INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians-
Universität
München

A Survey of Indexing Techniques for Semistructured Documents

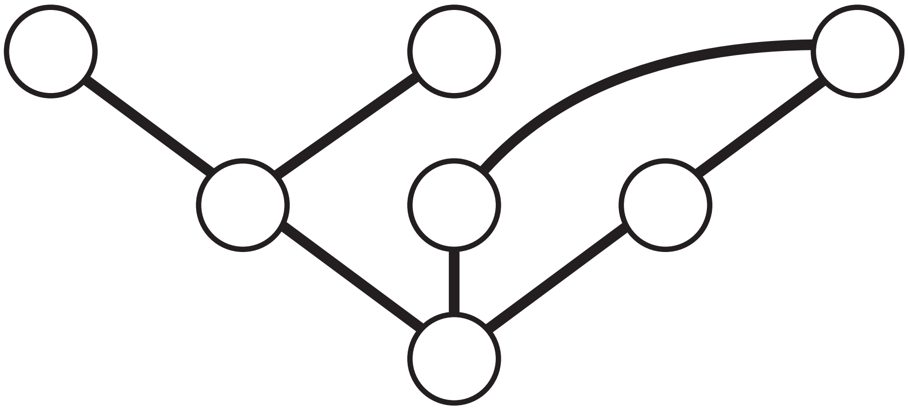
Felix Weigel

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	15.02.2002
Abgabe der Arbeit:	08.08.2002
Betreuer:	Prof. Dr. François Bry
	Dr. Holger Meuss

A Survey of Indexing Techniques for Semistructured Documents

Felix Weigel



Abstract

This work reviews fourteen approaches to indexing semistructured data, ranging from most simple look-up tables as used in traditional Information Retrieval to sophisticated graph-based index structures for combined structure and content queries. Each approach is described in detail, including illustrated examples. For comparison, we emphasize differences in the employed data structures, the document types supported, the kind of information to be retrieved (specified in terms of look-up input and output), the storage and update behaviour, and performance results. Moreover, specific features such as structural node identification, path templates, and internal navigation are examined. For some of the presented approaches, operative retrieval systems are described as "real-world" examples.

The major preliminaries from Information Retrieval, structured documents, and indexing in general are discussed in three introductory chapters. They include a generic retrieval system architecture and further data structures such as schema graphs, signatures, filters, and various Tries.

Zusammenfassung

Die vorliegende Arbeit stellt vierzehn Indexverfahren für semistrukturierte Daten vor. Die Bandbreite reicht von einfachen invertierten Listen, wie sie aus dem *Information Retrieval* bekannt sind, bis hin zu komplexen graphbasierten Ansätzen für kombinierte Text- und Strukturanfragen. Jedes Verfahren ist detailliert in Wort und Bild beschrieben. Als Vergleichkriterien dienen neben den jeweiligen Datenstrukturen die indizierbaren Dokumenttypen, die Zugriffsweise auf den Index (d. h. die jeweilige Ein- und Ausgabeinformation) sowie Leistungsmerkmale wie Speicherbedarf und Aktualisierungsverhalten bei Änderungen in der Datenbank. Zudem werden spezielle Funktionalitäten wie Strukturcodierung in Knotenidentifikatoren (*structural node identification*), Pfadvorlagen (*path templates*) und interne Navigation untersucht. Einsatzfähige Datenbanksysteme aus der Literatur illustrieren die praktische Anwendbarkeit der vorgestellten Ansätze.

Drei einleitende Kapitel fassen die wesentlichen Grundlagen und Begriffe aus dem *Information Retrieval* sowie für strukturierte Dokumente und Indexverfahren im allgemeinen zusammen. Insbesondere behandeln sie eine generischen Architektur für Datenbanksysteme sowie als weitere Datenstrukturen Signaturen, Filter und verschiedene Suchbäume (*Tries*).

Acknowledgements

The present work owes much to the kind support and expertise from various people at the Department of Computer Science and the Centre for Information and Language Processing at the University of Munich (LMU). In particular, the author would like to thank Holger Meuss, whose scientific insight and open-minded curiosity have made collaboration a pleasure. Sharing his know-how and theoretical background (including the generic retrieval system architecture in section 2.2), he has contributed much to this work. We are equally grateful for François Bry's sound and helpful advice and suggestions, as well as for his patience and confidence during the period leading to this work. Furthermore, our thanks go to Klaus U. Schulz for the example given in figure 6.13 on page 73 and his judicious comments on earlier drafts of this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Indexing semistructured data is at the frontier between Information Retrieval (IR) and database (DB) research. On the one hand, a variety of techniques have been developed over the years for finding information in text documents efficiently. Since the advent of the relational data model in the early seventies, numerous index structures for information stored in tables have been proposed and refined. It seems natural to investigate in how far these approaches can be applied to the textual part of semistructured data. On the other hand, the semistructured data model creates additional problems and demands, adding structure to what used to be flat documents or relations. It is true that object-oriented database systems have addressed these issues before, as they tried to store structured data in a more flexible and expressive manner than the relational or hierarchical databases of the early days. In the meantime, however, practical experience has shown that data to be exchanged by different parties (not only, but most prominently over the Web) often cannot be shoehorned into a predefined schema, as required by all traditional database management systems. Indices for semistructured data face a challenge hitherto ignored by all other models, namely to handle data with seemingly irregular structure whose underlying schema, although contained in their description, is usually not given explicitly.

The indexing approaches we present in this work build more or less all on the above-mentioned foundations in IR and DB research. Some may be more inclined to IR techniques, while others are easily recognized as originating from the database community. Nevertheless, their descriptions are either self-contained or refer to preliminaries from the preceding four chapters. That is to say, in neither of the two fields do we presuppose in-depth skills beyond a basic level of understanding. Each section begins with a concise summary, followed by a detailed analysis of the approach in question, and concludes with a short discussion of its primary features. The structure of this last part is fixed to ease comparison between the indexing approaches, and for later reference. A synopsis of all indices covered in this work can be found on page 41.

This work is organized as follows. Chapter 2 introduces the main principles of indexing in general. It also integrates indices into a generic retrieval system architecture, stressing the difference between query engine and index engine. Chapter 3 reviews major concepts and data structures from Information Retrieval. Chapter 4 examines how structured documents are represented in different data models, emphasizing semistructured data and their representation in a descriptive schema. Concluding the preliminary portion of the work, chapter 5 explains the salient comparison criteria employed in the subsequent survey of indexing approaches for semistructured documents. General solutions to selected issues are presented in this chapter, too, as a basis for later classification. Chapter 6 consists of the above-mentioned descriptions of various indexing approaches, grouped into three classes which are introduced in a short overview before. Chapter 7 summarizes the survey, putting together the pieces to form a picture of the state of the art in semistructured document indexing.

Chapter 2

Indexing

2.1 Concepts

Indices are data structures used to locate specific parts of a collection of data. Examples include the classic book index, which retrieves occurrences of a given word in a text, or indices in relational database management systems, which help to find certain records without searching an entire table. These cases illustrate the main benefit of an index: it speeds up query evaluation, allowing to access the relevant data *directly*, i.e. without a sequential scan of the database. The position of the data may be located by any means suitable for the indexed database and the querying party—while for a book page numbers are appropriate, an index for electronic files is likely to return byte offsets. Of course the benefit of accelerated query evaluation comes at a price: indices require additional storage space, and depending on the data structures involved this storage overhead may be significant. Hence some indexing approaches focus on space reduction, or else come with elaborated paging strategies for efficient disk access.

Our examples above highlight the most prominent application of index structures, namely information retrieval in a general sense. There are other cases where indices prove valuable, however, like data compression (see e.g. [LS00]). In this work we focus on retrieval systems, which make use of indices as auxiliary data structures for managing a database. In terms of the generic architecture of retrieval systems discussed in the next section, one can say that indices help to shift some of the work-load from the query engine to a dedicated component, the index engine.

Before examining retrieval systems in more detail, we would like to hint at some basic properties of index structures which are relevant for a later comparison of different approaches. First, an index may be more or less capable of adapting to changes in the database, i.e. insertion or deletion of records. If an existing index structure can be modified to reflect these changes, and the modifications are restricted to a small part of the index, it is said to support *incremental updates*. Most index structures we will encounter can be updated incrementally; some only to a certain extent, i.e. up to a limited number of changes in the database. There are cases, however, in which the whole index structure needs to be rebuilt from scratch. Obviously such indices are less recommended for dynamic, i.e. frequently changing databases.

Some index structures not only retrieve hits for a given query, but also propose a ranking according to their relevance. This *weighted retrieval* is most common with text queries and has become a standard feature of search engines on the Web by now. Due to the unformalized semantics of natural language, however, algorithms for determining how pertinent an individual hit is necessarily rely on heuristics. For instance, the BUS index computes the relevance, or *weight*, of a retrieved portion of the database based on how many times the desired keyword occurs in it. This heuristic formalization of the concept of "relevance" is grounded on an implicit assumption—the more often a word occurs in a text, the more it matters. [1]

[1] If multiple occurrences of the same keyword are directly contained by a single node, however, they usually are not accumulated. Direct containment is defined on page 25.

Finally, there are *exact* and *inexact* indices. Intuitively, an index is exact if it retrieves all relevant data for a given query, and no others. In terms of the performance measures introduced in the next chapter, exact indices are those with optimal *precision* and *recall*[2]. We would like to point out though that the notion of exactness is more intricate than it appears at first glance. An index may well perform with optimal precision and recall yet return more items than expected. This seeming contradiction can be explained when distinguishing two notions of exactness: a first one defined with respect to the user's query addressed to the retrieval system, and a second one related to the look-up criteria given to the index. Anticipating the terminology introduced in the next section, one can say that whether an index is considered exact or not depends on how precision and recall are measured: based on the input to either the *query engine* or the *index engine*. For example, the Context Index presented in section 6.3.2 filters keyword occurrences according to structural criteria. To this end, the user supplies the query engine with a label path, which is transformed into a filtering expression and handed over to the index engine. However, its lossy encoding as a bit string makes the filtering expression ambiguous, so that the Context Index may fail to filter out some occurrences which are not associated with the label path. Nevertheless the index engine applies the given (ambiguous) filtering criterion in an exact manner. The inexactness here is a matter of the filtering expression, not the index.

An index with optimal recall (but possibly suboptimal precision) is called *conservative*, whereas one with full precision (but maybe partial recall) is said to be *progressive*. Non-conservative indices are of limited interest for most applications, although relevant for screening out low-ranked results in weighted retrieval. Results from an inexact index are usually postprocessed by the query engine in a *verification* step.

2.2 Retrieval systems

Figure 2.1 below shows a generic architecture for retrieval systems. In the rest of this work we implicitly refer to this model, and the terminology associated with it, when talking about look-up patterns and strategies, query plans, index and query engines, and so on.

Viewed from a high level of abstraction, a generic retrieval system consists of two kinds of component: on the one hand, the *query engine*, and on the other hand, a set of *index engines*. The two components are represented by rectangles at either side of the grey bounding box symbolizing the retrieval system. In this idealized picture additional functionality apart from query evaluation and indexing, such as maintenance, synchronization, recovery, etc., is omitted. The query engine is responsible for accepting *user queries* and creating *query plans* to evaluate them, which may involve any number of *index look-ups*. Each of the index engines provides a specific *index interface* determining which sorts of look-up it can handle. We discuss *look-up patterns* as interface specifications in terms of input and output parameters in the next section. The external and internal representation of query expressions does not matter here. Notice, however, that the query engine may transform the user query in order to create *look-up queries* conforming to a look-up pattern in any of the index engines' interfaces. In particular, the query plan may require the user query to be split into multiple look-up queries, each retrieving hits for a part of the user query which need to be joined in the end. For example, tree queries may be handled this way when only indices supporting path expressions are available. In another common scenario, the query engine consults both a structure index and a content index to evaluate combined path and text queries. Here we distinguish a *top-down* query plan, which looks up the structural part first and then matches the keywords, from a *bottom-up* approach which inverts the look-up order. These examples illustrate that it is important to keep in mind the difference between user queries, addressed to the query engine, and look-up queries, processed by an index engine.[3]

---

[2] These two performance measures for retrieval systems are defined formally in the next chapter. Suffice it to say here that optimal *precision* guarantees that no irrelevant data are retrieved, while optimal *recall* means no relevant data are overlooked. The terms *query engine* and *index engine* are introduced in the next section, while *label paths* are covered in definition 8 on page 24.

[3] We will occasionally refer to either of them as *query* when it is clear from the context which one is intended.

