

A Generic Framework for Querying and Updating Secondary XML Index Structures *

Katharina Grün
Department of Business Informatics - Data & Knowledge Engineering
Johannes Kepler University Linz, Austria
gruen@dke.uni-linz.ac.at

ABSTRACT

To cope with the increasing number and size of XML documents, XML databases provide index structures to accelerate queries on the content and structure of documents. To adapt indices to the query workload, XML databases require various secondary index structures. This paper presents a generic index framework called SCIENS (Structure and Content Indexing with Extensible, Nestable Structures). In contrast to existing work on XML indexing, this framework can integrate arbitrary index structures and adapt them to different query requirements. It supports defining, accessing and maintaining indices without affecting query and update processing. By offering a great flexibility of what to index, the framework allows for processing queries more efficiently.

1. INTRODUCTION

Databases rely on index structures to efficiently process queries without scanning large amount of data. Besides providing one primary index structure, databases support defining secondary index structures on specific data fragments to efficiently answer frequently issued queries.

As the number and size of XML documents increase, XML databases need to provide a framework that supports indexing both the content and the structure of frequently queried document fragments. Developing such a framework poses the following challenges: (i) Which index structures are necessary to support queries on arbitrary properties of XML documents? (ii) How can secondary index structures be integrated into a common framework that allows for adapting index structures to different requirements without impacting query and update processing? (iii) How can index structures that are defined on arbitrary document fragments be maintained when updating documents?

An XML document can be represented as a tree of element, attribute and text nodes. Element and attribute nodes have a name, attribute and text nodes have associ-

*This work was supported by FIT-IT under grant 809262/9315-KA/HN.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Proceedings of SIGMOD2007 Ph.D. Workshop on Innovative Database Research 2007(IDAR2007), June 10, 2007, Beijing, China.

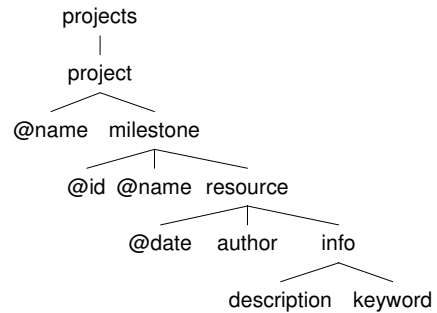


Figure 1: Sample schema

ated a value. The sequence of nodes from the root of the document to a specific node is the path of the node. The node names of the path form the labelpath of the node. The tree structure of documents can be viewed as a hierarchy, whereby the labelpaths represent the schema hierarchy and the paths the document hierarchy. Nodes also have a type, which can be part of a type hierarchy defined in a schema document. Documents can be organized into collections, which may be grouped hierarchically.

Tailoring index structures to the query workload is a challenging task in XML databases as queries may constrain not only node values, but also the schema, document, type or collection hierarchy. Concerning these hierarchies, queries may either select the entire hierarchy or a specific part of a hierarchy. All kinds of queries should be supported by secondary XML index structures.

Example 1. A sample use case is a database application that manages project resources. Figure 1 depicts a simplified schema that groups resources into project milestones. Assume that there are different types of resources, e.g. financial and technical reports. Table 1 lists sample queries using the XPath 2.0 syntax [1]. The first query selects all resources with a particular date. The second and third query limit the search to certain project (milestones) and thus only consider a specific part of the document hierarchy. Queries 4 and 5 retrieve the authors of those resources whose info element contains a word or word prefix using a regular expression. Thereby, the fifth query only looks at keywords and therefore further constrains the schema hierarchy compared to the fourth query. The sixth query constrains the types of resources by selecting reports with a particular author.

Index structures that have specifically been developed for XML [5] mainly focus on primary index structures. They

Table 1: Sample queries

Q1	//resource[@date \geq '2005-01-01']
Q2	//project[@name='SemCrypt']/milestone[@name='design']/resource[@date \geq '2007-01-01']
Q3	//project[@name='SemCrypt']//resource[@date \leq '2006-12-31']
Q4	//resource[info/fn:matches(*,'\\bindex\b')]/author
Q5	//resource[info/fn:matches(keyword,'\\bdata.*')]/author
Q6	//element(resource,Report)[author='Smith']

present proprietary structures that are based on different models and are tightly integrated with query and update processing. While the primary index structure covers whole documents and can only support certain queries best, secondary index structures should accelerate frequently issued queries. As such, they should offer flexibility of what to index and adaptability to various kinds of queries. Compared to the primary index structure, secondary index structures must not be tightly coupled with query and update processing as they can be defined on arbitrary fragments and index arbitrary properties of XML documents. The use of proprietary structures would make it difficult to adapt secondary indices to the query workload as each query kind would require a different index structure. Instead, an XML database should select a small set of index structures and adapt them to index various properties of XML documents. As there still does not exist an index framework that handles these requirements, current XML databases only offer limited support for secondary index structures.

