descendant::resource [attribute::date = ‘2007-10-31’] [child::title = ‘xml’] over projects.xml
Q2	evaluate /descendant::resource[attribute::date $\geq$ ‘2007-01-01’ and attribute::date $\leq$ ‘2008-01-01’] over projects.xml
Q3	evaluate /descendant::element(resource, TechnicalReport) [child::description/fn:matches(self::*, ‘\bdata.*’)] over projects.xml
Q4	evaluate /descendant::project [attribute::id = ‘26543’]/ descendant::milestone [child::title = ‘design’]/ descendant::resource [attribute::date $\geq$ ‘2007-01-01’] over projects.xml
I1	create index titleIndex on projects.xml for /descendant::resource [child::title = $V1]
I2	create index dateIndex on projects.xml for /descendant::resource [attribute::date >< $V1]
I3	create index descriptionTypeIndex on projects.xml for /descendant::element(resource, $V1) [child::description/ fn:matches(self::*, $V2)]
I4	create index milestoneDateIndex on projects.xml for /descendant::milestone [sc:label(self::*) = $V1] / descendant::resource [attribute::date >< $V2] nest by $V1,$V2

Table 7.2: Queries and indices.

generates a hash table for I1 and a B+-tree for I2. I3 contains two index variables to support hierarchical comparisons on types and word and word prefix queries on description values. As the extensible KDB-tree of SemCrypt can support both variables, SemCrypt physically uses this index structure for I3. I4 also contains two variables. The first variable indexes the path of milestones for hierarchical comparisons (this is indicated by function ‘sc:label’). The second variable defines to support range comparisons on date values. The index definition includes the expression ‘nest by’. It indicates to nest the index on dates beneath the index on milestone paths. SemCrypt therefore generates a B+-tree on milestone paths and nests a B+-tree on date values beneath this index structure.

Figure 7.9 depicts the internal query plans of queries Q1, Q2 and Q3 of Table 7.2 when no indices are available.

When defining indices I1, I2 and I3 of Table 7.2, SemCrypt generates the internal query plans depicted in Figure 7.10 for queries Q1, Q2 and Q3, respectively. Q1 is executed by accessing the index on titles and the index on dates and then joining index results. Q2 and Q3 simply require accessing the index on resource dates and on the type and description of resources, respectively.

Figure 7.11 demonstrates accessing a path index based on partial query results. The internal query plan on the left executes Q4 without accessing an index. In case that I2 is available, the query plan in the middle specifies to query the requested milestones, access I2 to retrieve resources with the requested date and the join resources and milestones. Note that I2 indexes all resources. Although the query first selects milestones, it needs to traverse the entire index on resource dates. I4 groups resource dates according to their milestone. The query plan on the right accesses this index with the requested milestones as

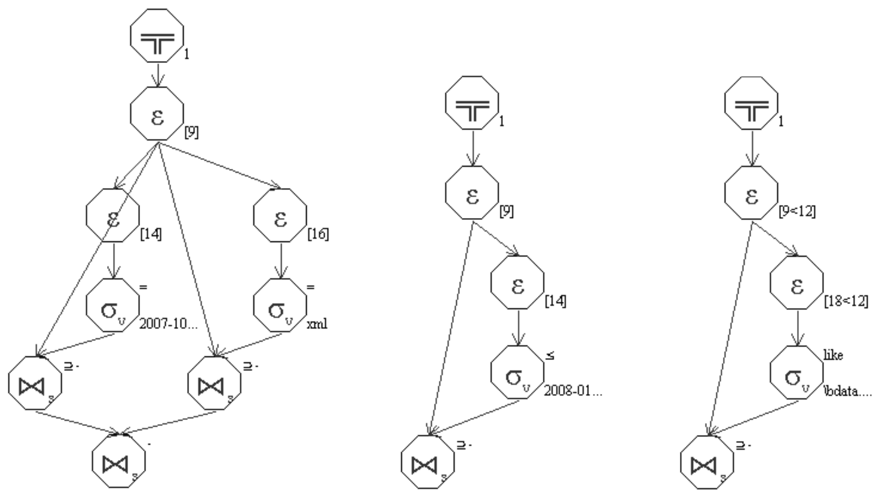


Figure 7.9: Queries Q1, Q2 and Q3 of Table 7.2 without indices.

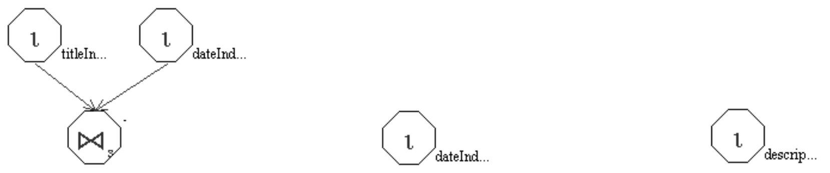


Figure 7.10: Queries Q1, Q2 and Q3 of Figure 7.9 with indices I1, I2 and I3 of Table 7.2.

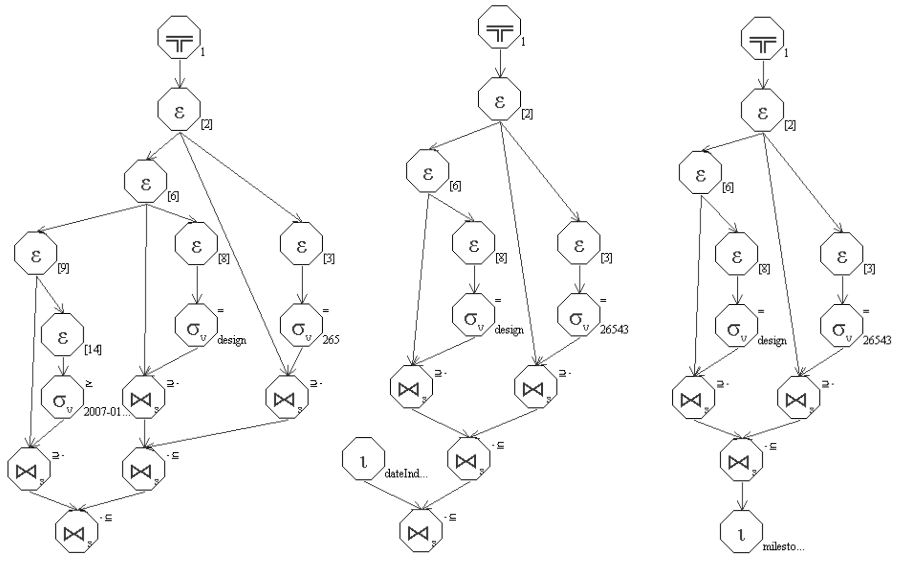


Figure 7.11: Query Q4 with no index (left), I2 (middle) and I4 (right) of Table 7.2.

input. The search configuration for the index therefore does not only specify a search condition on the date, but also includes the retrieved milestone paths (labels). As the index nests dates beneath milestones, it first traverses the index structure on milestone paths and then only needs to look within dates that belong to the requested milestones. The query plan therefore no longer requires a structural join between milestones and resources.

7.5 Summary

SemCrypt is a native, secure XML database that supports outsourcing sensitive XML documents to potentially untrustworthy storage providers. To guarantee data confidentiality and privacy, it encrypts all data with a cryptographic hash function and an encryption algorithm. To efficiently query and update encrypted documents, it exploits the schemas of documents and uses labels and indices. Labels enable SemCrypt to identify encrypted fragments and to perform certain query operations on labels instead of accessing encrypted data. With the help of flexible and selective indices, SemCrypt provides indices on frequently queried document fragments and can thus retrieve those encrypted fragments that are required for query and update processing. SemCrypt does not only use schema information within labels to optimize query processing, but also to select and maintain indices as well as to represent structural information within index structures. The concepts of this thesis are not restricted to SemCrypt, but are applicable to any native XML database. SemCrypt itself can be used as a standard, unencrypted XML database.

Chapter 8

Performance Studies

Contents

The indexing approach SCIENS provides flexible and selective indices and a maintenance algorithm to keep indices consistent with document updates. This chapter evaluates the performance of the proposed approach. Section 8.1 introduces comparison criteria and the test data sets and gives some remarks on the prototypical implementation of SCIENS. Section 8.2 studies index performance by focusing on flexibility and selectivity in indexing. The performance of the maintenance algorithm of SCIENS is investigated in Section 8.3. Section 8.4 looks at query and update processing with indices in the native XML database SemCrypt. Finally, Section 8.5 concludes the performance studies.

8.1 Introduction

The indexing approach SCIENS, which is presented in this thesis, proposes flexible and selective XML indexing. Flexibility enables a database to provide those indices that best match a query workload, whereas selectivity reduces index size and accelerates index traversal. Flexible and selective indices require a generic maintenance algorithm to keep indices consistent with document updates. To demonstrate the advantages of the proposed approach, this chapter evaluates the performance of SCIENS and compares it with existing approaches. Thereby, the focus is on the index structures, the index maintenance algorithm and the integration of SCIENS into the native XML database SemCrypt. We do not present performance studies on the labeling scheme. In our context, the labeling scheme only serves auxiliary purposes to represent and compare index keys and nodes. We therefore refer the reader to related work [40, 88, 97, 133], which compares the performance of various labeling approaches.

8.1.1 Comparison Criteria

Zobel et al. [207] propose guidelines for the presentation and comparison of indexing techniques. Based on these guidelines, we have chosen the following set of comparison criteria:

- *Query evaluation speed:* Indices have the purpose to accelerate query processing. This criteria refers to the time that an index requires to evaluate a query in comparison to other indices.
- *Disk traffic* refers to the amount of data that an index requires to evaluate a query, i.e. to the size of pages which it fetches from disk.
- *Disk space:* The amount of disk space that an index requires is measured by the size that its pages consume on disk.
- *Scalability:* The size of documents stored in databases constantly grows and the costs for storage space decreases. Index structures therefore need not only be able to handle small amount of data, but need to scale up to larger documents.
- *Index update* refers to the time which it takes to update an index when updating a document. Thereby, updates subsume insertions, deletions and modifications of documents.
- *Index construction* includes the time for constructing an index on an existing document.

As SCIENS reuses existing index structures (hash table, B+-tree, KDB-tree), we do not regard concurrency, transactions and recoverability and perform the tests in single-user mode.

All performance tests measure query evaluation speed and index update time in milliseconds (ms) and disk traffic and disk space in KB.

8.1.2 Test Data

The performance studies use two test data sets with varying document sizes. The first one corresponds to the running example of managing project resources, while the second is the XMark auction data set [173]. We have chosen the project data set to resume our running example and the XMark data set for being a public XML benchmark. Both test data sets use one large XML document. One could argue that many XML applications manage a large number of small documents. As the indices used in the performance tests refer to document fragments instead of to the entire document, they are independent of whether they index one large or several smaller documents.

As test data for the project example, we generated a 10 MB XML document consisting of project resources that is instance of the sample schema (cf. Figures 2.3 and 7.2). The document consists of 11 projects. Each project has between 5 and 30 milestones and each milestone has between 5 and 200 resources. There are 50 distinct milestone titles, 1000 distinct resource titles and 5 distinct resource types. To generate text, we used the text generator of XMach [5]. Each generated description value has between 5 and 50 sentences and each sentence has between 10 and 30 words. The generated titles have between 2 and 5 words each.

The XMark data set manages items that are sold at auctions. Figure 8.1 depicts part of the schema as a labeled schema tree. The schema groups items according to their region, i.e. each region consists of items that have the information shown in the schema. In the original XMark schema, description and text elements can contain mixed content and recursions. As the current implementation of SCIENS in SEMCRYPT neither supports mixed content nor recursion, we modified the documents such that the description and text elements contain text nodes with the entire content.

8.1.3 Implementation Notes

The basis for the performance studies is the prototypical implementation of the indexing approach SCIENS in the native XML database SEMCRYPT (cf. Chapter 7). The implementation uses Java 1.5, and Berkeley DB Java 3.1 in its default configuration [4] to store documents and indices. Data is stored locally with encryption and caches turned off. The performance studies use the Java Execution Time Measurement Library (JETM) 1.2.2 [3] to determine the time for executing queries. All experiments were carried out on a 3.2 GHz Pentium D processor with 2 GB RAM.

The Berkeley database provides a hash-based storage of id-value pairs of arbitrary sizes, which SEMCRYPT uses to store and retrieve pages. Each page represents a value and has assigned a unique id. References between pages are represented by their page ids. To store pages as values, SEMCRYPT serializes and deserializes the object representation of pages as bytes and vice versa. Serialization of a label in a page is based on the string representation of the label. Note that this approach has been chosen for ease of implementation. However, using a more efficient encoding for labels (cf. Section 3.2) would improve the overall performance and reduce disk space. As Berkeley only provides for storing id-value pairs, it is not possible to directly navigate from one page to a

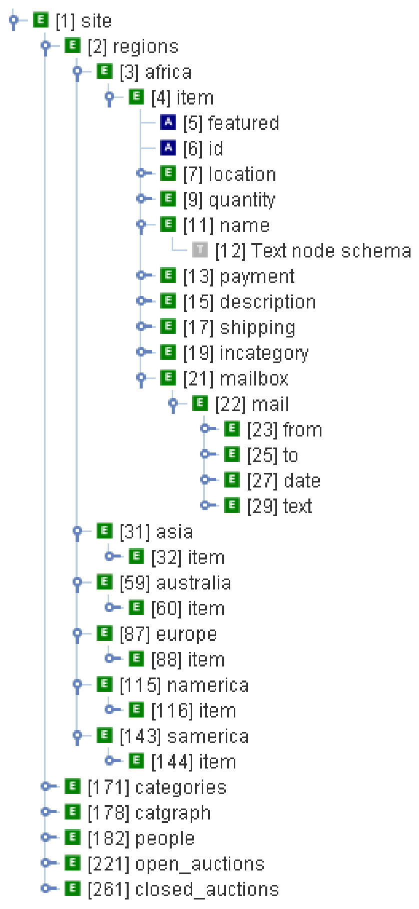


Figure 8.1: Part of the schema of the XMark auction data set.

referenced page. Instead, it is necessary to retrieve the entire page, deserialize it and find the id of the referenced page, which can then be fetched from the store. The hash-based storage increases security in an encrypted store. As it disperses related fragments, it can prevent statistical analysis. However, when using SemCrypt as an unencrypted database, it introduces additional overhead because related pages are not located next to each other. Further note that a page has a logical size in our implementation, but it need not correspond to a physical disk block. The master thesis of Dorninger [71] contains details on the implementation of storing and retrieving id-value pairs.