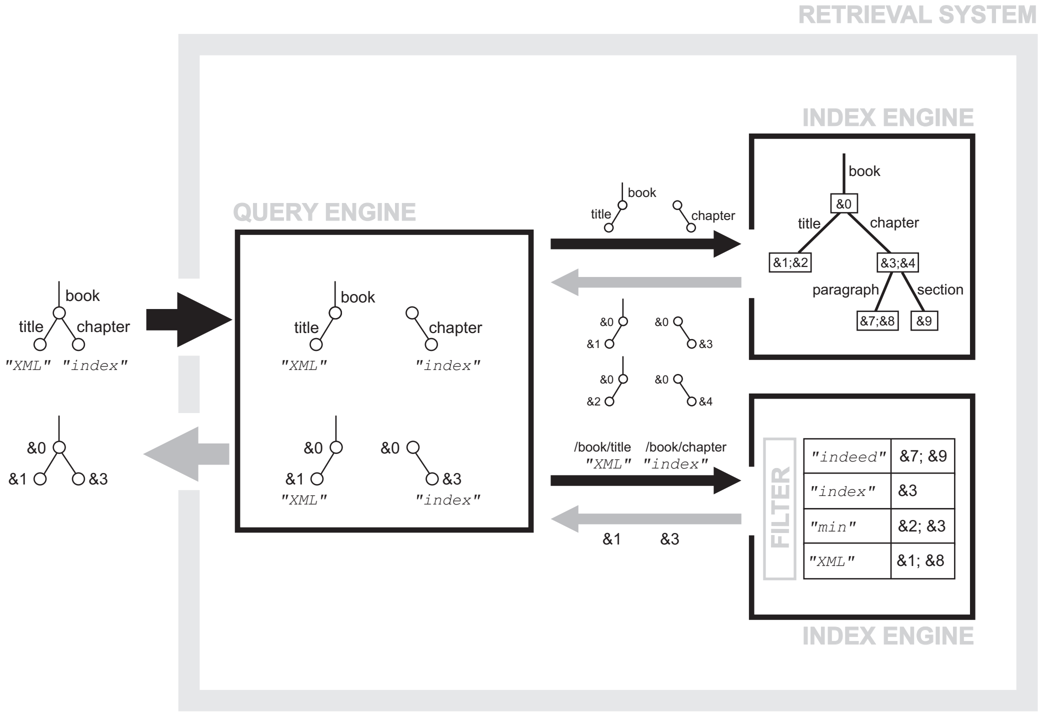


Figure 2.1: General architecture of a retrieval system

Analogously, *query plans* and *look-up strategies* must not be confused. While a query plan specifies which index look-ups need to be carried out in which order on behalf of the query engine (along with many other operations involved in query evaluation), a look-up strategy describes how a single index look-up is organized internally by the index engine. Look-ups in table-based indices are usually atomic operations, whereas graph index structures may be navigated either *top-down*, or *bottom-up*, or *inside-out*, or else in a *hybrid* fashion, as explained in section 5.4. Sometimes the distinction between a query plan and a look-up strategy is slightly blurred, especially when it is not clear whether secondary indices are part of another index structure. Often query plan and look-up strategy are arranged in parallel, as e.g. when an index with top-down navigation (such as the DataGuide in section 6.4.3) is used for the structural part of a top-down query plan. Consequently, in [MWA+98] the concept of a query plan includes internal index navigation. Here we insists on separating index engine and query engine, conforming to the generic retrieval system architecture presented above.

As a third component, *filters* are located inside index engines between their internal data structures and the index interface.[4] They intercept the index engine's answer to a look-up triggered by the query engine in order to discard those items which do not match a given filtering criterion. Obviously this only makes sense when the index returns more database references than judged fitting for a given query, and from this point of view is inexact. The previous section highlighted a pitfall concerning the exactness of indices. For filters the term is equally ambiguous: the same filter may be considered inexact with respect to the user query, but claimed exact when precision and recall are measured based on the filtering criterion.

Just like indices, filters are designed to handle either structure or content (see section 4.4).

[4] Filters are considered part of the index engine here, although one could also argue that being independent of the underlying index, they rather constitute components in their own right.

Among the index structures presented in this work, there are two with an integrated filter: the Context Index (section 6.3.2) and the Signature File Hierarchy (section 6.4.2). With the help of their respective filter, both anticipate a selection normally carried out by the query engine after the look-up, but in an inexact manner to avoid too much storage overhead. In case of keyword look-ups, this means filtering out some of the occurrences which are bound to fail at structure matching anyway. As a content index with a structure filter, the Context Index pursues this strategy. Analogously, when searching all nodes reachable by a given label path, one might want to single out those which clearly do not contain the desired keywords. This is done by the Signature File Hierarchy, a structure index with an integrated content filter. Both approaches are based on signatures which are covered in section 3.3.

2.3 Look-up patterns

As pointed out by [MS99a], indices implement a mapping from a specific domain (e.g., keywords or labels) to a set of occurrences, whose definition varies from index to index. For example, the label index described in section 6.2 maps element labels to node identifiers (IDs), whereas the CIS index from section 6.3.1 takes a keyword and returns label/ID paths as occurrences. The mappings supported by a given index are expressed by its index interface. Look-up patterns, which have already been introduced as a means to describe an index interface in terms of input and output, are therefore convenient for describing the functionality or mapping implemented by any index structure. For each approach presented in this work, the corresponding interface is given as a table of look-up patterns. Representing mappings, they are written in a functional notation $X \times Y \longrightarrow \mathcal{P}(Z)$ where X, Y are sets and $\mathcal{P}(Z)$ denotes the powerset of a set Z. The left-hand side of the pattern specifies the input parameters for the corresponding look-up, whereas the right-hand side indicates the type of the returned index entries, or *postings*. Individual index entries are written using a mapping notation, as in $(\mathsf{x}, \mathsf{y}) \mapsto \mathsf{z}$. The following names are used for the different input and output domains:

DocNodes	set of document nodes	*(section 4.1)*
Keywords	set of keywords	*(section 3.2)*
Labels	set of element labels	*(section 4.1)*
NodeIDs	set of node identifiers	
DocIDs	set of document identifiers	
Predicates	set of selection predicates	*(section 6.2.3.2)*
Signatures	set of signatures	*(section 3.3)*
Weights	set of weights	*(section 3.2)*
LabelPaths	set of label paths	*(definition 8, p. 24)*
SimplePaths	set of simple paths	*(definition 14, p. 25)*
IDPaths	set of identifier paths	*(definition 9, p. 24)*
PositionPaths	set of position paths	*(definition 11, p. 24)*
SignaturePaths	set of signature paths	
RelativePaths	set of relative paths	*(definition 13, p. 25)*
PrivilegedPaths	set of privileged paths	*(section 5.3)*

Table 2.1: Notational conventions for look-up patterns

Predicates is the set of selection expressions supported by a value index (see section 6.2.3.2). *IDPaths* are paths where each step is a node ID. *PositionPaths* are paths where each step indicates the position of an individual node, as e.g. its sibling number (see section 5.5.4). *SignaturePaths* are paths where each step is a signature (see section 3.3). *PrivilegedPaths* are those matching a given path template (see section 5.3). As a further convention, each such name in its singular form, beginning with a lowercase letter, denotes a generic element from the corresponding set. For instance, let $keyword \in Keywords$, $nodeID \in NodeIDs$, and so on.

Chapter 3

Information Retrieval

Modern Information Retrieval as we know it dates back to the late 1950s when automatic searching of bibliographic records began to gain momentum from the rapid progress in computer science and engineering. Amidst a prolific variety of algorithms and variations there seems to be a set of core definitions, techniques, and data structures today, part of which we summarize briefly here. For a more extensive overview the reader is referred to [SM83]. There are also a number of comprehensive on-line tutorials and other resources available, such as [Vir], [vR79], or [Sig].

The basic retrieval units in Information Retrieval are documents, which are characterized by the individual words they contain, regardless of their structure. While indexing structured documents usually requires a finer granularity of both queries and results, in this chapter we assume for the sake of simplicity that entire documents are retrieved.

3.1 Retrieval performance

The two main concepts for assessing the quality of search results for a given query are *precision* and *recall*. They can be generalized to measure the performance of the corresponding retrieval system as a whole. All of the following definitions refer to the set of documents in the collection as D. Besides, let O be the subset of documents matching the given query, and R the set of hits returned by the system. Figure 3.1 illustrates the three sets schematically. Note that O and R could also be disjoint or contain exactly the same elements.

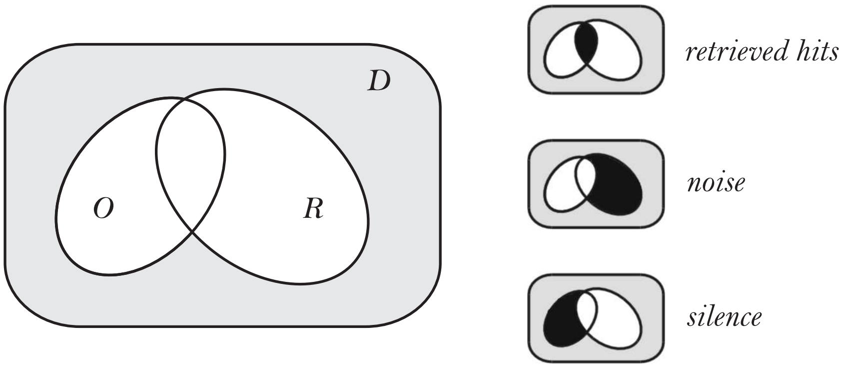


Figure 3.1: Retrieval precision and recall

Definition 1 *(Precision) The* precision *of the retrieval process for a given query can be computed as* $\frac{|O \cap R|}{|R|}$*, i.e. the proportion of relevant search results to all search results.*

Definition 2 *(Recall) The* recall *of the retrieval process for a given query can be computed as* $\frac{|O \cap R|}{|O|}$*, i.e. the proportion of relevant search results to all documents matching the query.*

Two supplementary definitions serve to reference the documents which have or have not been returned as search results erroneously:

Definition 3 *(Recall noise) The* recall noise *(or simply* noise*) contained in the search results for a given query is* $R \setminus O$*, i.e. the set of false hits.*

Definition 4 *(Silence) The amount of overlooked hits,* $O \setminus R$*, is called* silence.

3.2 Content representation

In case of "flat" documents, which lack internal structure, the whole information is encoded as textual content. However, indexing all words occurring in a large document collection is infeasible. Most retrieval systems rely on one or more of the following techniques to reduce the number of words to be taken into account. Notice that since the resulting set of *representative words*, or *keywords*, captures only part of the original information, all these heuristics may affect a system's performance in terms of either precision or recall, or both.

Stemming. Many occurrences of different words are similar in the sense that they actually denote the presence of the same keyword, but with morphological variations. For instance, the verb "to index" and its gerund "indexing" can be regarded as equivalent in this sense. Replacing variations of the same keyword by its stem reduces the number of distinct words in the database. In English, most inflected words contain their stem as a prefix, which simplifies the stemming algorithm.

Stop word list. An even simpler way of reducing the number of words to be indexed is to exclude some of them explicitly. Among these *stop words* there are usually terms playing a syntactic rather than semantic role (so-called *function words*, like e.g. prepositions or articles), or words occurring too frequently to select a reasonable subset of documents in a query. For example, if all documents in the collection happen to contain the word "index," then there is no point in storing this information in an index because a search based on this keyword would simply return the whole database. As explained in the previous chapter, indices strive to narrow down the search to specific parts of the database.

Weighting. There are cases where it is not enough to determine whether a word shall be indexed or not, but also to which degree it is representative of, or typical for, the document it occurs in. To this end keywords identified by the previous two techniques are assigned *weights* expressing their importance for indexing. (For details on common weighting algorithms, see e.g. [SM83].) A keyword's weight may be either constant for all its occurrences (*plain word weight*) or vary according to the context it appears in (*word-document weight*). For instance, a keyword occurring in only two out of several hundred documents could be considered characteristic and hence useful for indexing. However, if one of the two documents contains significantly more occurrences of this keyword than the other, it seems inappropriate to use the same weight in both cases. As a remedy, common weighting techniques like *tf.idf* (for "term frequency/inverted document frequency," see below) calculate weights based on three statistical parameters to model a keyword's relevance for a given document:

Definition 5 *(Term frequency) The* term frequency tf_{kd} *of a keyword* k *with respect to a document* d *is the number of times* k *occurs in* d*.*

Definition 6 *(Document frequency) The* document frequency df_k *of a keyword k is the number of documents containing at least one occurrence of k.*

Definition 7 *(Collection frequency) The* collection frequency cf_k *of a keyword k is the total number of k's occurrences throughout the document collection.*

All of the techniques above are subsumed under the term *indexing by extraction* (or *derived indexing*) because they extract the set of keywords from the documents. Alternatively, one can define document classes in advance, which are described using a fixed set of keywords. Indexing documents based on these keywords then means assigning them to one or more of the predefined classes. Hence this approach is called *assigned indexing*. The difference between the two methods could also be captured by the terms *dynamic* vs. *static keyword selection*.

3.3 Signatures

Another way to handle the textual content of large document collections is to represent it only approximately in the index. A data structure often used for this purpose is the *signature*. Signatures are bit strings of uniform length encoding which keywords occur in a document. Figure 3.2 shows how a signature is created for a document containing two representative words, which we assume have been identified beforehand as described in the previous section. First each such keyword is assigned a (preferrably unique) bit string, or signature, of a given length. Then the signatures of all keywords occurring in the document are combined by bitwise disjunction (`OR`) to produce the document's signature, which can be used for indexing as well as filtering.

"index"	01100000	≠	01000100	*"midi"*
"XML"	00000110	≠	00100010	*"min"*
"index" OR *"XML"*	01100110	=	01100110	*"midi"* OR *"min"*

Figure 3.2: Ambiguous signatures

Notice that every keyword participating in the disjunction leaves a "trace" at those positions where it has a `1` in its signature. However, when multiple keyword signatures have a bit set at the same position, a document's signature does not reveal which ones left a trace and which ones did not. For instance, one cannot tell from the signature `01100110` whether the word "index" occurs in the corresponding document, since a couple of other keywords together produce the same signature, as shown in the figure above. Obviously such ambiguities arise even when using unique keyword signatures. Of course signatures could be devised as to allow for exact content representation. A naive way to achieve this is to use bit strings reserving one bit for each keyword. However, the resulting space requirements are doomed to be unaffordable in case of a non-trivial database. When exact content representation is out of scope, there exists a trade-off between the number of keywords, the signature length, and the precision of the signature-based index or filter. Fixing one of the three leaves two antagonists. For instance, given a fixed number of keywords, precision decreases with the number of bits per signature. Likewise for signatures of a given length, precision declines as more representative terms are taken into account.

As the Signature File Hierarchy (section 6.4.2) shows, signatures encoding textual content are useful also for indexing semistructured data. But they can represent the structural context of a keyword occurrence, too, as e.g. with the Context Index (section 6.3.2). For an overview of earlier signature applications, see [Fal85] and [DML98].