In this paper, we present the SCIENS framework, which has been developed within a research project called *SemCrypt* [20]. SemCrypt is a secure XML database that aims at querying and updating outsourced, encrypted XML documents with the help of a schema-aware labeling scheme [11] and index structures that match the query workload. The main contributions of this paper are:

- Based on frequently issued queries, we select a small set of index structures and describe how to extend them to index various properties of XML documents. We further propose the concept of index nesting to adapt these index structures to hierarchical queries.
- We present a framework for processing indices based on an index model that defines which properties of the XML data model can be indexed as well as operations on these properties. The framework provides the basis for integrating, extending and nesting index structures without affecting query and update processing tasks.
- To determine which document updates affect which index structures, we propose a generic index maintenance algorithm that generates index entries based on index definitions and update fragments and that is independent of specific index structures used.

Note that neither selecting an index for a particular query nor suggesting which indices to build for a query workload is subject of this paper. Regarding these research issues, we refer the reader to [2] and [13], respectively.

This paper is organized as follows. Section 2 reviews related work and connects it to our setting. Based on current shortcomings, Section 3 describes the main ideas of the SCIENS framework. Section 4 presents performance studies and Section 5 outlines the research methodology. Section 6 concludes the paper.

2. RELATED WORK

Indexing plays an important role in database research. Widespread index structures include hash tables, B-trees and its variations, and inverted files. Various native XML databases [6] allow for indexing the values of nodes or paths having the same names. Supported operations on values include exact match, range queries and word prefix search. However, indexing in XML databases is still limited. Neither do they support multidimensional nor hierarchical indexing as they cannot index nodes along multiple axes. Integrating these concepts into current XML databases is difficult, as they do not clearly distinguish between query/update processing and indexing tasks.

A multitude of specific index structures has been proposed for indexing XML documents, which enhance traditional concepts, such as inverted files or B-trees, with structural information. They can broadly be classified into two categories, depending on whether they only index the structure of documents or combine structural and value indices.

Several approaches exist for indexing the document structure: (i) Summarize the paths which exist in a document and associate with each labelpath the nodes which can be reached along the labelpath (e.g. DataGuide [10], Index Fabric [8], F&B-Index [22]). (ii) Build separate indices for each node name that allow for evaluating structural relationships (e.g. XL+-tree [7]). (iii) Transform the document and the query into sequences and process queries based on subsequence matching (e.g. Prix [18]).

Indices on the structure do not directly support constraining node values. To query the document content without scanning all nodes of one labelpath, the following approaches exist: (i) Combine path indices and value indices by either grouping the nodes returned by a value index according to their labelpath or by building one value index for each labelpath (e.g. CADG [23]). (ii) Build indices on values and node names and process queries by structural joins (e.g. XISS [16]). (iii) Filter the structure and the content using a multidimensional index structure (e.g. BitCube [24]).

Indexing the structure of XML documents is related to indexing aggregation graphs and inheritance hierarchies in object-oriented databases [4]. Indices on aggregation graphs allow for indexing objects on values of nested objects. To query (partial) class hierarchies or single classes, object-oriented indices either group the nodes returned by a value index according to their class or build one value index for each class (e.g. CH and SC index [15]). The MT-index [17] maps the class hierarchy to a linear order and uses a multidimensional index structure in which the class hierarchy represents one dimension. In the same way as object-oriented databases index paths and inheritance hierarchies, XML databases should provide indices that support navigation along paths as well as queries on the schema, document, type and collection hierarchy.

One possibility to index several properties is to use multidimensional index structures [9], which enable exact and range queries on multiple dimensions. However, these index structures only consider values and do not address indexing structural aspects.

Extensible indexing refers to using one index structure for different data types. The Generalized Search Tree (GiST) [14] is a tree-structured index that can be adapted to handle different data types. Several object-relational databases offer similar concepts in order to extend the database with user-defined types and operations on these types. Oracle Data Cartridges [21], for example, allow for adapting index structures to support new data types. The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations and searching the index during query processing. Applying the concept of extensible indexing to XML provides the possibility to index different properties, representing various data types, with one index structure.