The index structures are configured as follows. The hash table uses the hashed-based storage of id-value pairs provided by the Berkeley database, which resolves hash collisions. The hash table currently does not use overflow pages when the size of a data page exceeds a maximum size. The KDB-tree and the B+-tree both use a fanout of at most 20 index keys in index (branch) pages and a maximum number of 70 node labels in data (leaf) pages. If there are more than 70 node labels with the same index key in a data page, the page does not further split. To split index pages, the KDB-tree basically employs a cyclic splitting strategy [169]. However, it prefers splitting dimensions with more than two distinct values. It uses this simple strategy to avoid that dimensions with a low number of distinct values, such as labelpath or type dimensions, split too often. When a dimension is chosen for a split, the KDB-tree selects the median index key of the dimension as splitting line (in lexicographic, numeric or document order). Details on the implementation of the index structures can be found in the master thesis of Lasinger [125].

As primary index structure, SemCrypt uses a hash table on node values and a B+-tree on node labels (cf. Section 7.3.3). The B+-tree uses a fanout of 50 labels per index page, while a data page contains at most 200 labels. To perform structural joins between partial query results, SemCrypt uses a tree-merge join algorithm [12].

SemCrypt maintains indices based on the maintenance algorithm presented in Chapter 6. As proposed in Subsection 6.4.1, the implementation pregenerates a list of matching pattern nodes for a path schema to find embeddings of an index pattern in an update fragment. If an update requires source queries, the algorithm executes queries with the help of labels as far as possible (cf. Subsections 3.5 and 6.4.2). Otherwise, the algorithm accesses the primary data structure to navigate to missing nodes and to retrieve their values.

8.2 Index Structures

This section evaluates the performance of the indexing approach sciens. The focus is on the comparison criteria query evaluation speed, disk traffic, disk space and scalability. sciens provides flexible and selective indexing in XML databases. Subsection 8.2.1 looks at the advantages of flexible index structures, whereas Subsection 8.2.2 compares non-selective with selective index structures. Both subsections evaluate the comparison criteria query evaluation speed, disk traffic and disk space. Subsection 8.2.3 focuses on the criteria scalability.

As this section evaluates the performance of index structures, the results only contain the time for accessing the indices. Some queries require access to the primary index structure or need to join index results. Section 8.4 contains performance studies including all execution steps. As sample test data set, we use the 10 MB project document to evaluate selectivity and flexibility. We then demonstrate scalability of selective and flexible indices based on the auction test data set.

8.2.1 Flexible Index Structures

SCIENS provides flexibility to adapt indices to various query workloads. This subsection demonstrates flexibility by comparing query evaluation speed, disk traffic and disk space of various index configurations.

Sample queries and indices

To show the effect of various index configurations, we execute three queries with four different indices. As sample data set, we use the 10 MB project document. The queries and indices are taken from the running example, which has been used throughout the thesis.

		(a)	(b)	(c)
Q1	//resource[@date ≥ **'2007-01-01'** and @date < **'2008-01-01'**]	1993	63	3
Q2	//project[title = 'semcrypt']//resource[@date ≥ **'2008-01-01'**]	217	63	1
Q3	//project[@id = '26543']/milestone[title = 'design']/resource[@date ≥ **'2008-01-01'**]	4	63	1

Table 8.1: Queries on 10 MB project document and number of resources returned for query executions (a-c) with varying date ranges.

The queries of Table 8.1 correspond to queries Q8, Q9 and Q10 of Table 2.2. These queries reflect that XML queries typically restrict search to differently sized subfragments of a document and contain value selections. While Q8 queries the date of all resources, Q2 restricts search to the resources of one project and Q3 only looks at the resources within one milestone of that project. Note that the queried date ranges, which are depicted in bold in the table, do not exactly correspond to the date ranges used in the performance tests. The tests use three different query executions (a), (b) and (c) which refer to distinct date ranges. The table lists the query result size, i.e. the number of resources returned, for each execution. Query execution (a) uses the same date range for each query. Q1 returns most resources for this execution as it does not restrict search to a project or milestone. Execution (b) varies the date range such that each query returns the same number of resources. It therefore uses a small date range for Q1 and a large date range for Q3. Execution (c) asks for an exact date.

The queries of Table 8.1 refer to resources within different projects and milestones and to various date ranges. SCIENS provides indices that support hierarchical queries within subfragments and range queries on date values. It can further adapt indices to either favor queries on the value or the hierarchy. To reflect the queried milestone hierarchy in the index, we index the paths of

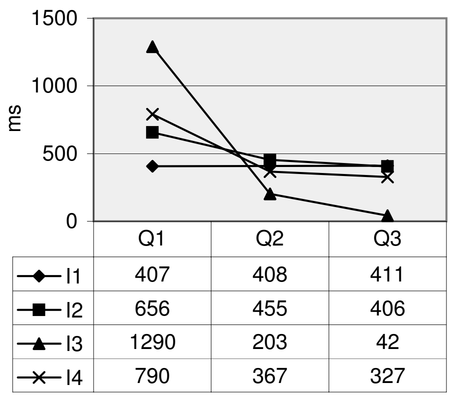
I1	B+-tree on date values
I2	B+-tree on date values, nested B+-tree on milestone paths
I3	B+-tree on milestone paths, nested B+-tree on date values
I4	KDB-tree on date values and milestone paths

Table 8.2: Indices on project document.

milestone nodes. Additionally, we index the values of resource dates. We use the indices of Table 8.2, which correspond to indices I6, I8, I9 and I10 of Table 4.4.

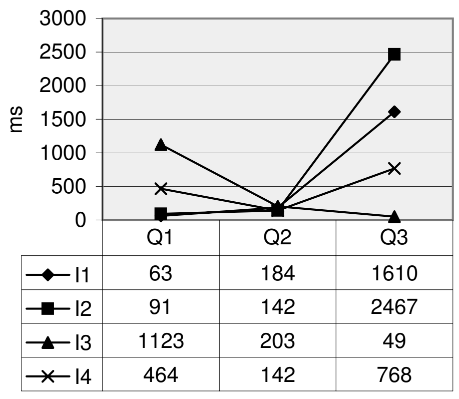
Query evaluation speed

The time for executing the queries of Table 8.1 with the indices of Table 8.2 depends on the queried date range and the number of affected resources, i.e. on whether the query refers to all projects, one project or one milestone.



	Q1	Q2	Q3
◆ I1	407	408	411
■ I2	656	455	406
▲ I3	1290	203	42
× I4	790	367	327

Figure 8.2: Query evaluation speed (a) - same date range.



	Q1	Q2	Q3
◆ I1	63	184	1610
■ I2	91	142	2467
▲ I3	1123	203	49
× I4	464	142	768

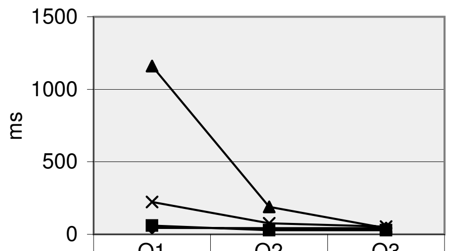
Figure 8.3: Query evaluation speed (b) - same result size.

Figure 8.2[1] depicts the time in milliseconds for query execution (a), which uses the same date range for each query. I1 performs equally for every query as it cannot limit search to a project or milestone. I2, which first groups index entries according to their date, is slightly less efficient than I1 for queries Q1 and Q2 because it also indexes milestone paths. However, with regard to Q3, the nested index on milestone paths allows for discarding resources that are not within the queried milestone. Index I3 has to traverse every nested date index. It therefore shows poor performance for Q1 and best performance for Q3. The KDB-tree I4 has average performance for each query.

Query execution (b) returns the same number of resources for each query by varying the date range. Figure 8.3 depicts the resulting query evaluation time. We can see that I1, I2 and I4 deteriorate when increasing the date range. I2 performs worse than I1 for queries Q1 and Q3 and I4 has average performance for each query. Regarding Q2, we can observe that I2 and I4 are more efficient

[1]We use line instead of point diagrams for easier comparison.

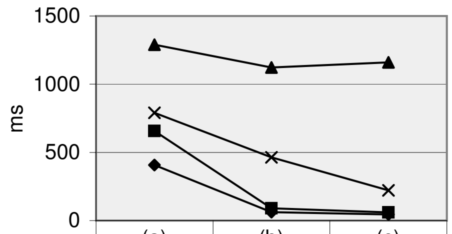
than I1. The reason is that I1 returns all resources with the requested date range, which need to be filtered afterwards to restrict search to the queried project. On the contrary, I2 and I4 can already filter the resources according to their project when traversing the index. With regard to Q3, I2 performs worse than I1 because it has to visit a large number of nested index structures. I3 improves when restricting search to certain subtrees as it only needs to look at the dates of the queried project or milestone. Therefore, although the queried date range remains unchanged, it can limit the date range by restricting index traversal to the queried subtree.



	Q1	Q2	Q3
—◆— I1	44	42	42
—■— I2	59	29	29
—▲— I3	1160	190	43
—×— I4	222	76	53

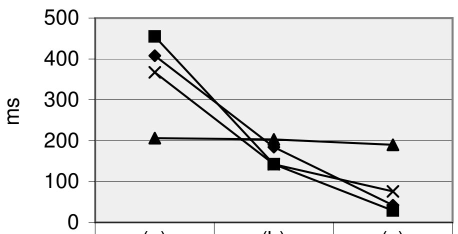
Figure 8.4: Query evaluation speed (c) - exact date.

Figure 8.4 compares the performance of the sample indices for query execution (c), which asks for an exact date. The overall picture corresponds to Figure 8.2 except for that the query evaluation time improves by limiting search to one date. Especially I1, I2 and I4 profit from only having to look for one date.



	(a)	(b)	(c)
—◆— I1	407	63	44
—■— I2	656	91	59
—▲— I3	1290	1123	1160
—×— I4	790	464	222

Figure 8.5: Q1 evaluation speed for query executions (a-c).



	(a)	(b)	(c)
—◆— I1	408	184	42
—■— I2	455	142	29
—▲— I3	206	203	190
—×— I4	367	142	76

Figure 8.6: Q2 evaluation speed for query executions (a-c).

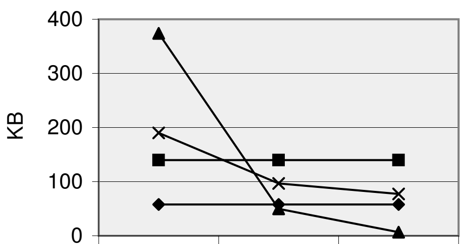
Figures 8.5 and 8.6 compare the time for evaluating Q1 and Q2 dependent on the queried date range (query executions (a-c)). Regarding Q1, we can see that I1 performs best as Q1 looks at all resources. I2 is slightly worse than I1 as it nests an index on milestone paths beneath the date values. I3 performs worst as it needs to visit one (nested) index structure on the date for every milestone path. In contrast to I2 and I3, I4 circularly splits the indexed dimensions, entailing that its performance is between I2 and I3.

Q2 looks at all resources of one project with a decreasing date range from query execution (a) to (c). For every query execution, I3 has to visit the same number of nested index structures on the date. However, as there is one nested index structure on the date for every milestone, each nested index structure is very small in size. Traversing this index structure hardly affects query evaluation time. I3 therefore has equal performance for every query execution. The performance of the remaining indices improves when decreasing the date range. As I1 and I2 primarily build a B+-tree on the date, they profit from a lower date range even more than I4.

The results show that solely indexing date values is only appropriate when querying all resources (I1, Q1) as this index cannot limit index traversal to certain milestones. The multidimensional index (I4) has average performance, while nesting index structures either favors querying the entire hierarchy (I2, Q2) or a specific part of the hierarchy (I3, Q3).

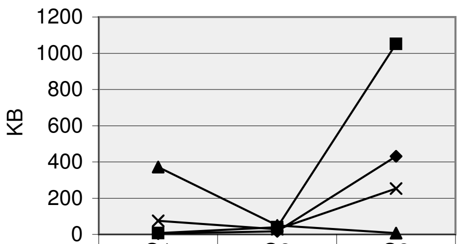
Disk traffic

Each index needs to fetch pages from disk to evaluate queries. The size and number of pages fetched strongly influences query evaluation time.