3.4 Index structures

A variety of index structures have been proposed to enhance text retrieval in unstructured documents. Here we briefly review some table-based and tree-based indices which are also at the heart of certain approaches for semistructured data.

Inverted file. *Inverted files*, as the most basic text indices, are similar to a classic book index. For each keyword they store a list of references to its occurrences in the text. Occurrences may be referenced by page number or byte offset or any other method, depending on how the indexed text is given. In case of a book index, a look-up for a given keyword yields the numbers of pages where the keyword appears in the text. The common way to use a book or file, however, is exactly the contrary: one looks at a given page to read the words which are written there—hence the name "inverted file". Figure 3.3 shows an inverted file for six keywords, each of which is associated with a list of the documents it occurs in. Section 6.2 presents different inverted files for semistructured data.

"die"	&4
"index"	&0; &3; &4
"indeed"	&1; &3
"midi"	&2
"min"	&3; &4
"XML"	&1; &2; &4

Figure 3.3: Inverted file

Document/term matrix. The *document/term matrix* is a two-dimensional matrix (k, d), which for each keyword (or *term*) k and each document d holds a `1` if k occurs in d, and a `0` otherwise.[5] Figure 3.4 depicts such an index for the six keywords and five documents from figure 3.3. A document/term matrix is similar to an inverted file in that each keyword is associated with references to its occurrences. However, the corresponding references are not kept in a separate list for each index entry. There is a single list instead containing the names (or identifiers) of all documents in the collection (the column headers in the matrix), and for each keyword the ones containing at least one of its occurrences are marked with a `1`. As a result, the document/term matrix differs from inverted lists in two ways. First, it has two look-up patterns instead of one: the index can be queried either for a given keyword or for a given document identifier, in which case it returns a list of the keywords occurring in the corresponding document. Second, the lists of occurrence references or keywords retrieved contain not only relevant items (which are marked with a `1`), but also list all other documents or keywords in the database. Obviously this changes the way look-up results are processed by the query engine, because the `0`s need to be filtered out. Besides, a document/term matrices for huge document collections with many representative words is likely to be sparsely populated because usually each document contains only a fraction of all keywords in the database. We will encounter this problem and a solution to it when examining the BitCube in section 6.3.3.

[5] A Boolean document/term matrix has only `0`s and `1`s. Alternatively, keyword weights (see section 3.2) can be stored.

	d_0	d_1	d_2	d_3	d_4
"die"	0	0	0	0	1
"index"	1	0	0	1	1
"indeed"	0	1	0	1	0
"midi"	0	0	1	0	0
"min"	0	0	0	1	1
"XML"	0	1	1	0	1

Figure 3.4: Document/term matrix

Trie. A *Trie* ([Fre60]) is a labelled tree index for re*trie*ving keywords over a given alphabet, as shown in figure 3.5 for the keywords from figure 3.3. Each symbol in a given keyword is represented by a single edge in the Trie, which is labelled with that symbol. The whole string can be reconstructed symbol by symbol by following the corresponding edges from the index root downwards. Index nodes can be annotated with references to the occurrences, e.g. byte offsets in a file, which have been omitted here. Keywords with a common prefix share the same path prefix in the Trie. For instance, since the words "index" and "indeed" differ only in their fifth letter, their paths in the Trie have the first four nodes in common before splitting up into an x and an e branch.

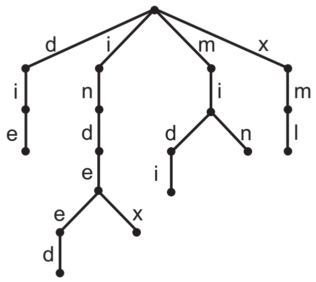


Figure 3.5: Trie

PATRICIA Trie. *PATRICIA Tries*[6] ([Mor68]) enhance ordinary Tries with string compression. The idea is to avoid chains of index nodes with only one outgoing edge. While the Trie represents each symbol in a given keyword as a separate edge, the PATRICIA Trie omits those symbols which would create a singleton edge, indicating that there is no other keyword with the same prefix and a different symbol at this position. Such an edge is not needed for disambiguation and can therefore be safely neglected. For instance, in figure 3.6 the whole path prefix /i/n/d/e is replaced by a single edge labelled i because among keywords beginning with "i", there is none whose second letter is not "n"; for those beginning with "in", the third letter is always "d", and so on. Following the i edge from the root, it is taken for granted that the next letters in the keyword

[6] PATRICIA is an acronym for *P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric.

are “n”, “d”, and “e”, as no other words appear in the database anyway. For the fifth letter, however, there is a choice between “x” and another “e”. Hence these two symbols are represented explicitly by two separate edges leaving the same node. In order to know which symbols in the keywords they stand for, the node reached by /i/n/d/e is annotated with the length of the prefix processed so far (in this case, 4). In other words, three symbols after the “i” have been skipped in creating this edge. This number can also be read as the level this node would reside on in the ordinary Trie (compare this to figure 3.5).

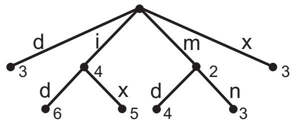


Figure 3.6: PATRICIA Trie

Obviously string compression in the PATRICIA Trie contributes to a much shallower shape of the index. However, it is lossy in the sense that a word not occurring in any of the documents may still match an index path, leading to a node which actually represents some other word. The reason is that if the two words differ only at positions skipped in index creation, then the query keyword is matched only at those positions where it is identical with the indexed keyword. For example, a search for “inbox” would end up in the leaf node created for “index”. However, the PATRICIA Trie can be further annotated to compensate for this lack of precision. When every index path stores the complete string it encodes, it is easy to find out whether a keyword matching this path really exists in the database.

The IndexFabric extends the concept of PATRICIA Tries to structured documents. We shall see in section 6.4.5 how label paths and keywords are stored within the same index structure.

Suffix tree. The *suffix tree* is a Trie built over all suffixes of a given set of strings. In other words, besides the actual strings from the database, the suffix tree also stores every other sequence of symbols which is a suffix of one of the original strings. Figure 3.7 shows the suffix tree for the keywords from figure 3.3. Unlike the Trie and PATRICIA Trie above, this index retrieves occurrences of the word “deed” with a single path traversal. The term is indexed because it is a suffix of the representative word “indeed”. In the other index structures this word might only be found after scanning the whole tree.

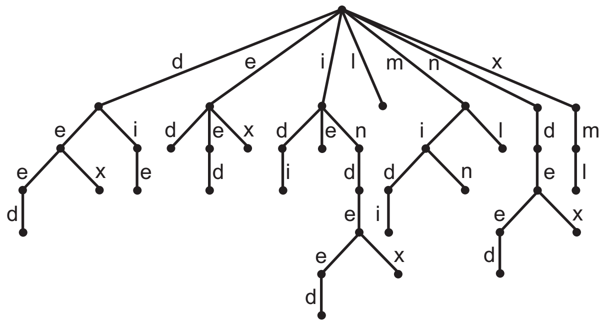


Figure 3.7: Suffix tree

PAT tree. Similar to the suffix tree, the *PAT tree* ([GBYS91]) indexes all keyword suffixes in the database, but with PATRICIA Trie compression. Figure 3.8 depicts the PAT tree corresponding to the suffix tree in figure 3.7 above. Again the PATRICIA Trie compression makes the index much shallower.

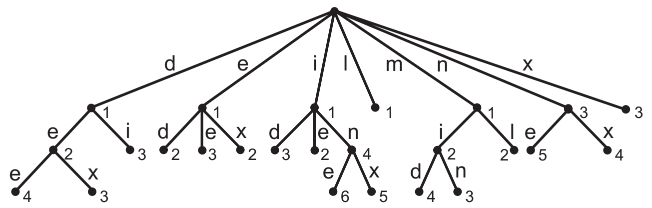


Figure 3.8: PAT tree

Chapter 4

Structured documents

4.1 Data models

As we have seen in the previous chapter, classic Information Retrieval used to regard documents as unstructured sequences of words. Since the late eighties, however, several data models have been proposed to capture the internal structure of documents. An overview of six different approaches can be found in [NBY97]. They range from *fields* covering selected portions of a document (*Hybrid Model*, [BY94]) over *regions* defined through pattern matching expressions (*PAT Expressions*, [ST92], *Overlapped Lists*, [CCB95]) to tree and graph structures (*Parsed Strings*, [GT87], *Tree Matching*, [KM93]; *Lists of References*, [Mac90]). While the first are restricted in their expressivity, still adhering to a linear view of the document, the latter are too complex to be implemented efficiently.

The *semistructured data model*[7] ([Abi97], [Bun97], [Suc98]) is a compromise between expressivity and efficiency. A wealth of applications based on XML (and the earlier SGML), along with implemented formalisms for querying, indexing, transforming, and rendering these data, prove that the semistructured data model is sufficiently expressive for a broad range of domains and at the same time can be processed efficiently. It is particularly suitable for collections of irregularly structured documents whose similarity suggests that they all belong to the same (hypothetical) class, which may still be hard to define explicitly. (The next two sections discuss this issue in more detail.)

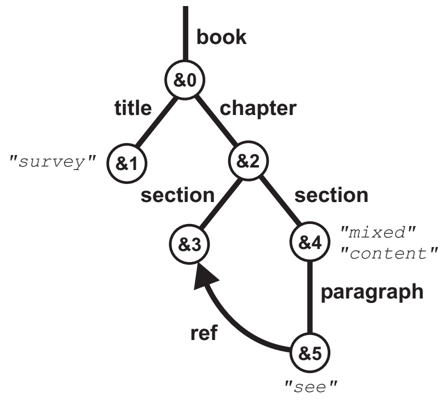
In any case, modelling the structure of documents creates new challenges for indexing, as pointed out by [MS99a]: instead of mapping a linear sequence of words to a linear storage layout, indices for structured documents must cope with a super-linear document topology. However, unless the structure imposed on the formerly flat document is defined externally, e.g. in a set of PAT Expressions or a grammar for Parsed Strings, it needs to be manifest in the physical representation of the document, which is still a linear string of characters. XML as a common serialization formalism for semistructured data specifies the document structure using metasymbols, or *markup*, interleaved with the original text. *Elements* enclose parts of the "atomic" text content, marking the beginning and end of such a range with labelled *tags*. The enclosed text may be marked up recursively, i.e. elements can be nested. As a logical abstraction of this text-oriented view of the document, the semistructured data model ignores start and end points in the text. Elements as the basic structural units are modelled as *nodes* in a labelled directed graph, and a node's children represent what is surrounded by the start and end tag of the corresponding element. In other words, text content as well as nested elements appear in the graph as nodes linked by edges. In

[7] To be precise, there are several models suitable for semistructured data, rather than a single semistructured data model. They differ e.g. in whether parent/child relationships and cross-links are conceptually different, whether there may be multiple roots, or whether the order among sibling nodes matters or not. The one we refer to as "the semistructured data model" throughout this work closely resembles the graph-based data model for XML presented in [GMW99], which was originally developed for migrating the LORE database system to XML (for details on LORE, see the end of section 6.4.3.3).

this work we refer to structural units in terms of both the (physical) serialization and the (logical) graph metaphor, depending on the context. In particular, when discussing the navigational indices presented in section 6.4 it is more natural to use the graph terminology.

```
<book id="ID0">
  <title id="ID1">
    survey
  </title>
  <chapter id="ID2">
    <section id="ID3"></section>
    <section id="ID4">
      mixed content
      <paragraph id="ID5" ref="ID3">
        see
      </paragraph>
    </section>
  </chapter>
</book>
```

(a) XML document



(b) graph representation of *(a)*

Figure 4.1: Physical vs. logical representation of a structured document

Figure 4.1 exemplifies the analogy between the physical and logical representation of a structured document. As can be seen in the figure, the name of an element in *(a)*, which appears in its start and end tag, is used in *(b)* to label the edge leading to the corresponding node in the graph. For instance, the root element `<book>` in *(a)* is homonymous with the root edge in *(b)*. In the serialized document an element's *type* is given in the tag name. Analogously, the type of a node in the graph can be deduced from the label of its incoming edge(s). Alternatively, the node itself could be labelled. When *node sharing* is allowed, however, i.e. when nodes may have multiple parents, labelling edges rather than nodes proves more flexible. Unlike XML, we consider sibling nodes as unordered, except when sibling numbers are used (see section 5.5.4), as e.g. by the Element Locator Scheme presented in section 6.3.1. Each node has a unique identifier which is a positive integer, preceded by the ampersand (&) sign.

The following hierarchical concepts are related to this graph-based data model:

Definition 8 *(label path; instances; determinism) A* label path *is a sequence of labels belonging to edges which together form a path in a given graph. For example, in the graph shown in figure 4.1 (b), the label path* /book/chapter/section *occurs twice. We say that the same label path may have multiple* instances *in a graph. When there is at most one instance of any label path in the graph, navigation in the graph, and also the graph itself, is said to be* deterministic.

Definition 9 *(ID path) An* ID path *is a sequence of node identifiers belonging to nodes on the same path in a given graph. For instance, in the graph shown in figure 4.1 (b), there is an ID path* &0/&2/&4.

Definition 10 *(label/ID path) A* label/ID path *is an ID path in which each ID is annotated with the label of the corresponding node. For example, in figure 4.1 (b), there is a label/ID path* /book$_{\&0}$/chapter$_{\&2}$/section$_{\&4}$.

Definition 11 *(position path) A* position path *is a sequence of tokens identifying nodes on the same path by their position in a given graph. For instance, the label/sibling paths described section 5.5.4 combine a positional node identification (based on the number of preceding siblings of a specific type) with labelling. In figure 4.1 (b), there is a position path* /book[0]/chapter[0]/section[1].