Determining which index best suits a particular query is an essential task when offering flexible index structures. Arion et al. [2] propose how to select access modules to optimize queries. As secondary index structures constitute such access modules, we do not further consider index selection. Updating index structures defined on arbitrary document fragments has widely been neglected so far. The approach of Hammerschmidt [13] does not exploit the structural relationships between nodes of update fragments, resulting in a poor performance when updating document fragments [12].

3. CONCEPTS

In the following, we present the main ideas of the SCIENS framework. First we select various index structures to index the content and structure of XML documents and then describe how to integrate them into a common framework based on an index model. We finally outline a new maintenance algorithm for updating arbitrary index structures.

3.1 Index Structures

Based on several use cases, we identified the following set of frequently issued queries to be supported by index structures. Concerning the document content, queries retrieve nodes whose value exactly matches a specific value, is within a certain value range or contains a specific word or word prefix. With respect to the document structure, queries navigate along paths to select nodes with a certain labelpath or node name. Structural queries may also reflect the hierarchical nature of XML documents, i.e. the schema, document, type or collection hierarchy. They either select the entire hierarchy or only a specific part of a hierarchy.

The main characteristic of queries on XML documents is that they not only constrain the content but also the structure of documents. If an XML database only provides value indices, the indices not only become very large, but also return a large number of nodes that need to be filtered according to the structural search conditions afterwards. It is therefore important to allow for combined value and structure indexing, i.e. to also support navigation along paths as well as hierarchically querying the structure of documents.

Example 2. Looking at Table 1, an index on the date best supports the first query as it avoids scanning all dates of resources. If we only index the date of resources, the sec-

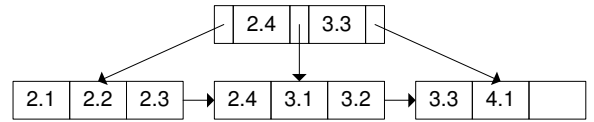


Figure 2: Sample B-tree on the document hierarchy

ond and third query are evaluated either by first retrieving all resources with the requested date and then filtering the ones that match the specified project (milestone), or without accessing an index. An index on both the date and the document hierarchy of projects and milestones better supports these queries as it only returns the requested resources. If we want to build an index for the fourth and fifth query, indexing the value and labelpath of info elements is most appropriate. Query 6 can be accelerated by creating an index on the type of resource and the value of author elements.

Widespread index structures for content indexing are hash tables and B-trees. While hash tables allow for searching nodes with a specific value, B-trees support range queries. A special kind of B-tree is the prefix B-tree [3], which uses minimal prefixes as keys. Indexing the individual words of a node value with a prefix B-tree enables word and word prefix search.

Example 3. Regarding the first query of Table 1, we can use a B-tree which maps values of date nodes to corresponding resources. In fact, this index also constrains the structure, but it does not allow for specifying search conditions on the structure.

Various index structures have been proposed to index the XML document structure (cf. Section 2). A hash table is appropriate to retrieve all nodes with a specific labelpath or node name. Only few indexing approaches on the hierarchical structure are optimized for disk storage [8, 22]. Instead of integrating proprietary index structures, we propose to use B-trees for indexing hierarchies as they are well-established and widely applicable. As all hierarchies in XML have the form of trees, the preorder traversal of the tree establishes a linear order between its nodes, which can be used for comparisons within the B-tree. More precisely, the order between the nodes of the hierarchies can be compared as follows. Labeling schemes (e.g. [11]) associate unique labels with nodes which allow for determining the document order between nodes. Regarding the schema and type hierarchy, the distinct labelpaths and types can equally be assigned unique labels [11]. In the same way, collections can be identified by unique labels. By assigning labels in preorder, hierarchical queries simply correspond to range queries. Only the operator which guides the search through the tree needs to be adapted such that a link is also followed if the search key is part of the corresponding hierarchy.

Example 4. Figure 2 shows part of a sample B-tree on the milestone hierarchy. It indexes the labels of milestone nodes, which are generated by a prefix labeling scheme [11]. The B-tree does not only allow for retrieving all nodes (e.g. resources) that belong to a particular milestone, but also those of a certain project. For example, assume that we are looking for all nodes of the project identified by the prefix label 3. Comparison with label 3.3 yields that this label is part

of the specified hierarchy and that the second leaf page contains requested nodes. As the last node of this leaf page is still part of the hierarchy, the third leaf page also needs to be visited. Search stops at node 4.1 which belongs to a different hierarchy, i.e. the corresponding milestone is part of another project. The index enables to selectively retrieve only those nodes that are part of a certain hierarchy, which has been selected in advance.