	Q1	Q2	Q3
◆ I1	58	58	58
■ I2	140	140	140
▲ I3	374	50	6
× I4	190	97	77

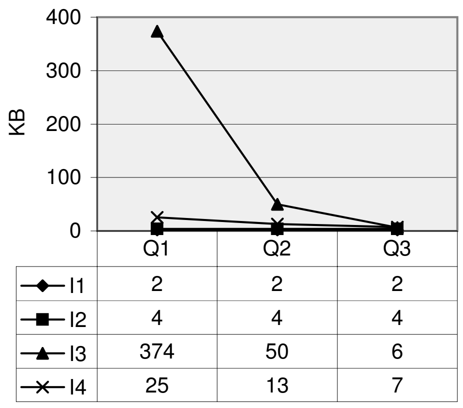
Figure 8.7: Disk traffic (a).



	Q1	Q2	Q3
◆ I1	4	17	431
■ I2	8	40	1051
▲ I3	372	50	6
× I4	75	28	253

Figure 8.8: Disk traffic (b).

Figures 8.7, 8.8 and 8.9 compare the disk traffic, i.e. the size of pages fetched in KB, to evaluate query executions (a-c) with the sample indices. We can observe that the query evaluation speed is proportional to the disk traffic. I1 requires least disk traffic as it is one-dimensional. I2 requires more disk traffic than I1. When querying the same date range (a), I1 and I2 require constant disk traffic. The disk traffic of I3 decreases when adding a selection on the path variable. The multidimensional index I4 always fetches an average size of data from disk.

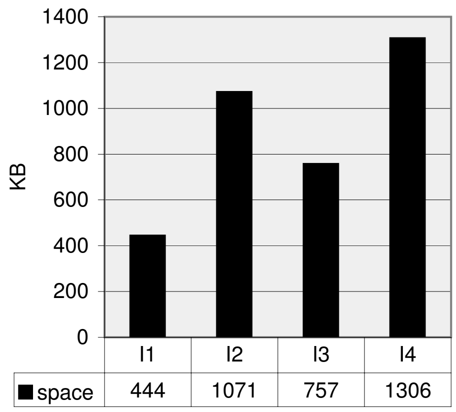


	Q1	Q2	Q3
◆ I1	2	2	2
■ I2	4	4	4
▲ I3	374	50	6
× I4	25	13	7

Figure 8.9: Disk traffic (c).

Disk space

The entire disk space of an index depends on the number of indexed dimensions and the kind of index structures used.



	I1	I2	I3	I4
■ space	444	1071	757	1306

Figure 8.10: Disk space of indices I1-I4.

Figure 8.10 shows the total size which is required by the sample indices on disk in KB. I1 is the smallest index as it only indexes one dimension, i.e. date values. I2 and I3 contain nested index structures. I3 builds one nested index on dates for each milestone, while I2 nests the index structures the other way round. As there are fewer distinct milestones than distinct dates, I3 has fewer nested index structures than I2 and therefore requires smaller space than I2. I4 requires most space as it needs to store splitting information for both dimensions.

Comparison

To index the document content and structure, SCIENS provides the flexibility to adapt indices to various query workloads. It basically offers (i) an index on the document content, (ii) an index on the document content with a nested index on the document structure, (iii) an index on the document structure with a nested index on the document content, and (iv) a multidimensional index on the document content and structure.

With regard to query evaluation speed, alternative (i) performs best for queries that do not restrict the hierarchical document structure. If queries only rarely refer to a subhierarchy or query a large part of the document structure, alternative (ii) is preferable. Alternative (iii) has best performance when queries mostly limit search to a specific subhierarchy. The multidimensional index (iv) is preferable when queries do not favor certain dimensions, i.e. are one time more selective regarding the content, another time regarding the document structure.

The disk traffic is proportional to the query evaluation time. The faster an index is, the fewer pages does it need to fetch from disk. Concerning the overall disk space, we can observe that multidimensional indices need to store splitting information and contain more index keys, resulting in an increased storage overhead. When nesting index structures, the number of distinct index keys in the superior index structure determines the number of nested index structures. Therefore, the lower the number of distinct index keys is in the superior index structure, the smaller is the size of the entire index.

8.2.2 Non-selective vs. Selective Index Structures

SCIENS provides selective indices to define indices on document fragments instead of on entire documents only. Selective indices are smaller in size, which accelerates index traversal, but they can only answer queries referring to the indexed fragments. This subsection contrasts non-selective and selective indices with regard to query evaluation speed, disk traffic and disk space. Non-selective indices are comparable to existing index structures. This subsection can therefore also be seen as a comparison of SCIENS with related work. As SCIENS can express existing non-selective index structures, we use the prototypical implementation of SCIENS for both non-selective and selective indices.

Sample queries and indices

To contrast non-selective and selective indices, we compare four queries and four indices based on the sample project document of 10 MB. We create each index (i) on the entire document and (ii) on the document fragment that is relevant for executing the queries.

Q1	//title	15425
Q2	//milestone/title	157
Q3	//title[. = ‘design’]	57
Q4	//milestone/title[. = ‘design’]	3

Table 8.3: Queries on project document and result size.

Table 8.3 depicts sample queries on the project document. Q1 and Q2 are structural queries, whereby Q1 selects all titles and Q2 limits search to milestone titles. Q3 and Q4 contain a value selection. While Q3 selects all titles with a certain value, Q4 only selects milestone titles with that value. The right-most column contains the number of nodes returned when executing the queries on the sample 10 MB project document.

I1	B+-tree on labelpaths
I2	Hash table on node names, B+-tree on values
I3	B+-tree on labelpaths with nested B+-tree on values
I4	B+-tree on values with nested B+-tree on labelpaths

Table 8.4: Non-selective indices on project document.

To support the sample queries, we generated four indices, as depicted in Table 8.4. I1 is a structural index, whereas I2-I4 index the document content and structure. Each index returns the nodes whose properties it indexes, i.e. the index patterns consist of a single pattern node. For example, I1 returns for every labelpath all nodes that can be reached along the labelpath. The sample indices correspond to existing XML index structures. I1 is comparable to the DataGuide [85], I2 to XISS [135], I3 to the Index Fabric and the structure-oriented CADG [59, 195] and I4 to the content-oriented CADG [195].

I1	B+-tree on title labelpaths
I2	Hash table on names of title nodes, B+-tree on values of title nodes
I3	B+-tree on title labelpaths with nested B+-tree on values of title nodes
I4	B+-tree on values of title nodes with nested B+-tree on title labelpaths

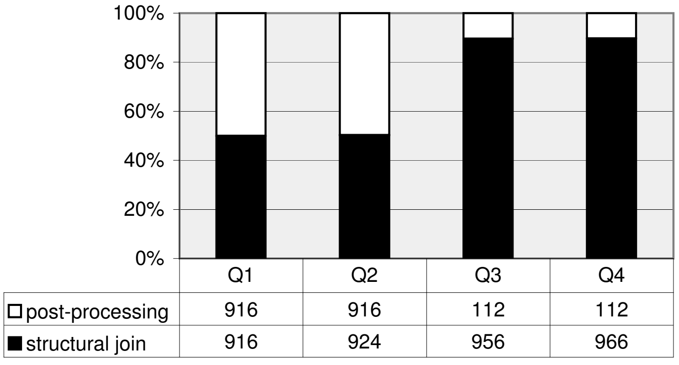
Table 8.5: Selective indices on project document.

The indices of Table 8.5 correspond to the indices of Table 8.4 except for that they are selective. While I1 of Table 8.4 indexes all labelpaths, I1 of Table 8.5 only refers to labelpaths of title nodes. Similarly, the remaining indices of Table 8.5 only index the names, labelpaths and/or values of title nodes.

Query evaluation speed

The time for executing the sample queries of Table 8.3 depends on whether the accessed indices are not selective (cf. Table 8.4) or selective (cf. Table 8.5). I1 can only answer queries Q1 and Q2 as it does not index the document content. The remaining indices contain all information required to answer the sample queries.

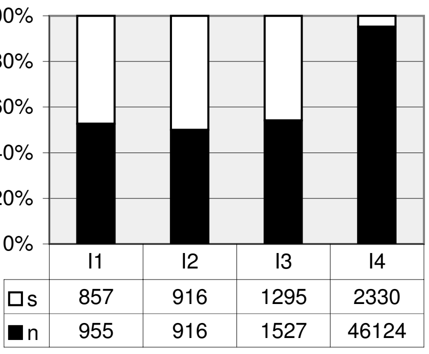
I2 builds two separate indices, a hash table on node names and a B+-tree on node values. The equivalent indexing approach XISS [135] proposes to access the indices for each queried node name and value and then join index results. For example, to evaluate Q2, the database can access the hash table on node names twice, once to retrieve all milestone nodes and once to retrieve all title nodes. To only return milestone titles, it is then necessary to join the results of the index accesses. The indices of SCIENS return node labels that encode structural information into schema labels (cf. Chapter 3). With the help of these schema labels, it is possible to post-process the nodes returned by indices. With regard to Q2, for example, it is possible to access the hash table on node names to retrieve all titles and then extract the requested milestone titles based on schema labels.



	Q1	Q2	Q3	Q4
□ post-processing	916	916	112	112
■ structural join	916	924	956	966

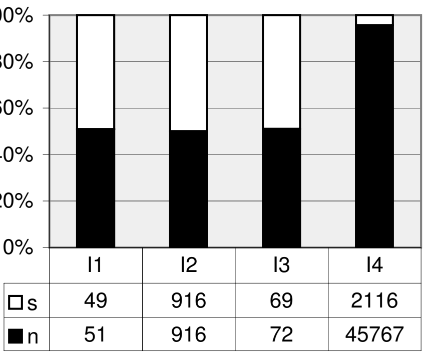
Figure 8.11: Query evaluation speed with I2 in ms by post-processing and structural joins.

Figure 8.11 contrasts executing the sample queries with the non-selective index I2 by either performing structural joins or post-processing the index result. In contrast to structural joins, post-processing requires one index access for each query. The more node names are queried, the more efficient is this approach. We therefore use post-processing in the following. Note that we generally only show the time for accessing the indices and not for post-processing index results.



	I1	I2	I3	I4
□ s	857	916	1295	2330
■ n	955	916	1527	46124

Figure 8.12: Q1 evaluation speed with (non-)selective indices.



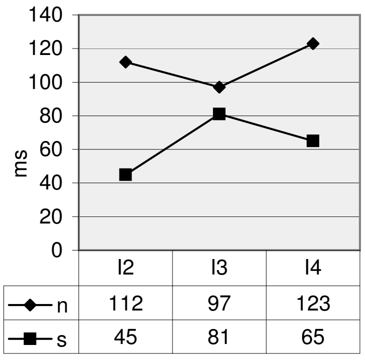
	I1	I2	I3	I4
□ s	49	916	69	2116
■ n	51	916	72	45767

Figure 8.13: Q2 evaluation speed with (non-)selective indices.

Figure 8.12 shows the time for executing query Q1 with selective (s) and non-selective (n) indices in milliseconds. Concerning the structural query Q1, indices I1 and I2 perform best. Note that in the selective case, I1 performs better than I2 although I1 is a B+-tree and I2 a hash table. The reason is that I2 returns a large number of titles, which are all contained in one page as the hash table currently does not split its pages. On the contrary, I1 splits the titles according to their labelpath. Accessing several smaller pages is slightly more efficient than accessing one large page in the prototypical implementation. In the non-selective case, I1 consists of more labelpaths, which slightly slows down index traversal. Indices I3 have to visit every nested value index although the query only restricts the labelpath of nodes to be returned. They therefore perform worse than I1 and I2. Indices I4 have the worst performance because they need to traverse the entire index on values and then access every nested index on labelpaths.

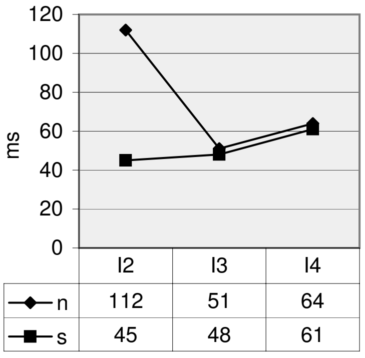
Because of the big differences in the query evaluation time, the figure contrasts selective and non-selective indices in percent. As the total number of labelpaths is very small, the performance of the index structures on the labelpaths (I1 and I3) only varies slightly between selective and non-selective indices. I2 requires one access to the hash table on node names to evaluate Q1 and therefore has equal performance in the non-selective and the selective case. The performance of I4 greatly improves when building a selective index as this index has to visit fewer nested index structures.

Executing Q2 with the sample indices shows similar results to the execution of Q1, as depicted in Figure 8.13. Compared to Q1, Q2 restricts search to one labelpath. The performance of indices I3 is nearly comparable to the ones of I1. Compared to I1, indices I3 have to visit one nested index structure on the date after having traversed the index on the labelpath.



	I2	I3	I4
◆ n	112	97	123
■ s	45	81	65

Figure 8.14: Q3 evaluation speed with (non-)selective indices.



	I2	I3	I4
◆ n	112	51	64
■ s	45	48	61

Figure 8.15: Q4 evaluation speed with (non-)selective indices.