Definition 12 *(absolute path/root path) An* absolute path *is a path starting at the root, or one of the roots if there are multiple graph roots.*

Definition 13 *(relative path)* Relative paths *are those starting at a node of arbitrary depth in the graph.*

Definition 14 *(simple path) A* simple path *is an absolute label path without wildcards (see the path notation below).*

Definition 15 *(target set) A path's* target set *in a given graph is the set of nodes reachable by that path in the graph. For example, in figure 4.1 (b), the target set of the simple path* /book/chapter/section *is* $\{\&3, \&4\}$.

Definition 16 *(direct containment) A node n_2 is* directly contained *in another node n_1 if n_1 is a parent of n_2.*

Definition 17 *(indirect containment) A node n_2 is* indirectly contained *in another node n_1 if n_1 is an ancestor of n_2.*

Note that nodes directly containing keywords need not be leaves if mixed content is allowed in the document. As an example, consider the parent of the paragraph node in figure 4.1 *(b)* above. Although not a leaf, it directly contains an occurrence of the keyword "index". As [MS99a] states, there are multiple definitions of keyword occurrences, some of which exclude mixed content.

Definitions 8 through 11 have already introduced the notation for label paths, ID paths, label/ID paths, and position paths. For all of them, let ϵ denote the empty path. A leading slash / indicates an absolute path. Path expressions written in `teletype` instead of sans-serif font do not stand for path fragments in a given graph, but are query expressions. In regular path expressions we use the underscore _ as a wildcard for any single label. The plus sign $^+$ stands for the transitive hull and an asterisk * for the reflexive-transitive hull of the preceding item. For instance, $\_^*$ matches any number of subsequent labels whatsoever. Analogously, `section`^{3} means a three-fold nesting of section nodes. The slash / separates a parent from its child, while the ancestor/descendant relationship can be expressed by $/\_^*/$. Attribute names are preceded by the at sign @. In path expressions, attributes are treated like child elements: e.g., `/book/@title` selects the title attribute of the root book. Disjunctions are denoted by a vertical bar | surrounded by parentheses, as in (`chapter`|`appendix`). Predicates selecting nodes based on criteria other than their label are enclosed in square brackets. For example, `/book/chapter["index"]` selects all chapter children of book nodes which contain an occurrence of the keyword "index". For ordered selection criteria like numbers, two dots .. denote a range (such as in `/book/price`[19.95..39.95]).

4.2 Descriptive schema

In chapter 2 we observed that the main goal of an index is to avoid sequential database scans. When evaluating structured queries against a collection of structured documents, a summary of the hierarchical relationships encountered in the database helps to achieve this goal. In fact, all navigational indices (see section 6.4) presented in this work make use of such a concise representation of the database in its current state, which we will call a *descriptive schema* (as opposed to a *normative* one, see the next section).

First we consider a collection of tree-shaped documents only.[8] Let P denote the set of distinct simple paths in the collection. A descriptive schema for a tree database is itself a tree where each member of P appears exactly once, and no other paths exist.[9] As an example, consider the schema

[8] The following definition is based on the DataGuide index introduced in [GW97a] (see section 6.4.3). In this paper, the term "dynamic schema" is used for what we refer to as "descriptive schema".

[9] It can be shown that when trees differing only in the order among siblings are considered isomorphic, there is exactly one descriptive schema for any tree-shaped database.

tree shown in figure 4.2 *(b)*, which summarizes the database depicted in *(a)*. It is easy to verify that indeed each absolute label path from the database is represented by exactly one label path in the schema. Identical label paths, like e.g. p_2 and p_3, seem to have been "merged". Compare this to the *Trie* described in section 3.4, which indexes keywords in much the same way as the schema tree handles label paths.

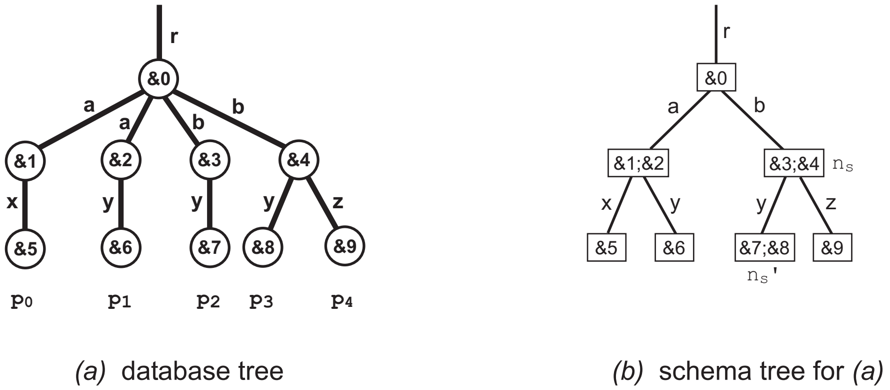


Figure 4.2: Database tree vs. schema tree

When matching a path query against such a schema tree, there is always at most one path to follow. With such a deterministic schema tree it is fairly easy to find out whether hits for a given query path exist in the database or not. Unless the query contains wildcards, all one needs to do at each level in the tree is to follow the appropriate label, provided there is one. If the whole query path can be matched this way, there is at least one hit; otherwise no matching paths exist in the database.

However, it is usually not enough to know whether there are any document paths matching the query or not. To locate the path occurrences in the database, and identify the individual nodes involved, each simple path in the schema is *annotated* with references to its target set in the database tree. This turns the schema into a navigational *structure index*. As can be seen in figure 4.2 *(b)*, the annotations are stored in the schema nodes. For instance, the simple path `/r/b/y` in the schema tree references the document nodes `&7` and `&8`, the only ones reached by `/r/b/y` in the database tree *(a)*. In a typical top-down look-up scenario, when given the query `/r/b/y["XML"]`, one would navigate the schema from the root down to the leaf node containing these references, and then consult a content index to find out whether the referenced document nodes contain the keyword "XML".

Formally, the nodes of a schema tree correspond to equivalence classes $[n]_{\equiv_p}$ induced by a relation $\equiv_p$, which is defined on the set N of document nodes such that for any two $n_1, n_2 \in N$, $n_1 \equiv_p n_2$ *iff* n_1 is accessible via the same simple path as n_2. In other words, each class $[n]_{\equiv_p}$ is just the target set of some simple path in the database. In the schema tree, the node corresponding to $[n]_{\equiv_p}$ is reached by that same path, and annotated with references to all members of $[n]_{\equiv_p}$. We take advantage of this fact when using the schema tree as an index, since it guarantees that any schema path leads to the right references, i.e. that the index is exact. It is easy to prove now that any document node is referenced by exactly one simple path in the schema tree. Since in a tree document no two paths share the same target set, and the $[n]_{\equiv_p}$ are disjoint (forming a partition of N), no document node is referenced more than once. On the other hand, there must be a schema path for each of the equivalence classes, whose union covers all nodes in N (again this follows from the definition of a partition). Hence there are references to all document nodes in the schema tree.

Now that we have seen how a schema tree is defined and used as structure index, we shall also mention its shortcomings. First, the simple matching procedure we sketched above becomes more complex for query paths with regular features, such as wildcards or disjunctions. As a special case, evaluating relative path queries requires a full search throughout the schema tree. Second, the index makes no assertion about hierarchical relationships between individual document nodes. For instance, there is no way to locate all nodes with an incoming y edge at any level in the database, let alone those which are children of a given document node. In particular, from the fact that in *(b)* there is an edge labelled y between n_s and n'_s, one cannot deduce that in the database the nodes &3 and &7, which are referenced by n_s and n'_s, respectively, are linked in the same way.

The reason is that although annotated with document node IDs, the schema as a structural summary abstracts from individual nodes in the database. An edge labelled l between schema nodes n_s and n'_s indicates that there is at least one l edge in the database linking one of the nodes referenced in n_s to one of those referenced in n'_s. Only when a schema node points to a single document node, one can be sure that all nodes referenced in the subtree below are linked to this document node by their respective label paths. As this is the case for the root, it is guaranteed that if in the schema tree there is a path p from the root to a node containing the node ID i, then in the database tree the node i can be reached via p from the root.

A third restriction of the schema tree is that being based on the equivalence relation $\equiv_p$, it is only defined for tree databases. When cycles and node sharing are admitted in the documents, there is no single schema definition reconciling its three benefits, namely that it summarizes the database in a compact tree which also serves as an exact structure index and can be navigated deterministically. In fact, for graph databases there is a trade-off between conciseness, exactness, and determinism. When the structure index is required to be exact, only non-deterministic navigation can stop it from growing exponentially in the size of the database.[10]

Accordingly, there are two major approaches to exact indexing of graph databases in the literature. While for tree databases both are equivalent to the schema tree described above, they differ in the way paths from arbitrary graph-shaped documents are represented—one is deterministic and the other non-deterministic. We discuss the first index, called *strong DataGuide*, in section 6.4.3. Just like the schema tree, it contains each simple path from the database exactly once, allowing for deterministic index navigation. To this end, $\equiv_p$ is replaced by an equivalence relation $\equiv_t$ on label paths. As a drawback of determinism, however, strong DataGuides incur the storage explosion just mentioned, because sometimes node references and in the worst case entire subgraphs need to be duplicated.

The second approach, known as *1-Index* (a specialization of the *T-Index* presented in section 6.4.4), sacrifices deterministic navigation to achieve linear storage consumption. Every document node is guaranteed to be referenced only once, whereas a label path from the database may appear multiple times in the index. While in theory the 1-Index is based on $\equiv_p$ generalized to graphs, a refinement $\approx_{b_1}$ is used instead to make the index creation more efficient.

The other navigational indices we present, namely the BUS index (section 6.4.1), the Signature File Hierarchy (section 6.4.2), and the IndexFabric (section 6.4.5), are restricted to tree databases, where the two approaches coincide.

4.3 Normative schema

In the previous section we have introduced the notion of a descriptive schema, which is a concise structural summary of the database at a given point in time. As shown for the special case of tree-shaped documents, a descriptive schema is derived from the database in its current state. When the database content is modified, the schema may need updating to reflect the changes. Thus the schema is dependent on, and restricted by, the database content, whereas the inverse is not the case.

[10] A similar trilemma involving conciseness, precision, and expressive power of a filtering technique is observed at the end of section 3.3.

However, the term "schema" is often used in a *normative* sense, too. In particular, the schema of a relational or object-oriented database management system is used to define constraints on the content of the database. For instance, the datatypes of fields in a table are defined in the database schema, just as possible restrictions regarding uniqueness or null values. There are normative schema formalisms for the semistructured data model, too. Among the most well-known are *DTDs* and *XML Schema*. Although differing in their expressive power, both serve to define classes of documents based on structural and content-related properties.

In general, normative schemata specify conditions to be fulfilled by data records claiming to belong to a certain class (or relation). As such they strive to cover explicitly all aspects which distinguish the members of the class, even though not all of them need to be represented in the current state of the database. For instance, a database schema may mark a certain attribute a of a relation R as accepting null values, while in the database all tuples in R currently have a non-null value at that position. When new records are added, however, the additional information is needed to find out whether those with a set to `null` are acceptable. In a descriptive schema, this information would be lost. Similarly, descriptive schemata often fail to recognize datatypes because the type information given implicitly in the data is ambiguous. Obviously they cannot be used for validating new data, since they adapt and hence conform to them by definition.

On the other hand, in many situations data come without an explicit definition of the class they belong to. First, its characteristics may be hard to identify properly. But even with a clear idea in mind, creating a correct class specification can be a very complex task, regardless of which formalism is used. As mentioned in section 4.1, the semistructured data model has been designed to formalize those characteristics which are implicitly available from the given data even when the full class specification is unknown.[11] Descriptive schemata exclusively rely on these implicit properties, which are made explicit in the mark-up. Hence they are available even when a normative schema is missing. In this work we only consider indexing approaches limited to descriptive schemata because of their broader range of applicability. The only exception is an optimization technique for the Context Index described at the end of section 6.3.2.3. Other investigations in optimizations based on normative schema information include e.g. [PH01] or the doctoral poster presentation about "SUPEX: A Schema-Guided Path Index for XML Data" at VLDB 2002.

4.4 Indexing and querying structured documents

Unlike indexing methods known from traditional Information Retrieval (see the previous chapter), indices for structured documents need to cope with both structure and content. However, few of them combine the two aspects in a single data structure, like e.g. the IndexFabric (section 6.4.5). Most approaches use separate *structure* and *content indices*. In particular, the text index and value index presented in section 6.2 often complement navigational structure indices (section 6.4). In these cases, both index structures are looked up separately, and their results need to be joined in a third step. The join may involve further access to the structure index to make sure an occurrence retrieved by the content index can be reached via a database path found by the structure index. (In section 6.4.1, we discuss the BUS index which accomplishes this join without accessing the structure index.)

The exact join procedure depends on how occurrences are actually located by the content index. [LYYB96] compares five approaches to indexing keywords in structured documents: there may either be a single posting for the node where the keyword occurs, or an additional posting for each of its ancestors, or else one posting per level and keyword. Alternatively, one may index the highest possible node in the document hierarchy whose descendants all contain an occurrence of the given keyword. While this approach reduces the number of postings, it involves navigating the structure index, as described above. As an extreme case, keywords may only be associated

[11] From this point of view, mark-up in semistructured data can be regarded as an ad-hoc definition of a minimal subclass comprising the given instances of a more general class of data.

with the document root. This means the retrieval granularity (see section 5.2) is not finer than document level, like in classic Information Retrieval with unstructured documents.

As an alternative to dedicated structure indices for semistructured data, one may take advantage of the sophisticated indices for relational databases. [DFS99] and [FK99] examine how to map semistructured data to a relational database schema. However, query evaluation with such relational systems may involve numerous joins and hence be inefficient.