So far, we have only considered the one-dimensional case. Indexing multiple properties in one index is more compact and efficient than scanning several indices for these properties. Object-oriented and XML indices either group values according to the type hierarchy and the document structure, respectively, or vice versa. This allows for favoring queries referring to either a specific part of the hierarchy or the entire hierarchy. Multidimensional index structures index several dimensions without favoring a specific dimension. To support both approaches, we propose two concepts: index nesting and an extensible multidimensional index.

Index nesting refers to placing one index structure beneath another index structure. It allows for grouping dimensions without developing proprietary index structures. Dependent on which dimension represents the highest level of the nesting hierarchy, different queries are better supported. For example, if a structure index on a hierarchy 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.

Example 5. To evaluate the second and third query of Table 1, we can nest a B-tree on the date beneath a B-tree on the document hierarchy. This results in one B-tree on the date for each milestone. If we nested the B-trees vice versa, the index would favor queries which are mostly not limited to certain milestones or projects.

Index nesting is not appropriate for dimensions that do not hierarchically relate to each other or when queries refer to various hierarchical levels. To index arbitrary dimensions without favoring a particular dimension, we propose an extensible multidimensional index. As we have already shown how to index a hierarchy with a B-tree, we propose to use a tree-structured multidimensional index, e.g. a KDB-tree [19, 25]. By using prefixes as keys, the KDB-tree can not only support range queries, but also word prefix search. To index hierarchies, it can integrate the same concepts as presented for B-trees. Note that the KDB-tree can be nested in the same way as the B-tree, which allows for adapting indices to the query workload in a very flexible way.

Example 6. We can also build a multidimensional index on the date and the milestone hierarchy for Example 5 to avoid favoring particular hierarchical queries.

A hash table, a prefix B-tree and a multidimensional index such as a KDB-tree are sufficient to index the content and structure of XML documents. The concept of index nesting enables to group index structures and support either queries on the entire hierarchy or on a specific part of the hierarchy. Note that a B-tree can substitute a hash table as well as a multidimensional index can also be used to index a single dimension. However, a hash table is more efficient for exact queries than a B-tree, and a KDB-tree with one dimension is less efficient than a B-tree.

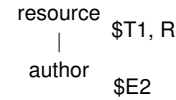


Figure 3: Sample index definition

3.2 Index Framework

Based on an index model, the index framework allows for integrating various index structures as well as extending and nesting them to index various properties of XML documents. Each index has an index definition that is used for selecting and maintaining indices without knowledge of the underlying index structures.

An *index* is a search function consisting of a set of index entries. Each entry in an index maps a list of index keys, specifying the search condition, to the nodes of the document that are returned by the search function. An index key can be any property of the XML data model, e.g. the value, type, path or labelpath of a node. The number of index keys determines the dimensionality of the index.

The *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. For example, an index variable may specify to enable range queries or word prefix search on node values. While an XPath like syntax can be used to create an index, an *index pattern* is adequate to process an index definition in a language-independent way to select and maintain indices [12]. The index definition language and its translation into patterns are out of scope of this paper.

Example 7. An index definition language can use the enhanced XPath expression `//element(resource, $V1)[author=$V2]` to define an index on the author value and resource type. Figure 3 depicts the corresponding index pattern. The symbols next to the pattern nodes refer to the index variables. Symbol \$T1 indicates that the first variable indexes the type, while \$E2 represents a variable on author nodes that allows for comparing their value on equality. Symbol R indicates which nodes the index returns.

An *index structure* is the specific data structure used to represent an index, e.g. a hash table, a prefix B+-tree or a KDB-tree. It builds a search structure on the index keys to efficiently retrieve the nodes that match certain keys. Each index structure organizes its index entries into pages, which are stored on disk. To traverse the structure, it uses certain operators to compare index keys. An index structure can index those index keys that can be compared with these operators. Nesting index structures entails that the index keys of one index structure do not map to nodes that match these keys, but to the pages of the nested index structure. The super index structure forwards the retrieval or manipulation task to the nested index structure, which traverses its structure accordingly.

Each index can use one or several index structures as underlying data structure. Using several index structures in one index implies nesting the index structures according to their order. To associate indices with their index structures, the *index configuration* specifies for each index variable of the index definition the kind of index structure to be used.

Example 8. A sample index configuration for the index of Example 7 specifies to index the first and second variable with a multidimensional index. Alternatively, we could define to build a hash table on the second variable and a (nested) B-tree on the first variable.