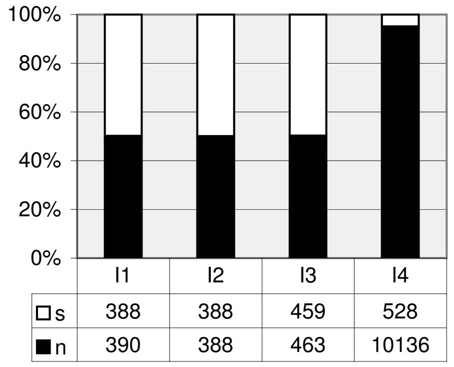
Queries Q3 and Q4 contain a value selection and are therefore not supported by I1. Figures 8.14 and 8.15 contrast the time for executing Q3 and Q4 with the sample indices I2-I4.

Regarding Q3, we can observe that the performance of I3 is least dependent on whether the index is selective or not. As the number of indexed labelpaths is small in the sample project document, also the non-selective variant can quickly restrict search to the queried titles. I2 has to retrieve all nodes with the queried value and then post-process the returned nodes. Q3 returns 57 title nodes, but the sample document contains 311 more nodes with the queried value. These nodes are returned by the non-selective index and need to be filtered. In contrast, the selective index variant of I2 only requires to access the value index to retrieve all queried titles. I4 performs similarly to I2. The non-selective variant needs to traverse a greater number of nested index structures and therefore performs worse than the selective counterpart.

Query Q4 restricts search to one labelpath and one value. Indices I3 and I4 can quickly limit search to the queried labelpath and value, respectively. Therefore, the performance of the selective and non-selective index variant does not greatly differ. Indices I2 have the same performance for Q4 as for Q3. The reason is that they have to evaluate the same search condition on the value index. Note that the time for post-processing the index result, i.e. to filter relevant titles, is not included. Compared to Q3, Q4 has to filter 54 resource titles which have the same title as the requested milestone.

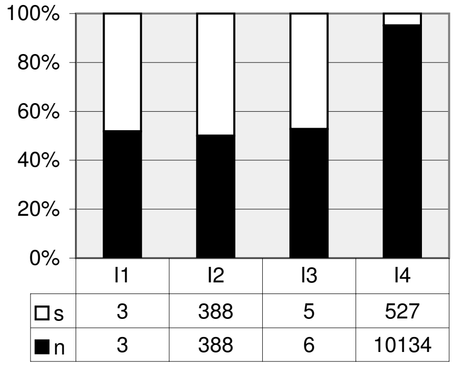
Disk traffic

The number of pages which an index needs to fetch from disk influences query evaluation speed. Non-selective indices need to retrieve and filter more data from disk, which deteriorates their query performance.



	I1	I2	I3	I4
□s	388	388	459	528
■n	390	388	463	10136

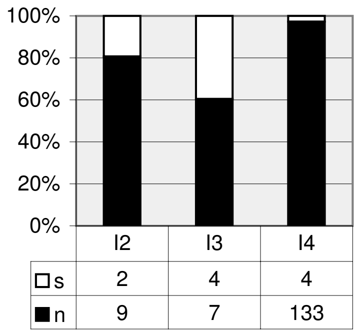
Figure 8.16: Disk traffic Q1.



	I1	I2	I3	I4
□s	3	388	5	527
■n	3	388	6	10134

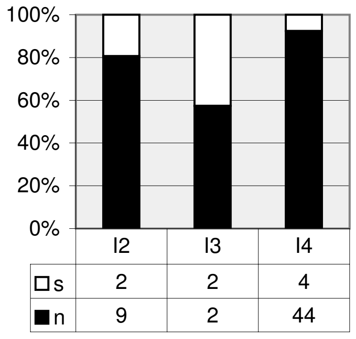
Figure 8.17: Disk traffic Q2.

Figures 8.16 and 8.17 contrast the disk traffic in KB for the various indices when executing query Q1 and Q2, respectively. The results are comparable to the query evaluation speed (cf. Figures 8.12 and 8.13). The reason is that the more data is accessed, the poorer is the query performance.



	I2	I3	I4
□s	2	4	4
■n	9	7	133

Figure 8.18: Disk traffic Q3.



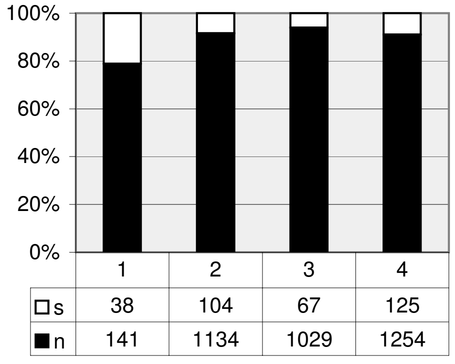
	I2	I3	I4
□s	2	2	4
■n	9	2	44

Figure 8.19: Disk traffic Q4.

Figures 8.18 and 8.19 contrast the disk traffic for evaluating queries Q3 and Q4 with selective and non-selective indices. The disk traffic is proportional to the query evaluation speed except for query Q4 with index I4. In this case, the non-selective index requires larger disk traffic than the selective one although the query evaluation speed is nearly the same. However, as both indices require small amount of data to evaluate Q4, the disk traffic has less impact on query evaluation speed.

Disk space

While non-selective indices refer to the entire document, selective indices are only defined on certain fragments of a document. Dependent on the size of the indexed fragments, selective indices occupy less disk space than non-selective indices.



	1	2	3	4
□ s	38	104	67	125
■ n	141	1134	1029	1254

Figure 8.20: Disk space of non-selective and selective indices on a 1 MB document in KB.

Figure 8.20 contrasts the disk space occupied by non-selective and selective indices. Note that we used a 1 MB document, consisting of only one project, to calculate disk space. Non-selective indices clearly need more space than selective ones. I2-I4 approximately have the same size as the document as they represent the entire document.

Comparison

Selective indices only index the queried fragment and can answer queries more efficiently than non-selective indices. Content-oriented indices show better performance improvements than structure-oriented indices because the document content is generally larger than the document structure.

As selective indices only refer to certain fragments of a document, they are smaller in size and require less disk traffic to evaluate queries. However, it needs to be added that selective indices can only answer the queries that refer to the indexed fragments. SCIENS offers the possibility to create non-selective and selective indices. It can therefore provide indices on those fragments that are accessed by queries.

8.2.3 Scalability

This subsection studies the performance of indices when increasing the document size. More precisely, we compare the query evaluation speed of various queries and indices. We omit disk traffic and disk space because the results for these criteria do not differ from those of the previous subsection.

Sample queries and indices

As sample document, we use the XMark auction data set and execute four queries with five different indices.

		10 MB	50 MB	100 MB
Q1	//item	1882	9410	18820
Q2	//europe//item	540	2700	5400
Q3	//item[fn:matches(name, '\bho.*')]	52	260	520
Q4	//europe//item[fn:matches(name, '\bho.*')]	12	60	120

Table 8.6: Queries on auction data set and result size (number of items returned).

Table 8.6 lists sample queries on the auction data set. Q1 and Q2 are structural queries that retrieve all items (Q1) and items sold in Europe (Q2). Q3 and Q4 similarly query all items and European items, respectively. They additionally contain a value selection on the item name to only return those items whose name contains the word prefix '`ho`'. The table also includes the number of items returned when varying the document size. Note that we actually generated a 10 MB document and reinserted this document several times, which explains the constant increase in the result size.

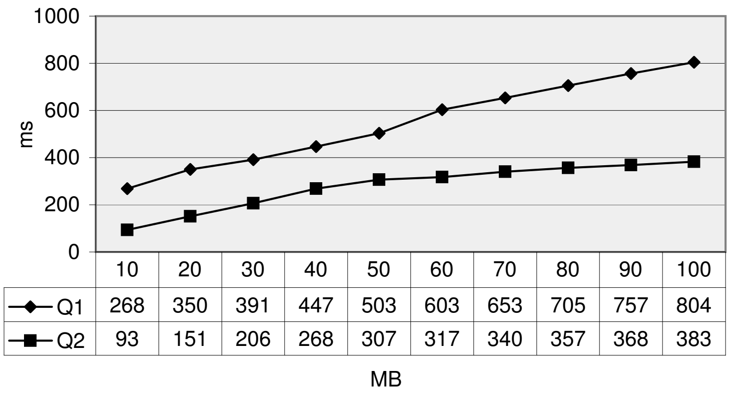
I1	B+-tree on the labelpath of items
I2	Prefix B+-tree on the value of item names
I3	Prefix B+-tree on the value of item names, nested B+-tree on the labelpath of items
I4	B+-tree on the labelpath of items, nested prefix B+-tree on the value of item names
I5	KDB-tree on the value of item names and the labelpath of items

Table 8.7: Indices on auction data set.

We use the indices of Table 8.7 to execute the sample queries of Table 8.6. I1 is a structural index on the labelpath of items and I2 indexes the value of item names. I3-I5 index both the labelpath of items and the value of item names. Thereby, I3 and I4 nest the labelpath beneath the value and vice versa, while I5 uses a multidimensional index.

Query evaluation speed

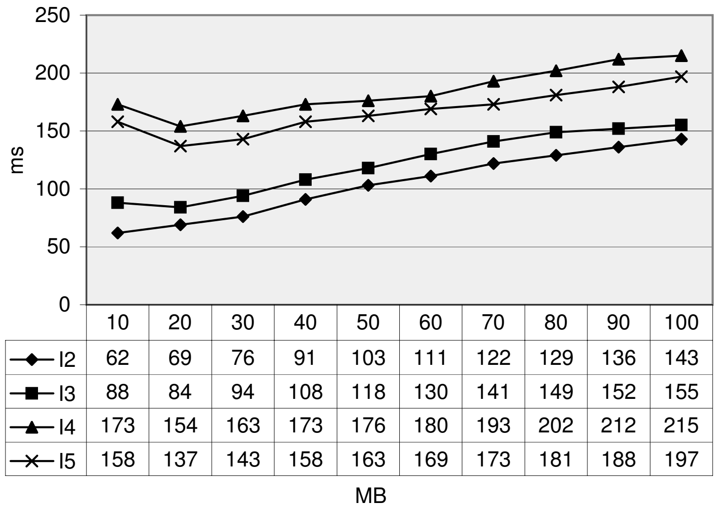
In the following, we compare the time for evaluating the sample queries of Table 8.6 with the indices of Table 8.7 by increasing the document size from 10 to 100 MB. We first look at evaluating queries Q1 and Q2 with the structural index I1. We then use the remaining indices to execute queries Q3 and Q4.



	10	20	30	40	50	60	70	80	90	100
—◆— Q1	268	350	391	447	503	603	653	705	757	804
—■— Q2	93	151	206	268	307	317	340	357	368	383

Figure 8.21: Time for evaluating Q1 and Q2 with index I1.

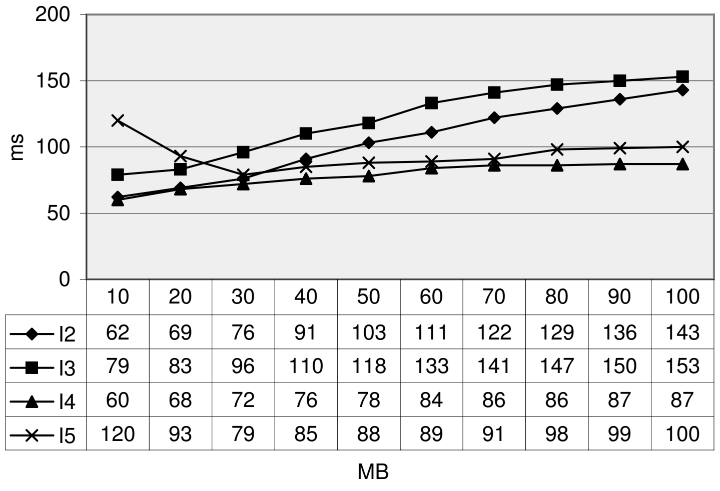
Considering Figure 8.21, the time for evaluating queries Q1 and Q2 with index I1 constantly increases. As Q2 only refers to one labelpath, the index can evaluate this query more efficiently than Q1.



	10	20	30	40	50	60	70	80	90	100
—◆— I2	62	69	76	91	103	111	122	129	136	143
—■— I3	88	84	94	108	118	130	141	149	152	155
—▲— I4	173	154	163	173	176	180	193	202	212	215
—×— I5	158	137	143	158	163	169	173	181	188	197

Figure 8.22: Time for evaluating Q3 with indices I2-I5.

Figure 8.22 depicts the time for executing query Q3 with indices I2-I5. Index I2, which only indexes node values, is most efficient as the query refers to all item labelpaths. I3 is slightly less efficient than I2. Index I4 has to traverse all value index structures which are nested beneath the B+-tree on item labelpaths and thus performs worst. The performance of the KDB-tree I5 is between the nested indices I3 and I4. The figure also shows that the performance of the multidimensional indices I3-I5 first decreases when increasing the document size. The reason it that the indices only refer to a small part of the document. As long as the number of index entries is small, there are only few page splits. Data pages then contain more nodes than are requested, but need to be entirely retrieved.



	10	20	30	40	50	60	70	80	90	100
—◆— I2	62	69	76	91	103	111	122	129	136	143
—■— I3	79	83	96	110	118	133	141	147	150	153
—▲— I4	60	68	72	76	78	84	86	86	87	87
—×— I5	120	93	79	85	88	89	91	98	99	100

Figure 8.23: Time for evaluating Q4 with indices I2-I5.