As mentioned in section 2.2, the query formalisms employed by different retrieval systems are beyond the scope of this work. Suffice it to say that there are several query languages for semistructured data which have been designed for use with either specific systems – e.g. LOREL for the LORE system (section 6.4.3.3) and StruQL for the Strudel system ([FFLS97], [FFK+98]) – or for specific data models or representation formalisms, e.g. SGQL for the SGML-based ELF data model ([AMFL+95]), and XQuery as well as XPath for an XML-based data model proposed by the W3C ([FMN02]). According to their respective expressivity, applications of these languages range from simple or regular path selection to complex querying and transformation of documents. While the elementary indices from section 6.2 only cover the simplest features (basically label and keyword tests), the more complex path look-up and navigational approaches (sections 6.3 and 6.4, respectively) mostly handle parent/child and sometimes ancestor/descendant relationships, too. Other query constructs involving sibling order, regular expressions on node labels, or aggregate functions are usually not supported by the index structures.

Chapter 5

Comparison criteria

This chapter discusses the most important features of indexing approaches for semistructured data. The later comparison and classification (see the next chapter, and table 6.2.2 on page 41 for an overview) is based on the following concepts.

5.1 Data representation

5.1.1 Index structure

The indexing approaches described in this work use either tables or directed graphs as main data structures. A graph-based index may be further restricted to tree shape. Note that we only consider the conceptual structure of an index, which can be implemented in various ways. For instance, an implementation may use tables to represent a tree, or associate a tree with tables to represent a graph. Moreover, primary and secondary indices may differ in structure; e.g. a typical graph-based structure index is used with a secondary content index and possibly a parent/child index, both of which are table-based. Text and content can also be incorporated into the same index structure, as the IndexFabric shows (section 6.4.5).

Another aspect regarding the index structure is whether it contains the entire database information in a form appropriate for fast query evaluation, or whether it represents only part of it in a more compact manner. For instance, the parent/child index (section 6.2.3.5) can be considered an alternative representation of the database without loss of information, whereas the DataGuide (section 6.4.3) misses out hierarchical relationships between individual nodes in the database, and hence is more compact.

5.1.2 Database structure

As described in the previous chapter, structured documents can be represented as directed graphs. Usually the graphs of all documents in the database are subsumed under a common database root to form a single *database graph*, which is then indexed. Alternatively, the individual *document graphs* may be indexed separately, as e.g. with the BitCube (section 6.3.3) or the BUS index (section 6.4.1). Note that other indexing approaches may easily be modified to handle such a *multi-graph database*, too. Normally one only needs to introduce document identifiers to be returned along with the results of each index look-up.

Not all indices support arbitrary graph-shaped documents. Some may be restricted to directed acyclic graphs, while others can only handle tree documents. The structure of the documents in the database sometimes has an impact on the overall performance, as the worst-case storage complexity of the DataGuide with graph-shaped documents illustrates. Another structural constraint on the database concerns mixed content. While most approaches suppose document nodes to contain either atomic text or child nodes, but not both, some of them might be adapted to support mixed content.

5.2 Index interface

Section 2.3 describes look-up patterns, i.e. functional specifications of the various mappings a given index realizes. The set of look-up patterns associated with an index makes up its *index interface*, summarizing what is given and what is retrieved in a look-up to this index. In particular, the output of the look-up patterns indicates the *retrieval granularity* of the index in question: when document IDs instead of node IDs are returned, query results are only located at the document level like in traditional Information Retrieval. Note, however, that the index interface does not reveal the retrieval precision or recall, just as the input and output parameters of a procedure say nothing about the algorithm it implements.

Throughout this work, the index interface is presented in a separate subsection for each of the approaches to facilitate comparison. The interface specifications are also included in the synopsis on page 41.

5.3 Path templates

Some of the navigational indexing approaches presented in this work support a selection of certain query classes for privileged evaluation. The idea is to represent all paths matching any of these privileged queries in a more compact manner, which both speeds up retrieval and saves storage. Other paths may even be excluded from the index altogether.

Privileged query classes are defined using *path templates*. In their most general form, which is introduced in [MS99b] for the T-Index, these are meta-path expressions $\mathsf{T}_0\, x_0 \ldots \mathsf{T}_k\, x_k$. For each $0 \leq i \leq k$, T_i is either a constant regular path expression (including wildcards) or one of the *placeholders* $\boxed{\mathsf{P}}$ and $\boxed{\mathsf{F}}$. $\boxed{\mathsf{P}}$ matches any regular path expression, whereas $\boxed{\mathsf{F}}$ can be regarded as a single-step $\boxed{\mathsf{P}}$ for our purposes.[12] Constant regular path expressions need to appear unchanged in any query instantiating the template. The x_i are *variables* bound to individual document nodes during retrieval, as we shall see below.

It is important to notice that a path template containing any of the placeholders $\boxed{\mathsf{P}}$ and $\boxed{\mathsf{F}}$ describes a class of regular path expressions rather than a class of label paths, and is in this sense a higher-order path expression. For instance, consider the path template /book/chapter $x_0\,\boxed{\mathsf{P}}\,x_1$ and the database shown in figure 5.1. Queries matching this template include (among others) the path expressions `/book/chapter` x_0 `section` x_1, `/book/chapter` x_0 (`section`|`paragraph`) x_1, and `/book/chapter` x_0 `_*` x_1, but not `/book` x_0 `_*` x_1. The document paths matching at least one of these privileged queries are /book/chapter, /book/chapter/section, and /book/chapter/paragraph, but not /book/appendix/paragraph. Retrieval hits are k-tuples of document node IDs, where k is the number of variables in the path template. To produce such a hit, each variable x_i is bound to a document node matching the part of the query which corresponds to T_i in the template. For the three matching document paths just mentioned, the corresponding bindings are $\{x_0 = \&2,\ x_1 = \&2\}$, $\{x_0 = \&2,\ x_1 = \&3\}$, and $\{x_0 = \&2,\ x_1 = \&4\}$, respectively.

To enhance the evaluation of privileged queries, those path fragments corresponding to a constant regular path expression in the template (and hence in all instantiating queries) appear as a single edge in the index graph. For example, given the template /book/(chapter|appendix) $x_0\,\boxed{\mathsf{P}}\,x_1$, the document paths /book/chapter/section, /book/chapter/paragraph, /book/appendix/paragraph can be represented symbolically as /S_0/section and /S_0/paragraph, respectively. Before evaluating a query expression, the prefixes `/book/chapter` and `/book/appendix` (if present) are replaced by /S_0. Thus all privileged queries retrieve the possible binding for x_0 in a single navigation step, regardless of how complex the original regular path expression was. By contrast, template placeholders span an ordinary subgraph in the index because they admit many different path expressions, which in general cannot all be represented symbolically. Further examples of path template indices can be found in the section about T-indices (section 6.4.4).

[12] Actually $\boxed{\mathsf{F}}$ matches Boolean formulae over basic predicates including, but not restricted to, label tests.

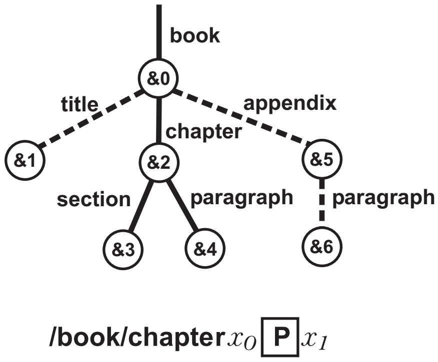


Figure 5.1: Path templates

Identifying the document paths which match a constant regular path expression actually is a kind of query pre-evaluation. As an extreme case, consider a template consisting entirely of constant regular path expressions, like e.g. the IndexFabric's *refined paths*. Obviously such a template can be instantiated by exactly one path query (which in turn may be matched by multiple document paths, of course). The resulting template index, if built over privileged paths only, degenerates to a chain of symbolic edges, which can be replaced by a single one representing the whole query expression. This path of length one then leads directly to all hits for the only privileged query.[13] However, unless exactly this path expression is looked up very often, the full pre-evaluation is of limited use because for more specific queries subsumed under the template, the precomputed result contains too much recall noise. In other words, usually there is no point in precompiling hits for queries with low selectivity. On the other hand, caching the results for many rare and highly selective queries is not affordable.

All document paths relevant to a privileged query are guaranteed to be represented in the index, albeit in symbolic form. Yet the treatment of other paths differs among template-based approaches, depending on whether they tend to favour more general or more compact index structures. The IndexFabric (section 6.4.5) follows the first strategy, indexing all document paths simultaneously, whether privileged or not. The T-Index (section 6.4.4), on the contrary, excludes paths irrelevant for privileged queries, at the risk of being useless for some queries not compatible with the path template. Figure 6.16 on page 79 shows two T-Indices for the same database, but with path templates of increasing selectivity. The resulting storage reduction is obvious.

5.4 Navigation

Navigational indices (section 6.4), which are based on directed graphs, are navigated during a look-up to match the structural part of the query. The term "index navigation" only refers to this evaluation internal to the index engine, but not to navigational features of the external query language (e.g., the `ancestor`:: and `following`:: axes in XPath). Navigation taking place in the index engine and the query engine must not be confused, analogously to the distinction between user queries and index queries, or query plans and look-up strategies (see section 2.2).

Similar to the four query execution strategies described in [MWA+98], there are four different kinds of internal index navigation: *bottom-up*, *top-down*, *hybrid*, and *inside-out*. Top-down navigation follows the edges in the index graph according to their native direction, i.e. from the root down to the leaves. This is the most common way to use a graph-based index. Navigating

[13] The path can be regarded as a materialized view defined on the set of document node references.

bottom-up means climbing up the hierarchy in the reverse direction. Hybrid navigation involves a top-down search followed by a bottom-up search from the leaves such that both paths meet at the result node. This approach is taken by the BUS index (section 6.4.1). The last possibility, inside-out, is rather theoretical; one could imagine starting from a node somewhere in the graph to match parts of the query in top-down and bottom-up direction. Depending on the underlying index structure, some of these navigation schemes may require secondary indices, e.g. for bottom-up navigation in a graph without backlinks. As with the index structure discussed in section 5.1.1, we only consider the conceptual navigation of an index, ignoring that sometimes it is carried out without actually accessing the index structure. When using the BUS index, e.g., the bottom-up step and join of the two paths can be simulated by a calculation on the node IDs.

5.5 Node identification

Indices with a retrieval granularity at node level need to identify individual document nodes unambiguously. Unique node identifiers are used with a bijective mapping $id : NodeIDs \longrightarrow DocNodes$, where *NodeIDs* is the set of node IDs and *DocNodes* is the set of document nodes (see table 2.1 on page 12). Although a node ID can be an arbitrary token in principle, it may also serve to encode structural relationships. Several methods have been proposed for this *structural node identification*, which are discussed in the rest of this section. They are all suitable for both the database and the schema graph (see e.g. a possible enhancement of the BUS index in section 6.4.1.5), and can be combined with a broad range of indexing approaches. Those encoding parent/child relationships are useful when joining the results of multiple look-ups, while the ones concerned with ancestor/descendant pairs make the evaluation of regular path queries more efficient. As a drawback, however, all of them are restricted to trees. Furthermore, inserting a node into the database may cause numerous IDs in the database to be invalidated, preventing incremental index updates.

5.5.1 Pre-/postorder encoding

The *pre-/postorder encoding* [Die82] encodes ancestor/descendant relationships between document nodes. A node identifier is a pair (*pre*, *post*) where *pre* is the preorder value and *post* the postorder value of the node in question. Obviously this approach only supports tree databases. It can be shown that a node n_2 is a descendant of another node n_1 *iff* n_1's preorder value is less than n_2's, and n_1's postorder value is greater than n_2's. While this encoding allows for ancestor/descendant tests in constant time, it requires numerous node identifiers to be recomputed when the database structure changes. The next subsection introduces a more flexible alternative. Both can be used to locate ancestor/descendant pairs with a fixed number of nodes in between by simply including the tree level in each node identifier tuple. As a special case, parent/child pairs can be handled as ancestor/descendant pairs with no intermediate nodes.

5.5.2 Interval encoding

The *interval encoding* ([LM01]) is based on the preorder traversal of the database graph, too. Like the pre-/postorder encoding, it is dedicated to ancestor/descendant relationships and restricted to tree databases. Instead of the postorder value, each node is annotated with the offset of the greatest preorder value among its descendants, as shown in figure 5.2 *(a)*. Hence for any node n_1 with the ID (pre_1, $size_1$), $size_1$ denotes the size of the interval of node IDs spanned by the subtree rooted at n_1. A node n_2 is a descendant of n_1 *iff* n_2's ID lies between pre_1 and $size_1$.

Although an interval-encoded ID consists of two components like a pre-/postorder ID, database updates, especially insertions, cause much less computational effort because the second component, the size of the ID interval, only changes for the ancestors of the inserted node. To avoid incrementing the preorder values of all document nodes following the inserted one (in preorder traversal), [LM01] proposes to choose the size of any interval greater than the actual number of

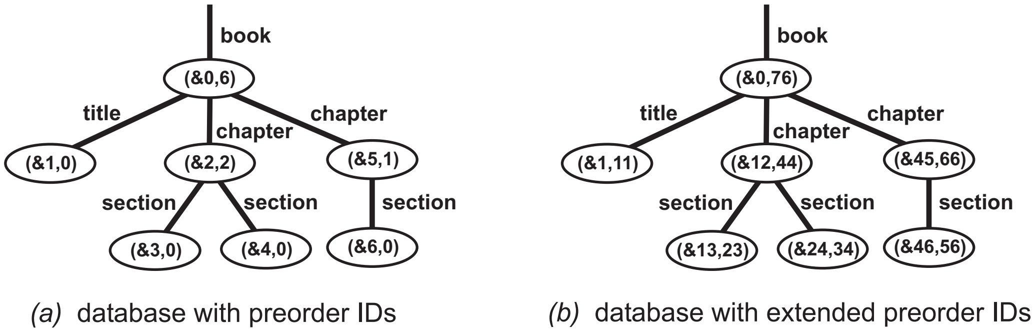


Figure 5.2: Structural node identification with the interval encoding

nodes in the corresponding subtree, a technique called *extended preorder*. Thus the index supports a number of incremental updates before it must be rebuilt from scratch. Figure 5.2 *(b)* depicts the same database as in *(a)*, but with intervals extended by a value of 10. Similar to the pre-/postorder encoding, storing level information in the node IDs enables the interval encoding to select descendants at a specific distance from an ancestor node.