The index structures define a common interface which allows for accessing and maintaining an index. To access an index, the index selection tool generates a *search configuration* based on the index definition. The search configuration specifies the index keys for the index variables to be searched in the index as well as specific comparison operators if necessary (e.g. for range queries). With the help of the index configuration, each index structure can extract the index keys which it needs to search.

Example 9. When accessing the index of Figure 3 to evaluate the sixth query of Table 1, the search configuration specifies the index keys 'Report' and 'Smith' for the first and second variable, respectively.

The index framework does not only allow for selecting and maintaining indices without depending on specific data structures. It also enables to easily integrate new index structures (e.g. for more sophisticated text retrieval) and extend existing index structures to support new variables.

3.3 Index Maintenance

As index structures must be consistent with the documents on which they are defined, updates on documents need to be propagated to affected index structures. The flexibility of indexing arbitrary document fragments entails that the document fragments which are inserted, deleted or modified need not contain all nodes that are required for index maintenance. At the same time, the maintenance algorithm must not depend on the maintenance of required nodes in auxiliary data structures as to ensure adaptability of index structures to various requirements.

In [12], we presented a new index maintenance algorithm to determine which indices need to be updated with which index entries when updating document fragments. It is solely based on index definitions and update fragments instead of on the maintenance of auxiliary data structures. The use of index definitions assures that the algorithm supports arbitrary index structures defined on arbitrary document fragments. By exploiting the structure of update fragments, the algorithm directly extracts the nodes which are required for index maintenance from the fragments. Source queries are only necessary if the fragment does not contain all nodes required for indexing.

Example 10. When inserting a new resource, the index of Figure 3 needs to be updated with this resource. In this case, the update fragment contains all nodes that are necessary to generate index entries and no source queries are executed. If we insert a new author, the algorithm needs to determine the corresponding type of resource to generate a new index entry for this index.

The main idea of the algorithm is to represent index definitions as tree patterns and find embeddings of index patterns in update fragments. Embeddings consist of the nodes of a document that structurally correspond to the pattern. If the update fragment does not contain all nodes that are

part of an embedding, the algorithm issues source queries to retrieve missing nodes. Each embedding represents an index entry which is then forwarded to the specific index structure for maintenance. By exploiting the structure of update fragments, the algorithm minimizes the number of source queries compared to the approach of updating individual nodes [13, 12]. With the help of a labeling scheme [11], the answers to many source queries can be computed without accessing base data, which further improves the maintenance process.

4. PERFORMANCE STUDIES

To analyze the performance of the SCIENS framework, we compared the effect of different index configurations, using Java 1.5 and Berkeley DB Java 3.1. As sample dataset, we generated a document of 2 MB that corresponds to the running example. All experiments were carried out on a 3.2 GHz Pentium D processor with 2 GB RAM. We executed three queries each of which selects resources by specifying a certain date range and the following structural conditions: (Q1) selects a specific project milestone, (Q2) queries all milestones of one project, (Q3) looks within all projects. By varying the date range, each query returns the same number of resources. We used the following index configurations: (I1) indexes the date, (I2) is a multidimensional index on the date and the milestone hierarchy, (I3) nests an index on the milestone hierarchy beneath an index on the date, (I4) nests an index on the date beneath an index on the milestone hierarchy.

Table 2: Performance

time (ms)	I1	I2	I3	I4
Q1	598	74	199	11
Q2	84	68	57	27
Q3	14	63	20	207
average	232	68	92	82

Table 2 depicts the performance results. An index on the date (I1) is only appropriate when querying all projects, as it cannot limit the search to certain milestones. The multidimensional index (I2) performs equally for all queries, while nesting index structures either favors querying the entire hierarchy (I3, Q3) or only a certain part of the hierarchy (I4, Q1). The results demonstrate that the flexibility of SCIENS allows for adapting secondary index structures to optimally match the query workload. It also has to be added that the index size increases when indexing multiple properties. I2 needs more space than I1 and I4 as it needs to store splitting information. I3 has the worst space consumption as it repeats the milestone hierarchy for each date. The space consumption of I1 and I4 only varies slightly.

5. RESEARCH METHODOLOGY

The ideas presented in this paper emerged from a research project called SemCrypt, which is a native XML database that uses secondary index structures to efficiently process queries. Based on several use cases, we first defined queries that should be supported by indices. An extensive literature study revealed that existing index structures support different kinds of queries. XML and object-oriented index structures mainly propose proprietary index structures, which are difficult to integrate into a common framework.