In contrast to Q3, query Q4 only looks at European items. Figure 8.23 contrasts the query evaluation speed of indices I2-I5 for this query. Note that index I2 returns all items, which would have to be filtered after accessing the index. We still included I2 for better comparison. Index I3 is similar to I2, but can filter the requested items after evaluating the search condition on the item name. As it has to traverse one B+-tree on item labelpaths for every distinct word of item names, it performs worse than I2. In total, index I4, which builds one B+-tree on item names for each distinct labelpath, performs best. The performance of the KDB-tree I5 is between the performance of the nested indices I3 and I4. However, I5 performs worst when the document size is very small. The reason is that in case of a low index size and few page splits, the data pages, which are accessed, contain more nodes than are requested.

Comparison

In this subsection, we have shown that the indices scale well when increasing the document size. The overall results correspond to Subsection 8.2.1 in that each index better supports different kinds of queries. In case of a low index size, the performance of multidimensional indices may be worse than for a larger document. The reason is that few page splits entail that data pages contain more nodes than are requested. If an index only contains a small number of index entries, it is advisable to decrease the maximum number of labels in a data page.

8.3 Index Maintenance

This section evaluates the index maintenance algorithm of SCIENS by focusing on the comparison criteria index update and index construction. The maintenance algorithm propagates updates on documents to affected indices. Updates comprise the insertion, deletion and modification of document fragments. The algorithm receives as input index patterns and update fragments and generates index entries by performing the following steps: (1) find embeddings of index patterns in update fragments, (2) execute queries to retrieve nodes missing in the update fragments, (3) generate index entries and forward these index entries to the index structures for maintenance. Each index structure provides algorithms to insert or delete index entries. As SCIENS uses well-known index structures, we do not evaluate the time for updating the index structures as such. Instead, we focus on the performance of the maintenance algorithm, i.e. on extracting index entries from update fragments.

In the following, we present sample index patterns and update fragments in Subsection 8.3.1, study the time for updating indices in Subsection 8.3.2 and the time for constructing indices in Subsection 8.3.3.

8.3.1 Index Patterns and Update Fragments

The maintenance algorithm receives as input index patterns, which define the indexed fragments, and update fragments, which are inserted, deleted or modified. The performance of the maintenance algorithm depends on what the index is defined on and thus on the shape of the index pattern. Further, the size of the update fragment influences performance. We therefore compare the effect of different index patterns and update fragments on index maintenance.

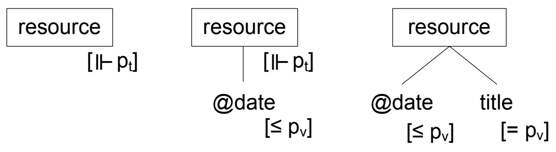


Figure 8.24: Index patterns on project document.

Figure 8.24 depicts three index patterns on the sample project document. The left-most pattern defines an index on resource types and only consists of one pattern node. The index pattern in the middle is a path index pattern and indexes resource types and date values. The right-most pattern defines an index on the values of resource dates and titles and has the form of a twig pattern.

U1	1 date	10 B
U2	1 resource	0.5 KB
U3	1 milestone with 100 resources	70 KB
U4	1 project with 10 milestones and 100 resources each	694 KB

Table 8.8: Update fragments and their size.

We update the indices with the fragments of Table 8.8. By varying the size of the update fragments, we take into account that updates in XML typically do not refer to single nodes, but to document fragments.

8.3.2 Index Update Time

This subsection studies the performance of the maintenance algorithm of SCIENS when inserting, deleting or modifying document fragments in a document. To keep indices defined on that document consistent with updates, the maintenance algorithm extracts index entries from the update fragments. Thereby, the algorithm performs the same steps regardless of whether the update is an insertion, a deletion or a modification. We therefore simply consider updates and do not specify the kind of update operation.

In the performance tests, we use a document containing one resource as base document. We then successively insert the four update fragments into the document and update each index. As the maintenance algorithm is independent of specific index structures, we do not include the time for updating the index structures with the generated index entries.

Step 2 of the algorithm executes queries to retrieve nodes that are missing in the update fragments. The performance of this step depends on the document size and the kind of index structures which a database uses. In the tests, queries to ancestors are executed via labels, whereas queries to descendants access the primary data structure of SemCrypt (cf. Subsection 7.3.3).

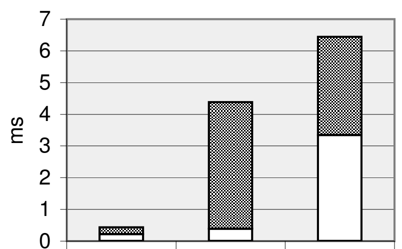
Existing XML index maintenance algorithms (cf. Subsection 6.2.3) have in common that they process each node of an update fragment individually. To compare the SCIENS maintenance algorithm with existing approaches, we first update indices with document fragments and then with single nodes.

Updating Indices with Document Fragments

When updating an index with a document fragment, the maintenance algorithm finds embeddings of the index pattern in the fragment (S1), executes queries to nodes that are not part of the update fragment (S2) and generates index entries (S3). In the following, we show the time required for performing steps S1-S3 of the maintenance algorithm when updating indices I1-I3 with the sample update fragments U1-U4.

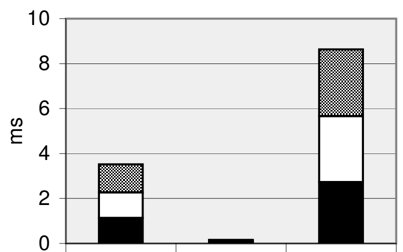
Figure 8.25 depicts the time required to update fragment U1 subdivided according to each maintenance step and each index. Update fragment U1 only consists of one date node. Index I1 is not affected by this update. When updating index I2, the algorithm has to determine the resource type to which the date belongs. It can efficiently perform this navigation with the help of the label of the date node. To update index I3, the algorithm has to query the title of the resource.

Update fragment U2 contains an entire resource. As each index entry of the sample indices refers to one resource, the algorithm does not need to query missing nodes. In Figure 8.26, we can observe that the maintenance algorithm requires the same update time for each index when updating U2.



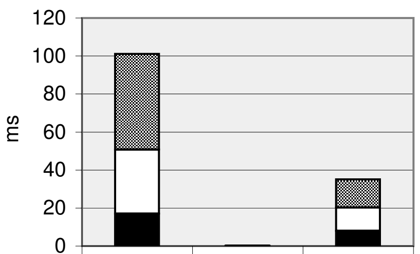
	S1	S2	S3
▨ I3	0	4	3
□ I2	0	0	3
■ I1	0	0	0

Figure 8.25: Updating fragment U1.



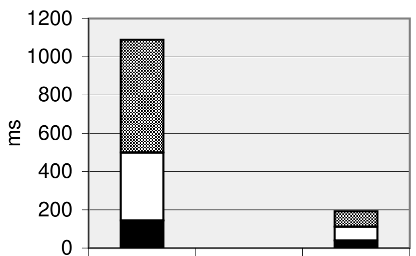
	S1	S2	S3
▨ I3	1	0	3
□ I2	1	0	3
■ I1	1	0	3

Figure 8.26: Updating fragment U2.



	S1	S2	S3
▨ I3	50	0	15
□ I2	34	0	12
■ I1	17	0	8

Figure 8.27: Updating fragment U3.



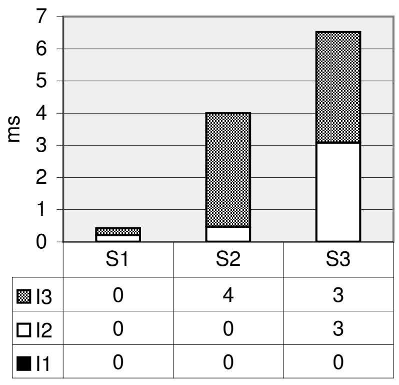
	S1	S2	S3
▨ I3	590	0	81
□ I2	354	0	73
■ I1	144	0	38

Figure 8.28: Updating fragment U4.

Figures 8.27 and 8.28 show the time for updating fragments U3 and U4. The time for finding embeddings increases with the size of the update fragment and the number of affected index entries. Each update fragment contains all nodes that are required for index maintenance, entailing that the algorithm does not have to execute step 2.

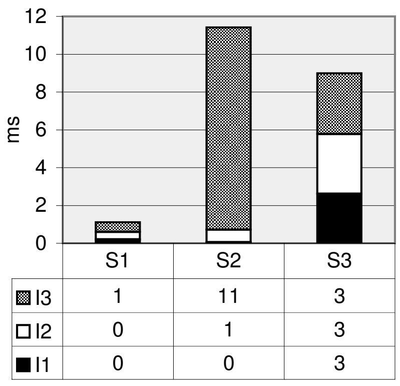
Updating Indices with Single Nodes

Existing maintenance algorithms do not consider update fragments as a whole but process each node of the fragment individually (cf. [94] and Subsection 6.2.3). For each node, they first determine whether the node affects the index, then query missing nodes and finally generate an index entry. As such, the basic steps required correspond to the maintenance algorithm of SCIENS. The main difference is that they also have to query nodes that are contained in the update fragment. In the following, we reuse the maintenance algorithm of SCIENS, but this time we split the fragment into single nodes and process each node individually.



	S1	S2	S3
▨ I3	0	4	3
□ I2	0	0	3
■ I1	0	0	0

Figure 8.29: Updating nodes of fragment U1.



	S1	S2	S3
▨ I3	1	11	3
□ I2	0	1	3
■ I1	0	0	3

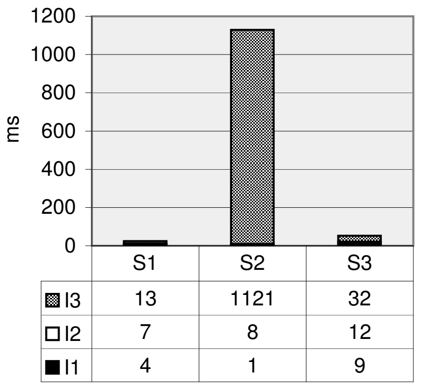
Figure 8.30: Updating nodes of fragment U2.

As update fragment U1 contains a single node, its update time, which is depicted in Figure 8.29, is the same as when processing update fragments as a whole. When updating U2 (cf. Figure 8.30), the maintenance algorithm first processes the date of the resource and then the title of the resource. With regard to index I3, it therefore has to execute two source queries, once to retrieve the title belonging to the date and once to retrieve the date that belongs to the updated title.

Looking at Figures 8.31 and 8.32, we can observe that the time required for executing queries heavily increases for update fragments U3 and U4. When processing single nodes, the maintenance algorithm executes source queries instead of extracting nodes from the update fragment.

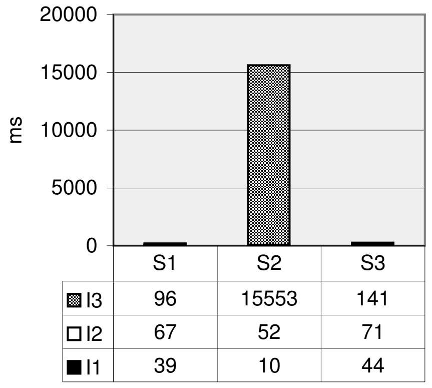
Comparison

The time required for updating indices depends on the size of the update fragment and the kind of index pattern. If an index pattern only consists of one



	S1	S2	S3
▨ I3	13	1121	32
□ I2	7	8	12
■ I1	4	1	9

Figure 8.31: Updating nodes of fragment U3.

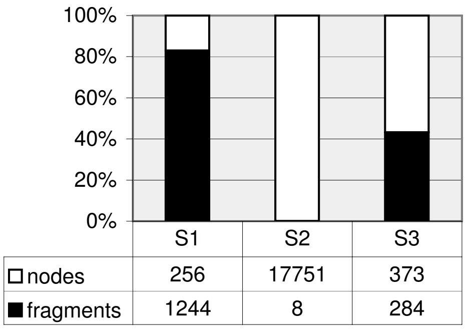


	S1	S2	S3
▨ I3	96	15553	141
□ I2	67	52	71
■ I1	39	10	44

Figure 8.32: Updating nodes of fragment U4.

pattern node, it is only necessary to find the nodes in the fragment that match the pattern node. In case of a path index pattern, it may be necessary to query missing ancestors. In SCIENS, such queries can be performed with the help of the labeling scheme. When updating a twig index pattern, it may be necessary to query missing descendants that are not contained in the update fragment. As such queries are expensive, it is more efficient to extract nodes from update fragments instead of querying them from base data.

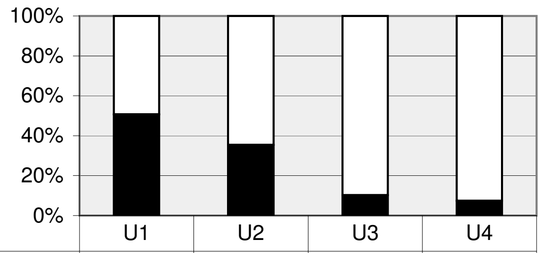
The performance of steps 1-3 of the maintenance algorithm depends on whether the algorithm processes update fragments as a whole or each update node individually. In the following, we compare these approaches by summing up the time for updating the sample indices.



	S1	S2	S3
□ nodes	256	17751	373
■ fragments	1244	8	284

Figure 8.33: Time for performing steps S1-S3 when processing update fragments and single nodes.

Figure 8.33 shows the total time (ms) which is required for each step of the algorithm dependent on whether the algorithm considers fragments or single nodes. Step 2 clearly requires more time when processing single nodes instead of fragments. Considering step 1, the algorithm traverses the entire update fragment to find embeddings. The larger the update fragment is, the more embeddings does the algorithm find and encode into stacks. As the current implementation uses a data structure with logarithmic access cost for the stack encoding, the time for finding embeddings increases with the size of the update fragment. Therefore, step 1 requires more time when processing entire fragments. Note that the performance of this step could be improved by directly generating an index entry as soon as an embedding has been found instead of keeping it in the stack encoding.