5.5.3 Virtual Nodes

While the former two approaches to structural node identification address the ancestor/descendant relationship and are based on depth-first tree traversal, the Virtual Nodes scheme ([LYYB96]) takes a breadth-first approach to encoding parent/child pairs. Obviously, it is restricted to tree-shaped databases, too. The document nodes are numbered as if the database tree were a complete tree of uniform arity a, i.e. as if any inner node in the database had exactly a children and all leaves were on the same level.[14] Any tree database can be regarded as a full a-ary tree for an appopriate a when missing nodes are inserted "virtually", i.e. reserved an ID although they do not appear physically in the database. Figure 5.3 illustrates a database with $a = 3$. Non-existent or "virtual" document nodes have dashed edges. Note that the Virtual Nodes as described in [LYYB96] requires the numbering to start with 1, not 0.

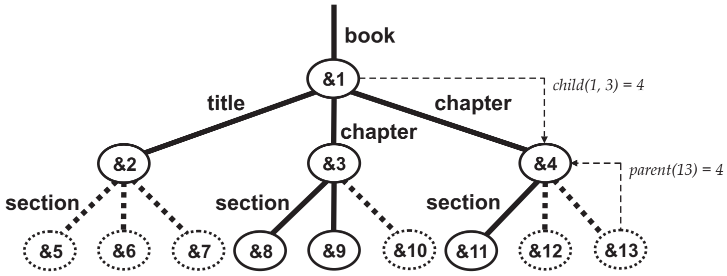


Figure 5.3: Structural node identification with Virtual Nodes

Like the extended preorder described in the previous subsection, the Virtual Nodes encoding scheme supports incremental updates for a certain number of node insertions before a complete

[14] Actually the rightmost a-tuples of nodes on the lowest level may be absent, too, i.e. there may be leaves on the next to last level.

reindexing is required, thanks to the uniform arity. Moreover, the parent and child IDs of any document node can be computed from its own identifier in constant time. The two functions $parent : NodeIDs \longrightarrow NodeIDs$ for calculating the parent ID and $child : NodeIDs \times \mathbb{N} \longrightarrow NodeIDs$ for calculating the ID of the k-th child, $k \in \mathbb{N}$, are defined as follows:

$$
\begin{aligned}
parent(i) &= \left\lfloor \frac{(i-2)}{a} + 1 \right\rfloor \\
child(i, k) &= a(i-1) + k + 1
\end{aligned}
$$

5.5.4 Sibling numbers

We briefly discuss a fourth approach to structural node identification here although it cannot be used to enhance joins or the evaluation of regular path queries. However, node IDs based on *sibling numbers* ([SDAMZ94]) encode structural information, too. In order to identify a document node unambiguously, the root path leading to this node is represented as follows: for each node on the path, the number of preceding siblings of the same type (i.e. with the same incoming label) is computed. This is its sibling number. The incoming labels of all nodes on the root path together form a label path (see definition 8 on page 24). Each label is annotated with the sibling number of the node it leads to (it may be omitted for singleton siblings). The resulting *label/sibling path* is a kind of position path (see definition 11 on page 24). Figure 6.2 on page 48 shows a small database *(a)* and the corresponding label/sibling paths *(c)*. Obviously, sibling numbers must be used with a data model which preserves the order among sibling nodes, and they only support tree databases.

As can be seen in the figure, children of the same parent node with identical labels cannot be confused thanks to the sibling numbers. For instance, the two chapter nodes directly below the document root are referred to as /book/chapter[0] and /book/chapter[1], respectively. A simple induction over the length of a label path proves that this is sufficient to distinguish any two nodes in the document.

Proof. (Unique node identification with label paths and sibling numbers) Obviously sibling numbers guarantee disambiguation for paths of length 1. Let n_1 and n_2 be document nodes with a label path of length l_1 and l_2, respectively. If $l_1 \neq l_2$, i.e. when they are on different levels in the tree, they cannot have the same path anyway. Now assume they share the same level $l_1 = l_2$. Then their parent nodes may be either identical or not. If they are, sibling numbers again suffice to distinguish between the two, as in the base case. Otherwise, we know from the induction hypothesis that their parents' paths differ. Since they are prefixes of n_1's and n_2's path, respectively, the two do not have the same path either. □

As far as database updates are concerned, sibling numbers prove more flexible than the preceding approaches because sibling numbers need only be recomputed for the siblings of the inserted node. Hence incremental index updates are fully supported. On the other hand, not only node insertions but also removals may cause sibling numbers to be invalidated. Section 6.3.1.4 compares the update behaviour of label/sibling paths to non-structural node identification.

5.6 Index update

The previous section illustrates how different database settings may affect the updating behaviour of an index. While some index structures support incremental updates, reflecting changes to portions of the database without side-effects to parts of the index which are not directly related, others may need to be rebuilt completely under certain circumstances, e.g. when a new node is inserted at a specific position. As a consequence, not all indices are well-suited for dynamic databases (see section 2.1). A dedicated subsection analyzes updating for each of the indexing approaches presented in this work.

5.7 Storage and performance

The different index structures vary widely in their storage consumption. Elementary indices (section 6.2) and path look-up indices (section 6.3) are usually small enough to fit main memory. Only the BitCube (section 6.3.3) tends to grow significantly with larger databases and therefore comes with a dedicated paging strategy. Navigational index structures (section 6.4) are often too complex to be held in main memory. To capture the storage requirements of an index, the notion of *storage overhead* is defined:

Definition 18 *(Storage overhead) The* storage overhead *of an index structure is the proportion of its own size to the size of the database,* $\frac{\textit{index size}}{\textit{database size}}$.

Besides the sheer size of an index, its organization of course has a significant impact on the *retrieval latency*, i.e. the time needed to return the results. The concept of *locality* touches the question whether related data items are kept spatially close to each other, and fetched from secondary storage devices at once because they are all likely to be needed in a short period of time. Since the query engine is supposed to locate all matches, the closer they are the better. For instance, the BitCube may be split into multiple small index structures according to the principle of locality. Ideally, when all documents containing a desired keyword or path are concentrated in one or a few BitCubes, the number of index structures to be accessed by the query engine is reduced, which is crucial for an acceptable retrieval latency.

While at query time the retrieval latency is the main performance indicator of an index, in the "off-line" periods it is the time it takes to build (or rebuild) the index. Here the notion of *throughput* serves as a performance measure:

Definition 19 *(Throughput) The* throughput *of an index structure is the number of database objects which can be indexed in a certain time,* $\frac{\textit{number of indexed nodes or links in the database}}{\textit{time unit}}$.

Both storage consumption and retrieval latency may be described in terms of a theoretical complexity analysis. Where complexity results are given, we included them in the presentations of the individual indexing approaches.

Chapter 6

Indexing approaches

6.1 Overview

The indexing approaches reviewed in this work can be divided into three classes: *elementary indices* (section 6.2), *path look-up indices* (section 6.3), and *navigational indices* (section 6.4). In contrast to [MS99a], whose classification of indexing approaches is based on the query execution strategy only (*bottom-up* vs. *flexible*; guided by the query's *structural* or *syntax tree*), our main criterion is the structure of the index, which also affects its interface. After a short description of the three classes, table 6.2.2 shows the most salient features of all indexing approaches in a synoptic view. The individual indices are presented in more detail throughout the rest of this chapter.

Elementary indices, as the most basic kind of index for structured documents, are simple look-up tables. Ignoring the hierarchical concepts of a path, tree, or directed graph, they index individual document nodes by either their type (i.e. the label of their incoming edge(s), see section 4.1), or content, or parent and child nodes. Path queries are not directly supported; instead look-up results from multiple elementary indices may be combined by the query engine to answer structural queries. In such a scenario, elementary indices are used to identify a set of document nodes for every single query step in isolation. The query engine then joins the sets to reconstruct document paths which match the whole query. Obviously processing complex structural queries or combined structure and content queries in this small-step manner requires numerous joins. Hence in real-world retrieval systems elementary indices are mostly used as secondary structures in conjunction with the more sophisticated navigational approaches. This involves fewer index look-ups, such that some of the work-load is shifted from the query engine to the index engine, as claimed favourable in chapter 2. Elementary indices are equally well suited for bottom-up and top-down query plans (see section 2.2).

Path look-up indices, as their name suggests, use entire document paths instead of nodes as basic structural units in the retrieval process. They differ from elementary indices in that they store each node's structural context, i.e. its path from the document root, in the index rather than having it reconstructed by the query engine at evaluation time. Table-based like elementary indices, however, the indices in this second class only support *atomic look-ups*: they retrieve the label paths and possibly the IDs of all nodes where a given keyword occurs (with some variations for the BitCube; see section 6.3.3 for details). The evaluation of structural queries given these label paths is part of the query engine exclusively. More complex index look-ups, including regular path or tree expressions, generally involve multiple internal steps and a compositional path concept, which in turn requires data structures with a finer granularity than path tables. Theses issues are addressed by the approaches in the third class, which also avoid redundant storage of paths. All path look-up indices are bound to bottom-up query plans: designed for classifying keyword occurrences based on their structural context, they are of limited use in a top-down scenario where the structural part of a query is evaluated before keyword look-ups. (This does not apply to the

BitCube, however, which supports path queries as well as keyword queries.)

Navigational indices use directed graphs as their main data structure, thus representing the document paths in a compositional, i.e. stepwise, and compact way. As with path indices from object-oriented database systems, each path in a navigational index can be considered a precomputed join of multiple look-ups in indices from the first two classes, which is evaluated by simply following the path top-down (note, however, that there is a variety of strategies for internal navigation, as explained in section 2.2). Navigational indices mostly rely on the descriptive schema (see section 4.2) as a more compact representation of the database graph. Therefore they do not duplicate identical path prefixes, contrary to path look-up indices. Except for the T-Index (section 6.4.4), which may omit entire document paths depending on its underlying path template, all of them can handle arbitrary regular path queries in principle, although this may require a full index scan in the worst case.

6.2 Elementary indices

6.2.1 Abstract

In this section we review simple indices for structured document queries, which are based on tables. Since they all ignore the notion of a path or even higher concepts, their exclusive structural unit is the element, or labelled node.

Perhaps the most basic index is the *text index*. Known as *inverted file* in Information Retrieval, it retrieves occurrences of a given keyword in the database. The *value index* is similar to the text index, but with data type-specific matching operations. Both can be restricted to selected element types, and to direct textual containment which avoids redundancy. [SDAMZ94] mentions two more elementary indexing techniques, the *label index* and the *Combined Vocabulary*. The former, locating nodes by their label, is complementary to the text index, whereas the latter is a combination of both text and label index. One can certainly disagree about the *parent/child index* being an index in its own right. We subsume it here because it shares some features with the other elementary indices, and serves as secondary index in some of the more complex approaches described in the following sections. Elementary indices can also be used as primary index structures in a retrieval system. As an example, the *XISS system* is presented along with its three algorithms for joining look-up results from multiple elementary indices.

6.2.2 Index interface

text index	$\mathit{Keywords} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$
value index	$\mathit{Predicates} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$
label index	$\mathit{Labels} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$
Combined Vocabulary	$\mathit{Labels} \times \mathit{Keywords} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$
parent/child index	$\mathit{NodeIDs} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$

Table 6.1: Look-up patterns for elementary indices

Look-ups on any of the five indices always return a set of node IDs indicating which document nodes match the given selection criteria. (Recall from section 2.3 that $\mathcal{P}(X)$ denotes the power set of a set X.) The text index retrieves nodes based on their textual content, while the label index selects nodes of a given type (i.e. with a given label). The Combined Vocabulary, as a combination of both, takes a label and a keyword as input and returns the IDs of all document nodes of the corresponding type which contain the given keyword.

cl.	index name	data represent.			index interface (I=input, O=output)										templates		internal navigation			ID enc.	upd.	storage		section
		index struct.	DB struct.	multi-doc.	node ID	doc. ID	pos.	cont.	lab.	simple path	reg. path	tree	sig.	wght.	place-hold.	atom.	bot.-up	top-down	hybr.	struct.	incr.	type	pag. strat.	
elementary	text index	table	graph		O			I													yes	1, 2		6.2.3.1
	value index	table	graph		O			I													yes	1, 2		6.2.3.2
	label index	table	graph		O				I												yes	1, 2		6.2.3.3
	Combined Vocabulary	table	graph		O			I	I												yes	1, 2		6.2.3.4
	parent/child index	table	graph		I,O																yes	1, 2		6.2.3.5
path look-up	Elt. Locator Scheme	table	tree				O	I		O												1, 2		6.3.1
	CIS index	table	tree		O			I		O												1, 2		6.3.1
	Context Index	table	graph		O			I					I								yes	1, 2		6.3.2
					O				I				I											
	BitCube	table	tree	yes		I		O		O											yes	1, 2	yes	6.3.3
						O		I		O														
						O		O		I														
						I		I		O														
						I		O		I														
						O		I		I														
navigational	BUS	tree	tree	yes	O	O		I		I				O					yes	virtual		2		6.4.1
	Signature File Hierarchy	tree	tree		O					I			I					yes			yes	2		6.4.2
					O							I	I											
	DataGuide	graph	graph		O					I								yes			yes	1, 2		6.4.3
	T-Index	graph	graph		O					I					yes	yes		yes			yes	1, 2		6.4.4
					O						I													
	IndexFabric	tree	tree		O					I						yes		yes			yes	2	yes	6.4.5

Table 6.2: Synopsis of indexing approaches

The value index behaves just like the text index, except that it accepts general predicates over specific data types, like e.g. numerical bounds or special text matching expressions (see below). The parent/child index is the only one to take a node ID as input. Details about the various kinds of index are discussed in the next subsection.