	U1	U2	U3	U4
☐ nodes	11	22	1206	16073
■ fragments	11	12	136	1280

Figure 8.34: Time for updating fragments U1-U4 when processing update fragments and single nodes.

The size of the update fragment influences the time required for maintaining the sample indices. Figure 8.34 depicts the total time (ms) required for updating the sample indices with each fragment dependent on whether the algorithm processes update fragments or single nodes. As update fragment U1 only consists of one node, both approaches perform equally. However, the larger the update fragment is, the more nodes need to be queried when processing single nodes.

8.3.3 Index Construction Time

When constructing an index on an existing document, the database has to determine its index entries and insert them into the index structure. To determine index entries, it can use the maintenance algorithm, which queries relevant nodes in step 2 and generates index entries in step 3. The performance of step 2 depends on how efficiently the database can query indexed nodes, i.e. on available index structures. We therefore do not include detailed performance studies for this step.

The time required to insert index entries into an index structure depends on the kind of index structure and the number of index entries. As SCIENS reuses existing index structures, we do not compare the index structures in detail. Basically, we can say that constructing an index using a hash table is more efficient than an index using a B+-tree. In case of a multidimensional index, SCIENS proposes to nest index structures or to use a KDB-tree. The nesting variant that has fewer distinct index keys in the superior index structure is most efficient, while the KDB-tree requires most time for inserting index entries.

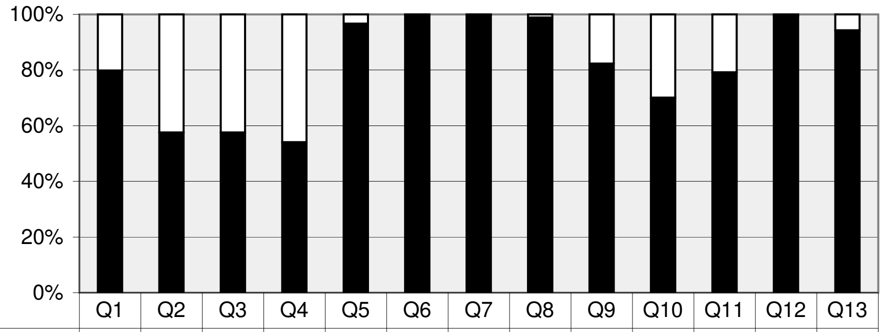
8.4 Indexing in SemCrypt

The native XML database SemCrypt (cf. Chapter 7) uses the indexing approach SCIENS to process queries and updates on XML documents. As primary index structure, SemCrypt uses a hash table on node values and a B+-tree on node labels (cf. Subsection 7.3.3). As long as no secondary indices are defined, SemCrypt has to evaluate queries based on the primary index structure. In this section, we show how secondary indices improve query and update processing in SemCrypt.

I1	KDB-tree on title values and title labelpaths
I2	B+-tree on milestone paths, nested B+-tree on date values
I3	B+-tree on resource types
I4	B+-tree on description texts

Table 8.9: Indices on project document.

As sample data set, we use the 10 MB project document and execute the queries of Table 2.2, which have been used as running example throughout the thesis. As sample indices, we have chosen a KDB-tree, a nested index and two separate B+-trees. Table 8.9 lists the sample indices, which correspond to indices I4, I8, I11 and I12 of Table 4.4.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
□ index	965	2867	2888	3569	144	217	236	1248	1286	1950	1075	290	681
■ wo index	3796	3881	3912	4196	4031	12765	10171	10167	5946	4555	4065	49826	11046

Figure 8.35: Time for evaluating sample queries with and without indices.

When using indices, SemCrypt accesses I1 to evaluate Q1-Q6 and index I2 to evaluate Q8. Q7, Q9 and Q10[2] are supported by accessing indices I1 and I2. Query Q11 accesses index I3 and queries Q12-Q13 access index I4. Figure 8.35 contrasts the time (ms) which SemCrypt requires to evaluate the sample queries with and without the sample indices. We can see that the query evaluation time strongly improves when indices are available.

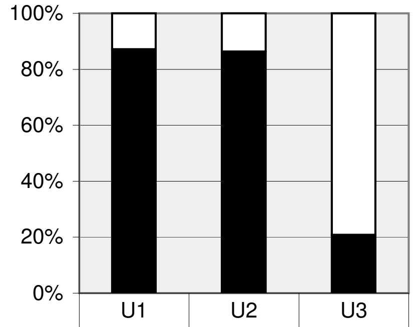
Indices do not only influence query but also update processing. To show the effect of indices on updates, we execute the updates of Table 8.10 on the sample 10 MB project document. U1 selects one resource by specifying its

[2]In the performance tests, we queried the project by its title instead of by its id.

U1	modify date of resource //project[title = 'semcrypt']/milestone[title = 'design']/resource[title = 'xml']
U2	delete resource //project[title = 'semcrypt']/milestone[title = 'design']/ resource[title = 'xml']
U3	insert 1 milestone with 10 resources into project //project[title = 'semcrypt']

Table 8.10: Updates on project document.

project, milestone and resource title and changes the date of this resource. U2 deletes a resource of a certain milestone, which it determines via the project and milestone title. Finally, U3 selects a project via its title and inserts a new milestone with ten resources into this project.



	U1	U2	U3
☐ index	591	644	15558
■ wo index	3996	4035	4078

Figure 8.36: Time for evaluating sample updates with and without indices.

Figure 8.36 depicts the time (ms) for executing the updates with and without indices. Each update first has to evaluate a query on one or more title values to determine the context of the update. In the example, index I1 can accelerate these queries. Therefore, indices even positively influence updates U1 and U2. However, the performance of update U3 decreases with indices. The reason is that the update affects several index entries, which need to be inserted into the corresponding index structures.

8.5 Summary

In this chapter, we studied the performance of the indexing approach SCIENS based on its prototypical implementation in the native XML database SemCrypt. As comparison criteria, the performance tests evaluated query evaluation speed, disk traffic, disk space, scalability, index update and index construction time.

SCIENS provides flexible and selective indices. Flexible indices enable the adaptation of indices to specific query workloads by choosing between various underlying index structures. In contrast to non-selective indices, selective indices improve query evaluation speed and reduce disk traffic and storage space. However, as they only index part of documents, they can only evaluate queries that refer to the indexed fragments. SCIENS offers the possibility to either define default secondary indices, similar to existing approaches, or to adapt indices to the query workload. As SCIENS uses balanced index structures, the indices scale well to larger documents.

The maintenance algorithm of SCIENS extracts index entries from update fragments based on index patterns. It exploits the structure of the update fragment to find all nodes that are part of index entries. In contrast to existing maintenance approaches, it need not execute source queries to retrieve the nodes that are contained in the update fragment. As source queries are expensive, the algorithm of SCIENS has better update performance, especially when updating document fragments instead of single nodes. The time for constructing an index on an existing document depends on how efficiently the database can execute queries to determine index entries as well on the specific index structure used.

The indexing approach SCIENS greatly improves query processing in SemCrypt. With regard to updates, the performance studies revealed that indices can accelerate the query which determines the context nodes for the update. However, when the update affects many index entries, updating the index structures with these index entries deteriorates update performance.

Chapter 9

Conclusion

XML databases require indices to efficiently process queries on the content and structure of documents. In addition to indices on node values, XML databases need to provide indices on names, labelpaths, paths and types of the nodes in XML documents. Indices have to be able to perform various operations on these properties to support exact, hierarchical, range and full-text queries.

Indices of (object-)relational and object-oriented databases are not adequate for the hierarchical, semi-structured data model of XML. Current XML databases only offer limited support for indexing. XML index structures presented in literature mostly index entire documents and can only support some queries. They use proprietary data structures and algorithms, which complicates providing indices for various queries.

The indexing approach SCIENS (Structure and Content Indexing with Extensible, Nestable Structures), which is presented in this thesis, provides flexible and selective indexing for XML databases. *Flexibility* refers to indexing structure- and/or content-oriented properties (values, names, labelpaths, paths and types) and to supporting various operations on these properties (exact, range, full-text and hierarchical comparisons). *Selectivity* refers to indexing frequently queried document fragments instead of entire documents only.

To represent and process indexed properties, SCIENS assigns labels to paths, labelpaths and types based on existing labeling approaches. Integrating path, labelpath and type labels into node labels improves query and update processing. With regard to processing indices, these enhanced node labels facilitate selecting appropriate indices for queries and determining which updates on documents need to be propagated to which indices.

To index arbitrary properties and support various operations on these properties, SCIENS provides extensible, nestable index structures. SCIENS proposes to index values and structural properties based on the hash table, the B+-tree and the KDB-tree. As queries on the hierarchical structure of XML documents correspond to range queries, the B+-tree and the KDB-tree only need to bind comparison operators to indexed properties. No further modification of their data structures or algorithms is necessary to support indexing content- and structure-oriented properties. Nesting index structures enables SCIENS to adapt indices to different query workloads. Compared to existing approaches, SCIENS provides indices on path and type hierarchies as well as indices that support any combination of content- and structure-oriented properties.

An XML database needs to process various indices which are defined on arbitrary properties and document fragments. The index model of SCIENS represents each index as a tree pattern. The nodes of a tree pattern have associated index variables defining indexed properties and supported operations. Physically, the index model represents each index by one - or in case of index nesting - several index structures. Based on index patterns, SCIENS presents an index framework that can process arbitrary indices of the index model without depending on their physical index structures. The framework enables SCIENS to define new indices or to implement new index structures. Index processing comprises selecting indices during query optimization, accessing indices during query execution and maintaining indices when updating documents.

To keep indices consistent with document updates, SCIENS proposes an index maintenance algorithm. As the algorithm is based on the index model, it can determine relevant updates for arbitrary indices. Basically, it consists of the following steps: (i) find embeddings of index patterns in update fragments and encode them into stacks, (ii) execute queries if nodes are required for maintenance that are not part of the update fragment, (iii) generate index entries from the embeddings and forward them to affected index structures. In contrast to existing approaches, the maintenance algorithm of SCIENS exploits schema information and node labels to determine which update nodes affect which indices. Further, it considers update fragments as a whole instead of processing each update node individually. It therefore need not query nodes that are contained in the update fragment. As updates usually refer to fragments and not to single nodes in XML, the efficiency of the proposed algorithm outperforms existing approaches.

To evaluate SCIENS, the proposed concepts have been implemented and integrated into the native, secure XML database SEMCRYPT. Integral parts of SEMCRYPT are the labeling scheme and flexible, selective indexing. While these concepts enable SEMCRYPT to process encrypted documents, they can be equally applied to unencrypted XML databases. The performance analysis demonstrate that flexibility enables SCIENS to define those indices that best match the query workload. Selectivity reduces index size and accelerates index traversal. By exploiting the structure of update fragments, the maintenance algorithm requires fewer source queries than existing approaches and can therefore process updates more efficiently.

To summarize, SCIENS can define those indices that best match the query workload and guarantees that querying and updating documents remains unaffected by specific indices used. The proposed concepts can be extended in several ways. Automatically suggesting indices for query workloads would unburden the database administrator from defining appropriate indices during database design. When optimizing queries, the index selection algorithm currently chooses the query plan with the minimum number of operators. Integrating a cost model would enable the query optimizer to choose the query plan with minimum access costs. When an update affects a large number of indices, the maintenance algorithm matches each node of the update fragment against each index pattern. Sharing commonalities between index patterns could further improve the efficiency of the maintenance algorithm.

List of Figures

List of Tables

Bibliography

[1] Configuring Database Indexes. eXist Open Source Native XML Database Documentation. `http://exist.sourceforge.net/indexing.html`.

[2] eXist Open Source Native XML Database. `http://exist.sourceforge.net/`.

[3] Java Execution Time Measurement Library. `http://jetm.void.fm/`.

[4] Oracle Berkeley DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[5] XMach-1: A Benchmark for XML Data Management. `http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html`.

[6] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science, pages 1–18. Springer, 1997.

[7] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[8] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact Labeling Schemes for Ancestor Queries. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms (SODA)*, pages 547–556. ACM/SIAM, 2001.

[9] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB)*, pages 38–49. Morgan Kaufmann, 1998.

[10] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 253–262. ACM Press, 1989.

[11] Jung-Ho Ahn, Ha-Joo Song, and Hyoung-Joo Kim. Index Set: A Practical Indexing Scheme for Object Database Systems. *Data & Knowledge Engineering*, 33(3):199–217, 2000.

[12] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient

XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 141–152. IEEE Computer Society, 2002.

[13] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 705–707. IEEE Computer Society, 2003.

[14] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree Pattern Query Minimization. *International Journal on Very Large Data Bases (VLDB Journal)*, 11(4):315–331, 2002.

[15] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and Full-text Querying for XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 83–94. ACM Press, 2004.

[16] Andrei Arion. *XML Access Modules: Towards Physical Data Independence in XML Databases.* PhD thesis, Université Paris Sud, 2007.

[17] Andrei Arion, Véronique Benzaken, and Ioana Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. In *Proceedings of the 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2005.

[18] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured Materialized Views for XML Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 87–98. ACM, 2007.

[19] Andrei Arion, Véronique Benzaken, Ioana Manolescu, Yannis Papakonstantinou, and Ravi Vijay. Algebra-based Identification of Tree Patterns in XQuery. In *Proceedings of the 7th International Conference on Flexible Query Answering Systems (FQAS)*, Lecture Notes in Computer Science, pages 13–25. Springer, 2006.

[20] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[21] Michael Barg and Raymond K. Wong. A Fast and Versatile Path Index for Querying Semi-structured Data. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 249–256. IEEE Computer Society, 2003.

[22] Michael G. Bauer, Frank Ramsak, and Rudolf Bayer. Multidimensional Mapping and Indexing of XML. In *Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz (BTW)*, pages 305–323, 2003.

[23] Rudolf Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. In *Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA)*, Lecture Notes in Computer Science, pages 198–209. Springer, 1997.

[24] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[25] Rudolf Bayer and Karl Unterauer. Prefix B-trees. *ACM Transactions on Database Systems (TODS)*, 2(1):11–26, 1977.

[26] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[27] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jerome Siméon. XML Path Language (XPath) 2.0. W3C Recommendation 23 January 2007. `http://www.w3.org/TR/2007/REC-xpath20-20070123/`.

[28] Elisa Bertino and Paola Foscoli. Index Organizations for Object-oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):193–209, 1995.

[29] Elisa Bertino and Won Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[30] Elisa Bertino, Beng Chin Ooi, Ron Sacks-Davis, Kian-Lee Tan, Justin Zobel, Boris Shidlovsky, and Barbara Catania. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.

[31] Philip Bille and Inge Li Gørtz. The Tree Inclusion Problem: In Optimal Space and Faster. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, pages 66–77. Springer, 2005.

[32] Rasa Bliujute, Simonas Saltenis, Giedrius Slivinskas, and Christian S. Jensen. Developing a DataBlade for a New Index. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 314–323. IEEE Computer Society, 1999.

[33] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérome Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007. `http://www.w3.org/TR/2007/REC-xquery-20070123`.

[34] Luc Bouganim, François Dang Ngoc, Philippe Pucheral, and Lilan Wu. Chip-Secured Data Access: Reconciling Access Rights with Data Encryption. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 1133–1136. Morgan Kaufmann, 2003.

[35] Luc Bouganim and Philippe Pucheral. Chip-Secured Data Access: Confidential Data on Untrusted Servers. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 131–142. Morgan Kaufmann, 2002.

[36] Ronald Bourret. XML Database Products: XML-Enabled Databases, 2007. `http://www.rpbourret.com/xml/ProdsXMLEnabled.htm`.

[37] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 705–716. IEEE Computer Society, 2005.

[38] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation 16 August 2006. `http://www.w3.org/TR/2006/REC-xml-20060816`.

[39] Jan-Marco Bremer and Michael Gertz. An Efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, Department of Computer Science, University of California, Davis, 2003.

[40] Jan-Marco Bremer and Michael Gertz. Integrating Document and Data Retrieval Based on XML. *International Journal on Very Large Data Bases (VLDB Journal)*, 15(1):53–83, 2006.

[41] Richard Brinkman, Ling Feng, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. Efficient Tree Search in Encrypted Data. In *Proceedings of the 2nd International Workshop on Security In Information Systems (WOSIS)*, pages 126–135. INSTICC Press, 2004.

[42] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 310–321. ACM Press, 2002.

[43] Barbara Catania, Anna Maddalena, and Athena Vakali. XML Document Indexes: A Classification. *IEEE Internet Computing*, 9(5):64–71, 2005.

[44] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. W3C Working Group Note 23 March 2007. `http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323`.

[45] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *International Journal on Very Large Data Bases (VLDB Journal)*, 11(4):354–379, 2002.

[46] Chee Yong Chan, Cheng Hian Goh, and Beng Chin Ooi. Indexing OODB Instances Based on Access Proximity. In *Proceedings of the 13th Internal Conference on Data Engineering (ICDE)*, pages 14–21. IEEE Computer Society, 1997.

[47] Akmal B. Chaudhri, Roberto Zicari, and Awais Rashid. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison-Wesley Longman Publishing, 2003.

[48] Li Chen and Elke A. Rundensteiner. Aggregate Path Index for Incremental Web View Maintenance. In *Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, pages 231–238, 2000.

[49] Qun Chen, Andrew Lim, Kian Win Ong, and Jiqing Tang. Indexing XML Documents for XPath Query Processing in External Memory. *Data & Knowledge Engineering*, 59(3):681–699, 2006.

[50] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 455–466. ACM, 2005.

[51] Yan Chen, Sanjay Kumar Madria, Kalpdrum Passi, and Sourav S. Bhowmick. Efficient Processing of XPath Queries Using Indexes. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA)*, Lecture Notes in Computer Science, pages 721–730. Springer, 2002.

[52] Yangjun Chen and Karl Aberer. Layered Index Structures in Document Database Systems. In *Proceedings of the 7th International ACM Conference on Information and Knowledge Management (CIKM)*, pages 406–413. ACM, 1998.

[53] Yangjun Chen and Yibin Chen. A New Tree Inclusion Algorithm. *Information Processing Letters*, 98(6):253–262, 2006.

[54] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 237–248. Morgan Kaufmann, 2003.

[55] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274. Morgan Kaufmann, 2002.

[56] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2002.

[57] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 271–281. ACM, 2002.

[58] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[59] 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 (VLDB)*, pages 341–350. Morgan Kaufmann, 2001.

[60] John Cowan and Richard Tobin. XML Information Set (Second Edition). W3C Recommendation 4 February 2004. `http://www.w3.org/TR/2004/REC-xml-infoset-20040204`.

[61] Ernesto Damiani, Sabrina De Capitani di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 93–102. ACM, 2003.

[62] Tran Khanh Dang. Extreme Security Protocols for Outsourcing Database Services. In *Proceedings of the 6th International Conference on Information Integration and Web-based Applications Services (iiWAS)*. Austrian Computer Society, 2004.

[63] Bruno H. M. Denuit and Stefano Stefani. Secondary Index and Indexed View Maintenance for Updates to Complex Types. United States Patent 20060015490, 2006.

[64] Stefan Deßloch, Weidong Chen, Jyh-Herng Chow, You-Chin Fuh, Jean Grandbois, Michelle Jou, Nelson Mendonça Mattos, Raiko Nitzsche, Brian T. Tran, and Yun Wang. Extensible Indexing Support in DB2 Universal Database. In *Component Database Systems*, pages 105–138. Morgan Kaufmann, 2001.

[65] Melvil Dewey. A Classification and Subject Index for Cataloguing and Arranging the Books and Pamphlets of a Library, 1876.

[66] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[67] Paul F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 122–127. ACM Press, 1982.

[68] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-Sensitive View Maintenance of Materialized XQuery Views. In *Proceedings of the 22nd International Conference on Conceptual Modeling*, Lecture Notes in Computer Science, pages 144–157. Springer, 2003.

[69] Klaus R. Dittrich and Andreas Geppert. *Component Database Systems*. Morgan Kaufmann, 2001.

[70] Xin Dong and Alon Y. Halevy. Indexing Dataspaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 43–54. ACM Press, 2007.

[71] Walter Dorninger. Securing Remote Data Stores - Design and Implementation of an Encrypted Data Store. Master's thesis, Johannes Kepler University Linz, Department of Business Informatics - Data & Knowledge Engineering, 2005.

[72] Maggie Duong and Yanchun Zhang. LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In *Proceedings of the 16th Australasian Database Conference (ADC)*, CRPIT, pages 185–193. Australian Computer Society, 2005.

[73] Takeharu Eda, Yasushi Sakurai, Toshiyuki Amagasa, Masatoshi Yoshikawa, Shunsuke Uemura, and Takashi Honishi. Dynamic Range Labeling for XML Trees. In *Current Trends in Database Technology - EDBT Workshops*, Lecture Notes in Computer Science, pages 230–239, 2004.

[74] Maged El-Sayed, Ling Wang, Luping Ding, and Elke Rundensteiner. An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views. In *Proceedings of the 4th ACM CIKM International Workshop on Web Information and Data Management (WIDM)*, pages 88–91. ACM, 2002.

[75] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.

[76] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation 23 Januaray 2007. `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123`.

[77] Mary F. Fernández, Jérôme Siméon, and Philip Wadler. An Algebra for XML Query. In *Proceedings of the 20th Conference Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science, pages 11–45. Springer, 2000.

[78] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the Minimization of XPath Queries. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 153–164. Morgan Kaufmann, 2003.

[79] Kenny C.K. Fong. Potential Security Holes in Hacigümüs' Scheme of Executing SQL over Encrypted Data. `http://www.cs.siu.edu/~kfong/research/database.pdf`.

[80] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: An Algebra for XML Query Optimization. In *Proceedings of the 13th Australasian Conference on Database Technologies (ADC)*, CRPIT, pages 49–56. Australian Computer Society, 2002.

[81] Kei Fujimoto, Dao Dinh Kha, Masatoshi Yoshikawa, and Toshiyuki Amagasa. A Mapping Scheme of XML Documents into Relational Databases Using Schema-based Path Identifiers. In *Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration (WIRI)*, pages 82–90. IEEE Computer Society, 2005.

[82] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[83] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[84] Charles F. Goldfarb and Paul Prescod. *Charles F. Goldfarb's XML Handbook*. Prentice Hall, 2003.

[85] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pages 436–445. Morgan Kaufmann, 1997.

[86] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems (TODS)*, 30(2):444–491, 2005.

[87] Katharina Grün. A Generic Framework for Querying and Updating Secondary XML Index Structures. In *Proceedings of the SIGMOD Ph.D. Workshop on Innovative Database Research (IDAR)*, pages 27–32, 2007.

[88] Katharina Grün, Michael Karlinger, and Michael Schrefl. Schema-aware Labelling of XML Documents for Efficient Query and Update Processing in SemCrypt. *Computer Systems Science and Engineering*, 21(1):65–82, 2006.

[89] Katharina Grün and Michael Schrefl. Exploiting the Structure of Update Fragments for Efficient XML Index Maintenance. In *Proceedings of the Joint 9th Asia-Pacific Web Conference and the 8th International Conference on Web-Age Information Management (APWeb/WAIM)*, Lecture Notes in Computer Science, pages 471–478. Springer, 2007.

[90] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1):91–131, 2004.

[91] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data(base) Engineering Bulletin*, 18(2):3–18, 1995.

[92] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 216–227. ACM Press, 2002.

[93] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed Mode XML Query Processing. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 225–236. Morgan Kaufmann, 2003.

[94] Beda Christoph Hammerschmidt. *KeyX: Selective Key-Oriented Indexing in Native XML-Databases*. PhD thesis, University of Lübeck, 2005.

[95] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. The Index Update Problem for XML Data in XDBMS. In *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS)*, pages 27–34, 2005.

[96] Zhongming Han, Congting Xi, and Jiajin Le. Efficiently Coding and Indexing XML Document. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, Lecture Notes in Computer Science, pages 138–150. Springer, 2005.

[97] Theo Härder, Michael Haustein, Christian Mathis, and Markus Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, 2007.

[98] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001.

[99] Ragib Hasan, Marianne Winslett, and Radu Sion. Requirements of Secure Storage Systems for Healthcare Records. In *Proceedings of the 4th VLDB Workshop on Secure Data Management (SDM)*, Lecture Notes in Computer Science, pages 174–180. Springer, 2007.

[100] Hao He and Jun Yang. Multiresolution Indexing of XML for Frequent Queries. In *Proceedings of the 20th International Conference on Data Engineering (VLDB)*, pages 683–694. IEEE Computer Society, 2004.

[101] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of 21th International Conference on Very Large Data Bases (VLDB)*, pages 562–573. Morgan Kaufmann, 1995.

[102] Andreas Henrich. The Update of Index Structures in Object-oriented DBMS. In *Proceedings of the 6th International Conference on Information and Knowledge Management (CIKM)*, pages 136–143. ACM, 1997.

[103] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[104] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation 7 April 2004. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`.

[105] Walter L. Hürsch. Should Superclasses be Abstract? In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 12–31. Springer, 1994.

[106] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A Native XML Database. *International Journal on Very Large Data Bases (VLDB Journal)*, 11(4):274–291, 2002.

[107] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Proceedings of the 8th International Workshop on Database Programming Languages (DBPL)*, Lecture Notes in Computer Science, pages 149–164. Springer, 2002.

[108] Ravi Chandra Jammalamadaka and Sharad Mehrotra. Querying Encrypted XML Documents. In *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS)*, pages 129–136. IEEE Computer Society, 2006.

[109] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 253–264. IEEE Computer Society, 2003.

[110] Haim Kaplan, Tova Milo, and Ronen Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 954–963. ACM/SIAM, 2002.

[111] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–144. ACM Press, 2002.

[112] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for Structure Indexes. *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 239–250, 2002.

[113] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-structured Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 129–140. IEEE Computer Society, 2002.

[114] Dao Dinh Kha and Masatoshi Yoshikawa. XML Query Processing Using a Schema-Based Numbering Scheme. In *Proceedings of the 2nd International XML Database Symposium on Database and XML Technologies (XSym)*, Lecture Notes in Computer Science, pages 21–34. Springer, 2004.

[115] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. A Structural Numbering Scheme for XML Data. In *XML-Based Data Management and Multimedia Engineering - EDBT Workshops*, Lecture Notes in Computer Science, pages 91–108. Springer, 2002.

[116] Aye Aye Khaing and Ni Lar Thein. A Persistent Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 498–501. IEEE Computer Society Washington, DC, USA, 2006.