6.2.3 Details

6.2.3.1 Text index

The *text index* maps a given keyword to all document nodes containing this keyword. It may be restricted to direct textual containment (see definition 16 on page 25), i.e. consider only the lowest nodes in the database tree containing the keyword, but not their ancestors. Text indices supporting indirect textual containment (see definition 17) make sense when there is no additional means to check for an ancestor-descendant relationship between document nodes. Note that they are redundant in the sense that the same keyword occurrence is indexed anew for each node on the corresponding root path.

Since a text index does not help with the structural part of a query, the query engine needs to combine the results from a text index look-up with those obtained from other indices, e.g. a label index (see section 6.2.3.3). When a text index supporting indirect textual containment is used, the set of node IDs returned by the text index is intersected with the one corresponding to the last label in the query path. The IDs in the intersection then belong to nodes of the type required by the query path which contain the keyword either directly or indirectly. If the text index is restricted to direct textual containment, the query engine may either use a parent/child index (see section 6.2.3.5), or rely on ID encoding (like the XISS system described in section 6.2.3.6), or find some other means to check whether an inner document node indirectly contains the desired keyword.

[SDAMZ94] suggests building separate text indices for selected element types. [MWA+98] proposes the same strategy for value indices. In both cases the index interface is left unaffected. The advantages are that smaller indices are more efficient and can be set up selectively for frequently used labels. However, while separate indices make the look-up more efficient when only nodes of a given type are considered relevant, multiple look-ups are required in case the keyword is to be found in nodes of any type. Moreover, redundancy increases when multiple indices are built for the same set of keywords. The Combined Vocabulary, introduced in section 6.2.3.4, addresses these shortcomings.

Text indices, as well as the value and label indices discussed below, can be implemented using standard data structures from Information Retrieval, e.g. Tries (see section 3.4), or approaches from the database community such as the B-tree. Before building a content index linguistic normalization techniques like stemming or stop-word lists (see section 3.2) are often used to reduce the number of keywords to be indexed.

6.2.3.2 Value index

The *value index* ([MWA+98]) is a generalization of the text index featuring additional selection criteria. Usually there are different kinds of selection, each one associated with a predefined data type. The syntax and expressive power of the selection predicates depend on the implementation, of course. Just to hint at how possible predicates might look like, consider a simple value index supporting two data types, `string` and `number`, and predicates of the form $[\theta v]$, where θ is an operator for the type of content expected in this part of the document, say `contains` for `string` and $<$, $>$ for `number`, and v is a value of that type. A look-up for nodes satisfying the predicate $[< \ 5]$ would then return the IDs of all document nodes with numerical content smaller than 5, and [`contains "XML"`] would yield the same results as a text index look-up for the keyword “XML”, plus the occurrences of keywords containing the substring “XML”.

Like text indices, value indices may be restricted to direct containment in order to avoid redundancy. There can also be separate value indices for different element types. [MWA+98] introduces a technique for handling ambiguously typed values. Given a query for the string “05”, e.g., it might be desirable to find not only occurrences of this particular character sequence but also of the numbers 5.0 or 5. This involves multiple value indices dedicated each to a specific *type coercion*, i.e. conversion from a more general type to a more specific one. For instance, there is a value index containing only those strings which can be coerced into real numbers, such as “05” and “2.4e-3”. At query time, the types a given value can be converted to determine which indices must be consulted.

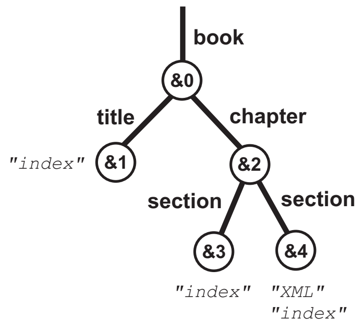
6.2.3.3 Label index

A *label index* retrieves all elements with a given label from the database, i.e. it selects document nodes of a given type. (See section 4.1 for a brief discussion of the element label/node type dualism.) It may be used to match a given label path against the database tree in a step-wise manner, if no higher-level structure index is available.

6.2.3.4 Combined Vocabulary

In section 6.2.3.1 we stated that separate text (or value) indices can be maintained for selected or even all element types in the database. While this saves a join with label index results when a keyword query is restricted to a few element types, multiple index look-ups are required as soon as the type of the node containing the keyword occurrence is not uniquely defined. Furthermore, keeping several indices over the same set of keys is necessarily redundant, regardless of the inherent redundancy caused by indexing with indirect textual containment.

To overcome these drawbacks, a *Combined Vocabulary* ([SDAMZ94]) puts separate text or value indices together while preserving the distinction between different element types. An example is shown in figure 6.1 below. For each keyword, it stores the relevant node IDs as a set of ID lists, one for each element type. Thus an entry in a Combined Vocabulary can be thought of as the concatenation of all entries for that same key in separate text or value indices. As a consequence, one can locate relevant nodes reachable by any of an arbitrary set of labels with a look-up to just one index structure, which is slightly more complex than the separate indices it replaces, of course. In case secondary storage devices are involved, a single index is more likely to be stored contiguously on disk than multiple separate indices, which would reduce response latency.



(a) database tree

"index"	title	&1
	section	&3;&4
"XML"	section	&4

(b) Combined Vocabulary (direct containment)

Figure 6.1: Combined Vocabulary

6.2.3.5 Parent/child index

As discussed in section 6.2.3.1, answering structural queries with elementary indices involves path reconstruction. For two given sets of node IDs, one needs to find the pairs corresponding to parent and child nodes in the database tree. Although these pairs can be retrieved directly in the database, it is more efficient to use a child or a parent index instead. For a given ID of a document node, the former retrieves the IDs of this node's children, whereas the latter locates its parents (a singleton set in case of a tree-shaped database). Both can be combined in a single data structure, of course.

Actually the *parent/child index* is an alternative representation of the entire database tree without labels. As can be seen from its interface, it ignores any abstraction from individual document nodes. Still it offers the main benefit of any index as discussed in section 2.1, providing direct access to the relevant data as opposed to a linear scan of the whole database. This is one reason why we list it as an index here. A second reason is that other, more sophisticated index structures, such as the ToXin index (see the end of section 6.4.3.3), rely on parent/child indices as a secondary data structure.

6.2.3.6 Applications of elementary indices: the XISS system

We shortly review an example from the literature now which illustrates how multiple elementary indices[15] can be combined to answer structural queries.

[LM01] introduces the *X*ML *I*ndexing and *S*torage *S*ystem (XISS), an index engine for queries with regular path expressions. XISS is based on the interval encoding. Recall from section 5.5.2 that the interval encoding scheme uniquely identifies document nodes by their preorder value *pre* in the database tree. Each node spans an interval of preorder numbers, which indicates the maximum number *size* of descendants it can have without reindexing.

The XISS system consists of five elementary index structures, three of which are variations of the ones described above. The last two indices, *Name Index* and *Value Table*, serve technical purposes only, and are therefore of no interest here. Suffice it to say that they allow string constants like labels or textual content to be handled more efficiently. Note that [LM01] does not mention any text or value index, though they should be easy to integrate.

Element Index The Element Index does the same job as the label index from section 6.2.3.3. Multiple document trees are supported in the XISS system, so the input to the Element Index is not only a label but also a document ID. Besides a pair (*pre*, *size*) for node identification, it returns the node's level in the document tree, which is needed for the $\mathcal{EA}$-Join (see below).

Attribute Index The Attribute Index is very similar in structure to the Element Index described in the previous paragraph. The only difference is that its output includes a reference to the attribute's content which can be looked up in the Value Table. It remains unclear, however, why this is not the case for the Element Index.

Structure Index Just like the parent/child index examined in section 6.2.3.5, the Structure Index assists in finding the parent and children of a given document node.[16] Given that the XISS system is restricted to tree-shaped databases (as explained in section 6.2.4), there is at most one parent node for any document node. Among all children (and all attributes, which are distinguished from elements the XISS system), only the first is reported in a look-up. In addition, the first sibling of each node can be located using the Structure Index. Thus it is possible to find all children and all attributes of a given node, albeit with multiple index look-ups.

[15] The join operations described hereafter are applicable to some other index structures, too, as indicated in the respective sections below.

[16] Note that in contrast to our more general usage of this term, "Structure Index" (with captital letters) denotes a particular index structure in the XISS system.

[LM01] describes three join procedures for use with these indices: the $\mathcal{EA}$-Join[17] for finding direct element/attribute or element/element pairs, the $\mathcal{EE}$-Join[17] for indirect element/attribute or element/element pairs, and the $\mathcal{KC}$-Join[17] for computing the transitive hull of a label in a path, i.e. for handling regular path expressions involving one or more repetitions of the same step, like in `section`$^+$. Each join is carried out on node sets N_i containing (*docID*, *pre*, *size*, *level*) tuples, where *docID* is the document identifier of the node in question, *pre* is its extended preorder value, *size* is the size of the interval it spans, and *level* is its level in the document tree.

All three joins are computed in a *two-phase sort-merge* procedure as follows. In the first phase, the nodes in the two sets to be joined (elements or attributes) are grouped by their document identifier (sort), and subsets from either set belonging to the same document are then processed together in the second phase (merge). Here both node sets are sorted by *pre* value and then merged according to the specific join semantics (see the following sections for details). The element and attribute indices are guaranteed to return node sets which are sorted by document ID in the first place and then by preorder value, so both phases operate on sorted sets without additional cost.

$\mathcal{EA}$-Join (parent/child index) The $\mathcal{EA}$-Join is used to join either a set of element nodes and a set of attribute nodes or two sets of element nodes. It returns pairs (n_1, n_2) from $N_1 \times N_2$ where n_2 is a child or attribute of n_1, and thus supports path expressions such as `chapter/section` or `figure/@caption`. The first phase of the computation takes place as described in the previous section. In the second phase, elements from both node sets are checked for parent-child relationship, based on their *level* value. (Recall from section 5.5.2 that using the node level along with the interval encoding, not only the ancestor-descendant relation but also parent-child pairs can be computed in constant time.)

A single pass is required to merge N_1 and N_2, i.e. complexity is $\mathcal{O}(\,|N_1| + |N_2|\,)$. On the one hand, this is due to the fact that the node sets are sorted by *pre* values. Searching for the children of a given node n in N_1, only nodes in N_2 with an *pre* value between n's and the one of n's successor in N_1 need to be considered. N_2 being ordered, this excludes the whole subset of nodes examined for n's predecessors in N_1. Another presupposition for this single-pass solution, however, is that we only care for parent/child pairs in the $\mathcal{EA}$-Join. As can be seen in the next section, indirect ancestor/descendant pairings may entail multiple scans of N_2.

$\mathcal{EE}$-Join (ancestor/descendant index) In contrast to the $\mathcal{EA}$-Join, the $\mathcal{EE}$-Join determines which nodes from an element set N_1 are ancestors (not necessarily parents) of which other nodes from an element or attribute set N_2. Example queries include `chapter/_*/figure` and `chapter/_*/@caption`. The procedure is essentially the same as for the $\mathcal{EA}$-Join, except multiple passes over N_2 may be required in the second phase. This is because a node can have more than one ancestor in the database tree, hence for each member of N_1, all nodes in N_2 need to be reconsidered. This is done in $\mathcal{O}(\,|N_1| \cdot |N_2|\,)$ steps. [LM01] gives a worst case example, which is a database tree degenerated to a linear list of chapter nodes with the last one branching into multiple figure leaves. Finding all chapter/figure pairs obviously has quadratic complexity.

As far as ancestor/descendant pairs spanning an arbitrary number of levels are concerned, a node's depth in the database tree does not matter; all information required is encoded in its *pre* and *size* values. However, the *level* value may be used to compute fixed-length paths matching queries like `chapter/_/_/figure` or `figure/_/@caption`.

$\mathcal{KC}$-Join A third type of query can be handled using the $\mathcal{KC}$-Join ($\mathcal{KC}$ stands for *K*leene *C*losure, which means reflexive-transitive hull). These are regular path expressions involving the $^+$ operator applied to a label, not to the placeholder _ as before. For instance, consider the query `section`$^+$.

[17] $\mathcal{EA}$-Join stands for "element/attribute", and $\mathcal{EE}$-Join for "element/element". As discussed in the following paragraphs, however, the two differ in that they handle either parent/child or arbitrary ancestor/descendant pairs rather than being restricted to element/element or element/attribute pairings. Maybe $\mathcal{PC}$-Join and $\mathcal{AD}$-Join would be more intuitive notations, but to avoid confusing readers of [LM01], we stick with the original names. $\mathcal{KC}$-Join is short for "*K*leene *C*losure".