[117] Christoph Kilger and Guido Moerkotte. Indexing Multiple Sets. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 180–191. Morgan Kaufmann, 1994.

[118] Won Kim, Kyung-Chang Kim, and Alfred G. Dale. *Indexing Techniques for Object-oriented Databases.* ACM Press, 1989.

[119] Lau Hoi Kit and Vincent Ng. Enumerating XML Data for Dynamic Updating. In *Proceedings of the 16th Australasian Database Conference (ADC)*, CRPIT, pages 75–84. Australian Computer Society, 2005.

[120] Hye-Kyeong Ko and SangKeun Lee. An Efficient Scheme to Completely Avoid Re-labeling in XML Updates. In *Proceedings of the 7th International Conference on Web Information Systems Engineering (WISE)*, Lecture Notes in Computer Science, pages 259–264. Springer, 2006.

[121] Evangelos Kotsakis. Structured Information Retrieval in XML Documents. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pages 663–667. ACM Press, 2002.

[122] Michal Kratky and Radim Baca. A Cost-based Join Selection for XML Twig Content-based Queries. In *Proceedings of the 3rd International Workshop on Database Technologies for Handling XML Information on the Web (DataX) - EDBT Workshops.* ACM Press, 2008.

[123] April Kwong and Michael Gertz. Schema-based Optimization of XPath Expressions. Technical report, University of California, 2002.

[124] Wilburt Labio, Dallan Quass, and Brad Adelberg. Physical Database Design for Data Warehouses. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 277–288. IEEE Computer Society, 1997.

[125] Peter Lasinger. Indexing Encrypted XML Documents in the SemCrypt Database Management System. Master's thesis, Johannes Kepler University Linz, Department of Business Informatics - Data & Knowledge Engineering, 2006.

[126] Jong-Hak Lee, Kyu-Young Whang, Wook-Shin Han, Wan-Sup Cho, and Il-Yeol Song. 2D-CHI: A Tunable Two-Dimensional Class Hierarchy Index for Object-Oriented Databases. In *Proceedings of the 24th International Computer Software and Applications Conference (COMPSAC)*, pages 598–607. IEEE Computer Society, 2000.

[127] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index Structures for Structured Documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 91–99. ACM, 1996.

[128] Krishna P. Leela and Jayant R. Haritsa. Schema-conscious XML Indexing. *Information Systems*, 32(2):344–364, 2007.

[129] Yves Lépouchard, John L. Pfaltz, and Ratko Orlandic. Performance of KDB-trees with Query-based Splitting. In *Proceedings of the International Symposium on Information Technology (ITCC)*, pages 218–223. IEEE Computer Society, 2002.

[130] Changqing Li and Tok Wang Ling. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, Lecture Notes in Computer Science, pages 125–137. Springer, 2005.

[131] Changqing Li and Tok Wang Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proceedings of the 14th ACM CIKM International Conference on Information and Knowledge Management (CIKM)*, pages 501–508. ACM Press, 2005.

[132] Changqing Li, Tok Wang Ling, and Min Hu. Efficient Processing of Updates in Dynamic XML Data. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 13. IEEE Computer Society, 2006.

[133] Changqing Li, Tok Wang Ling, and Min Hu. Efficient Updates in Dynamic XML Data: From Binary String to Quaternary String. *International Journal on Very Large Data Bases (VLDB Journal)*, 17(3):573–601, 2008.

[134] Hanyu Li, Mong-Li Lee, and Wynne Hsu. A Path-Based Labeling Scheme for Efficient Structural Join. In *Proceedings of the 3rd International XML Database Symposium on Database and XML Technologies (XSym)*, Lecture Notes in Computer Science, pages 34–48. Springer, 2005.

[135] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370. Morgan Kaufmann, 2001.

[136] Hartmut Liefke and Susan B. Davidson. View Maintenance for Hierarchical Semistructured Data. In *Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, Lecture Notes in Computer Science, pages 114–125. Springer, 2000.

[137] Hung-Yi Lin and Po-Whei Huang. Perfect KDB-Tree: A Compact KDB-Tree Structure for Indexing Multidimensional Data. In *Proceedings of the 3rd International Conference on Information Technology and Applications (ICITA)*, pages 411–414. IEEE Computer Society, 2005.

[138] Ping Lin and K. Selçuk Candan. Secure and Privacy Preserving Outsourcing of Tree Structured Data. In *Proceedings of the VLDB Workshop on Secure Data Management (SDM)*, Lecture Notes in Computer Science, pages 1–17. Springer, 2004.

[139] Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, pages 212–223. IEEE Computer Society, 1980.

[140] Peter C. Lockemann and Klaus R. Dittrich. *Architektur von Datenbanksystemen*. dpunkt, 2004.

[141] David B. Lomet and Betty Salzberg. The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems (TODS)*, 15(4):625–658, 1990.

[142] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 193–204. ACM, 2005.

[143] Bertram Ludäscher, Ilkay Altintas, and Amarnath Gupta. Time to Leave the Trees: From Syntactic to Conceptual Querying of XML. In *XML-Based Data Management and Multimedia Engineering - EDBT Workshops*, Lecture Notes in Computer Science, pages 148–168. Springer, 2002.

[144] David Maier and Jacob Stein. Indexing in an Object-oriented Database. In *Proceedings of the IEEE Workshop on Object-Oriented DBMSs*, 1986.

[145] Ioana Manolescu and Yannis Papakonstantinou. A Unified Tuple-based Algebra for XQuery. Technical report, Gemo, 2005. `ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-434.pdf`.

[146] Norman May, Matthias Brantner, Alexander Böhm 0002, Carl-Christian Kanne, and Guido Moerkotte. Index vs. Navigation in XPath Evaluation. In *Proceedings of the 4th International XML Database Symposium on Database and XML Technologies (XSym)*, Lecture Notes in Computer Science, pages 16–30. Springer, 2006.

[147] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[148] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, Lecture Notes in Computer Science, pages 169–183. Springer, 2002.

[149] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 65–76. ACM, 2002.

[150] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science, pages 277–295. Springer, 1999.

[151] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[152] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[153] Thomas A. Mueck and Martin L. Polaschek. A Configurable Type Hierarchy Index for OODB. *International Journal on Very Large Data Bases (VLDB Journal)*, 6(4):312–332, 1997.

[154] 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 (ICDT)*, Lecture Notes in Computer Science, pages 312–326. Springer, 2003.

[155] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[156] Shirish K. Nilekar. Self Maintenance of Materialized XQuery Views via Query Containment and Re-writing. Master's thesis, Worcester Polytechnic Institute, 2006.

[157] Patrick E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59. Springer, 1989.

[158] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 903–908. ACM, 2004.

[159] ongjian Fu, Jui-Che Teng, and S. R. Subramanya. Node Splitting Algorithms in Tree-structured High-dimensional Indexes for Similarity Search. In *Proceedings of the ACM Symposium on Applied Computing*, pages 766–770. ACM Press, 2002.

[160] Beng Chin Ooi, Jiawei Han, Hongjun Lu, and Kian-Lee Tan. Index Nesting - An Efficient Approach to Indexing in Object-oriented Databases. *International Journal on Very Large Data Bases (VLDB Journal)*, 5(3):215–228, 1996.

[161] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 251–260. IEEE Computer Society, 1995.

[162] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An Adaptive Structural Summary for Graph-structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 134–144. ACM, 2003.

[163] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. W3C Recommendation 24 December 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`.

[164] Raghu Ramakrishnan and Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 2003.

[165] Prakash Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 299–309. ACM Press, 2002.

[166] Sridhar Ramaswamy and Paris C. Kanellakis. OODB indexing by Class-Division. *ACM SIGMOD Record*, 24(2):139–150, 1995.

[167] Praveen Rao and Bongki Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 288–300. IEEE Computer Society, 2004.

[168] Flavio Rizzolo and Alberto O. Mendelzon. Indexing XML Data with ToXin. In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB)*, pages 49–54, 2001.

[169] John T. Robinson. The KDB-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18. ACM Press, 1981.

[170] Ron Sacks-Davis, Tuong Dao, James A. Thom, and Justin Zobel. Indexing Documents for Queries on Structure, Content and Attributes. In *Proceedings of the International Symposium on Digital Media Information Base (DMIB)*, pages 236–245, 1997.

[171] Daniel Sanders. System and Method for Efficient Maintenance of Indexes for XML Files. United States Patent 20070220420.

[172] Mario Schkolnick and Paolo Tiberio. Estimating the Cost of Updates in a Relational Database. *ACM Transactions on Database Systems*, 10(2):163–179, 1985.

[173] 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. Morgan Kaufmann, 2002.

[174] Karsten Schmidt and Theo Härder. An Adaptive Storage Manager for XML Documents. In *Datenbanksysteme in Business, Technologie und Web, BTW Workshop Proceedings*, pages 317–328. Verlagshaus Mainz, 2007.

[175] Harald Schöning. Tamino - A DBMS Designed for XML. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 149–154. IEEE Computer Society, 2001.

[176] Michael Schrefl, Jürgen Dorn, and Katharina Grün. SemCrypt - Ensuring Privacy of Electronic Documents through Semantic-based Encrypted Query Processing. In *Proceedings of the International Workshop on Privacy Data Management (PDM), ICDE Workshops*, page 1191. IEEE Computer Society Press, 2005.

[177] Thomas Schwentick. XPath Query Containment. *ACM SIGMOD Record*, 33(1):101–109, 2004.

[178] Bernhard Seeger and Hans-Peter Kriegel. The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pages 590–601. Morgan Kaufmann, 1990.

[179] Toshiyuki Shimizu and Masatoshi Yoshikawa. Full-Text and Structural Indexing of XML Documents on B+-Tree. *IEICE Transactions on Information and Systems*, (1):237–247, 2006.

[180] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. In *Proceedings of the 3rd ACM Conference on Digital Libraries*, pages 235–243. ACM, 1998.

[181] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proceedings of the 21st International Conference on Data Engineering, (ICDE)*, pages 285–296. IEEE Computer Society, 2005.

[182] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[183] B. Sreenath and S. Seshadri. The hcC-tree: An Efficient Index Structure for Object Oriented Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 203–213. Morgan Kaufmann, 1994.

[184] Jagannathan Srinivasan, Ravi Murthy, Seema Sundara, Nipun Agarwal, and Samuel DeFazio. Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 91–100. IEEE Computer Society, 2000.

[185] Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the 2nd International Conference on Data Engineering (ICDE)*, pages 262–269. IEEE Computer Society, 1986.

[186] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215. ACM, 2002.

[187] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. W3C Recommendation 28 October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028`.

[188] Ashish Thusoo, Sivasankaran Chandrasekar, Ravi Murthy, Nipun Agarwal, Eric Sedlar, Sreedhar Mukkamall, and Reema Koo. Efficient Queribility and Manageability of an XML Index with Path Subsetting. United States Patent 20050228791, 2005.

[189] Vijay Vaishnavi and Bill Kuechler. Design Research in Information Systems, 2004. `http://www.isworld.org/Researchdesign/drisISworld.htm`.

[190] Athena Vakali, Barbara Catania, and Anna Maddalena. XML Data Stores: Emerging Practices. *IEEE Internet Computing*, 9(2):62–69, 2005.

[191] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[192] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 110–121. ACM Press, 2003.

[193] Hui Wang and Laks V. S. Lakshmanan. Efficient Secure Query Evaluation over Encrypted XML Databases. *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 127–138, 2006.

[194] J. Wang, X. Meng, and S. Wang. Integrating Path Index with Value Index for XML Data. In *Proceedings of the 5th Asian-Pacific Web Conference on Web Technologies and Applications (APWeb)*, Lecture Notes in Computer Science, pages 95–100. Springer, 2003.

[195] Felix Weigel, Holger Meuss, François Bry, and Klaus U. Schulz. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Proceedings of the 26th European Conference on IR Research, Advances in Information Retrieval (ECIR)*, Lecture Notes in Computer Science, pages 378–393. Springer, 2004.

[196] 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. ACM Press, 2004.

[197] Lauren Weinstein. Inside Risks 164. *Communications of the ACM (CACM)*, 47(2), 2004.

[198] Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 66–78. IEEE Computer Society, 2004.

[199] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of 20th International Conference on Data Engineering (ICDE)*, pages 621–632. IEEE Computer Society, 2004.

[200] Jingtao Yao and Ming Zhang. A Fast Tree Pattern Matching Algorithm for XML Query. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 235–241. IEEE Computer Society, 2004.

[201] Ke Yi, Hao He, Ioana Stanoi, and Jun Yang. Incremental Maintenance of XML Structural Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 491–502. ACM, 2004.

[202] Jong P. Yoon, Vijay Raghavan, Venu Chakilam, and Larry Kerschberg. BitCube: A Three-Dimensional Bitmap Indexing for XML Documents. *Journal of Intelligent Information Systems*, 17(2):241–254, 2001.

[203] Byunggu Yu, Ratko Orlandic, Thomas Bailey, and Jothi Somavaram. KDBKD-Tree: A Compact KDB-Tree Structure for Indexing Multidimensional Data. In *Proceedings of the International Symposium on Information Technology (ITCC)*, pages 676–680. IEEE Computer Society, 2003.

[204] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng, and Hongjun Lu. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web*, 8(1):5–26, 2005.

[205] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 289–300. ACM, 2005.

[206] Xin Zhang and Elke A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, 2002.

[207] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. *ACM SIGMOD Record*, 25(3):10–15, 1996.