Given the set N of all section nodes, which can be obtained from the label index (see the beginning of this subsection), the $\mathcal{KC}$-Join returns the subset of $\bigcup_{i=1}^{\infty} N^i$ containing all path fragments of arbitrary length from the database which only contain section nodes. The $\mathcal{KC}$-Join recursively constructs such paths fragments of increasing length from the database tree. Those consisting of just one node are given in N. Longer fragments are built from shorter ones by repeatedly applying the $\mathcal{EA}$-Join. A path $n_i/n_j/n_k$ of document nodes n_i, n_j, n_k e.g. may be obtained by combining the fragments n_i and n_j/n_k, provided the $\mathcal{EA}$-Join of the one- and two-node-path sets computed in previous steps confirms that n_j is a child of n_i. When no new path fragments can be constructed this way, i.e. when there are no more paths ending in nodes with suitable children to extend them, the algorithm stops and returns the union of all sets of path fragments it has constructed so far.

6.2.4 Features

Data representation. Elementary indices can only retrieve individual document nodes. More complex structural units, such as paths or trees, need to be constructed by the query engine using join operations. Therefore the index itself is *flat*, i.e. besides textual containment, it does not encode any structural relationships between document nodes. All indices described in this section have a simple table as main data structure, except the Combined Vocabulary which uses a nested table instead. Since elementary indices ignore structural concepts, they are not restricted to tree-shaped databases, locating nodes in arbitrary graph structures. This also contributes to their broad range of applicability. Multi-graph databases can be supported as described in section 5.1.2.

Navigation. Since elementary indices lack the notion of structural units more complex than nodes, the concept of navigation is meaningless in this context.

Path templates. For the same reason as with navigation, no path templates can be defined.

Node identification. The elementary indices introduced in the beginning of this section do not rely on any special numbering scheme for identifying document nodes. However, the join algorithms used with the XISS system are based on interval encoding(see section 5.5.2). The $\mathcal{EE}$-Join in particular, handling regular path expressions with wildcard components of arbitrary length, requires interval-encoded node IDs to locate ancestor/descendant pairs in constant time.

Index update. Since the elementary index structures described above do not depend on any ID encoding scheme, they are amenable to incremental updates. The XISS system, by contrast, is sensitive to database updates, in the sense that inserting nodes into the database tree may cause identification conflicts with existing nodes. To address this shortcoming, the extended preorder encoding, on which the interval encoding is based, reserves a certain number of node IDs to provide for a graceful update behaviour of the index structures (see section 5.5.2 for details). Nevertheless, while updates after node removal can always be carried out incrementally, adding nodes to the database triggers complete reindexing when the extended interval has been used up.

Storage. A typical elementary index is relatively modest in its space consumption and may well fit main memory even for larger databases. However, being used mostly in conjunction with other (elementary or more complex) index structures, it is likely to be stored on secondary devices anyway. The approaches discussed in this section are all amenable to standard data structures for large amounts of relational data, like the B-tree. No further techniques for secondary storage are described.

6.3 Path Look-up Indices

6.3.1 Annotated path look-up indices

6.3.1.1 Abstract

The *Element Locator Scheme* as proposed in [SDAMZ94] is a simple index structure for retrieving keyword occurrences and their paths in a database. The index structure consists of a single table matching keywords to label paths and sibling numbers for node identification. The *CIS index* ([Meu00]) relies on unique node IDs instead, which may also bear structural information. Being very similar in structure, both are subsumed under the label *annotated path look-up index* here.

6.3.1.2 Index interface

Element Locator Scheme	*Keywords*	$\longrightarrow$	$\mathcal{P}(\,SimplePaths \times PositionPaths\,)$
CIS index	*Keywords*	$\longrightarrow$	$\mathcal{P}(\,SimplePaths \times IDPaths\,)$

Table 6.3: Look-up patterns for Element Locator Scheme and CIS index

There is only one look-up pattern for each of the two indices. Both take a single keyword as input, but no label or path. The reason is that structural query evaluation takes place outside the index and therefore is not manifest in its interface. The Element Locator Scheme returns an absolute label path for each node containing the keyword, along with sibling numbers for those nodes in the path which cannot be uniquely identified by their label only. The CIS index, by contrast, outputs a label and a unique identifier for every node in the root path of an occurrence, whether ambiguous or not.

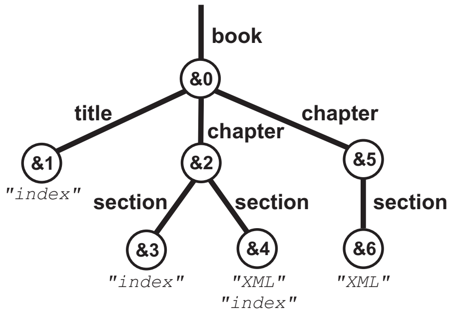
6.3.1.3 Details

Path annotation. Both Element Locator Scheme and CIS index consist of a single look-up table mapping each keyword occurrence to the simple path by which it can be reached in the database. Without further information, however, ambiguities may arise. For instance, there may be multiple nodes with the label path /book/chapter in the database, as shown in figure 6.2. Element Locator Scheme and CIS index differ in the way they distinguish document nodes (inner as well as leaf nodes) which share the same label path.

Relying on unique node identifiers, the CIS index maps keyword occurrences to label/ID paths, as shown in figure 6.2 *(b)*. Since the IDs can be assigned arbitrarily as long as no two nodes share the same identifier, the CIS index is compatible with any numbering scheme. For instance, the query engine may take advantage of the interval encoding (section 5.5.2) to locate ancestor/descendant pairs (see the next paragraph). While the complexity of the operation is still $\mathcal{O}(\,|P_1|\cdot|P_2|\,)$ as shown for the $\mathcal{EE}$-Join in section 6.2.3.6, it may be performed significantly faster in case of deeply nested documents.

The Element Locator Scheme does not require document nodes to bear identifiers, referring to any of them by the number of its preceding siblings with the same label (see figure 6.2 *(c)*). This way of identifying individual document nodes is discussed in section 5.5.4. Of course path annotation with sibling numbers does not preclude the use of additional node IDs. But when IDs are available they can be used for annotation instead. Hence the Element Locator Scheme is recommended primarily for environments where no node IDs are given, e.g. document collections kept in a file system instead of a database.

Path query evaluation. Retrieving keyword occurrences along with their root path, path look-up indices allow to evaluate structural queries at document level much more efficiently than



(a) database

"index"	/book$_{\&0}$/title$_{\&1}$; /book$_{\&0}$/chapter$_{\&2}$/section$_{\&3}$; /book$_{\&0}$/chapter$_{\&2}$/section$_{\&4}$
"XML"	/book$_{\&0}$/chapter$_{\&2}$/section$_{\&4}$; /book$_{\&0}$/chapter$_{\&5}$/section$_{\&6}$

(b) CIS Index for *(a)*

"index"	/book/title; /book/chapter[0]/section[0]; /book/chapter[0]/section[1]
"XML"	/book/chapter[0]/section[1]; /book/chapter[1]/section

(b) Element Locator Scheme for *(a)*

Figure 6.2: Annotated path look-up indices

elementary indices. For a query without indirect containment steps, there is no need for join operations after the look-up to reconstruct a path in the database. Moreover, when the retrieved paths are annotated as with the Element Locator Scheme and CIS index, even queries at node level involving the ancestor/descendant relationship can be processed with fewer index look-ups and less computational effort.

As an example, consider the query `/book["XML"]/_*/figure["index"]`. When only elementary indices are available, the query engine needs to join the results from four look-ups: first, a label index is queried for nodes of type book and figure, returning two sets L_1 and L_2. Then a text index retrieves two sets K_1 and K_2 containing occurrences of the keywords "XML" and "index", respectively. In order to identify nodes satisfying either part of the query, first L_1 is intersected with K_1, and then L_2 with K_2. Finally the query engine must join the two resulting sets to find node pairs matching the query. Certain node identification schemes, such as the interval encoding, provide sufficient structural information to avoid further disk accesses. Without their help, joining the ancestor/descendant pairs requires searching in the database.

When an annotated path look-up index is used, only two look-ups are required: one to retrieve a set P_1 of label/ID paths for the first predicate, and another to get a set P_2 for the second. Obviously only those paths from P_2 which have a /book path from P_1 as prefix match the query. They can be identified in $\mathcal{O}(\,|P_1|\cdot|P_2|\,)$ steps. For instance, using the Element Locator Scheme a pair $(p_1, p_2) \in P_1 \times P_2$ with $p_1 =$/book and $p_2 =$/book/chapter[2]/section[4]/figure would be recognized as a hit. Analogously, $p_1 =$/book$_{\&0}$ and $p_2 =$/book$_{\&0}$/chapter$_{\&52}$/section$_{\&75}$/figure$_{\&90}$ would match in a CIS index scenario.

Path compression. Figure 6.2 above illustrates that all path look-up indices store the upper parts of the label paths redundantly: no advantage is drawn from the fact that in tree-shaped documents, multiple root-to-leaf paths share a common path prefix. In this context, [SDAMZ94] alleges that optimization techniques such as string compression could lead the Element Locator Scheme to more efficient storage. Another solution to the problem of redundancy is suggested by navigational approaches in section 6.4. Their main data structure being a tree or graph instead of a table, there is no need to store the same path prefix more than once. Tree paths may be split into multiple branches at any point, which then physically share the same prefix from the root down to the splitting point. Put aside that additional path compression techniques for the Element Locator Scheme may also be applicable to these index structures, the main advantage of the navigational approach to path compression is that it leaves the look-up latency unaffected. String compression, by contrast, slows down the retrieval process because the original path has to be reconstructed at query time.

6.3.1.4 Features

Data representation. As explained in section 6.1, path look-up indices are by definition flat, keeping hierarchical items such as label paths as atomic tokens in a table. Neither the Element Locator Scheme nor the CIS index is an exception to this rule: as can be seen by their index interfaces, they both lack a compositional representation of label paths. Multi-graph databases can be supported as described in section 5.1.2.

For obvious reasons, path look-up indices are confined to tree-shaped databases, unlike elementary indices (see section 6.2.4) and some navigational approaches (see e.g. section 6.4.3 or 6.4.4). First, prefix matching is much more complicated in graphs because any node may be reachable by an exponential number of paths, or even infinitely many in the case of cyclic graphs. For the Element Locator Scheme it would even be difficult to identify nodes uniquely for this reason. Besides, each keyword look-up might return a multitude of paths for the same document node, which is probably not desirable in most cases.

Navigation. Path look-up indices do not build a tree- or graph-like representation of the database contents, so navigating index structures is impossible.

Path templates. Path templates are by definition inapplicable to keyword-based approaches annotated path look-up indices.

Node identification. The Element Locator Scheme identifies individual document nodes by their sibling numbers, i.e. by counting the preceding siblings of the same type. No node IDs are used and thus the approach is independent of any numbering scheme. The CIS index does require nodes to have unique identifiers, but with no preference for any specific ID encoding. As illustrated in section 6.3.1.3, it may be combined with any numbering scheme to ameliorate query performance.

Index update. Incremental updates are usually spoilt by interdependencies among the document nodes, as they arise e.g. when certain ID numbering schemes are used. The CIS index itself can be updated incrementally. If it is combined with such techniques, however, adding nodes to the database may entail a complete reindexing under certain circumstances, depending on the flexibility of the encoding (see section 5.6 for details).

Although the Element Locator Scheme does not identify nodes this way, it faces the same problem because sibling numbers, which indicate relative positions in the document tree, are associated with ambiguous steps in a label path. Consider the database tree shown in 6.3, for example. Before the update, some nodes in the tree are assigned sibling numbers because there are several other nodes of the same type on their level *(a)*. When a new path is added to the tree, a part of them (in this case all) need to be updated because now there is one more sibling of the same type on this level, incrementing the sibling numbers of all following nodes of this type by one *(b)*. Counting the siblings from right to left obviously does not solve the general problem.

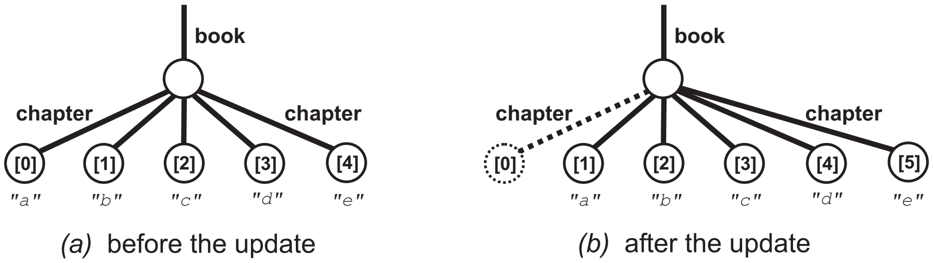


Figure 6.3: Index update with the Element Locator Scheme

Storage. The space required by the look-up table depends mostly on the number of keyword occurrences in the database. Since paths are stored redundantly as explained above, both the Element Locator Scheme and the CIS index might fail to fit main memory for sufficiently large databases, especially when the index entries are of uniform length. When indexing huge amounts of data, navigational indices may prove more favourable. Neither approach presented in this subsection covers storage on secondary devices. Notwithstanding they can both profit from the paging strategies of the underlying relational database system.

In case of the Element Locator Scheme, the average number of sibling nodes of the same type influences the table's space requirements, too. If there are hardly any ambiguities in label paths, few sibling numbers need to be stored. This saves space compared to the CIS index which assigns an identifier to every document node, regardless of any ambiguities. However, the optimization comes at the cost of retrieval efficiency after the look-up. When fetching individual document nodes from the database, giving the sibling number instead of a unique node ID may cause irrelevant pages to be loaded before the right node is located.

6.3.1.5 Remarks

The Element Locator Scheme beats the CIS index in terms of space requirements, all the more when there are few label repetitions on the same level. However, its limited support for incremental updates makes it most suitable for nearly static databases. When the database changes more frequently or regular path queries on deeply nested documents need to be supported by the query engine, the CIS index may be preferred, being compatible with structural node identification and amenable to incremental updates.

In most applications where modest space consumption or support for regular path queries are crucial, however, navigational indices are the solution of choice, some of which also come with dedicated techniques for secondary storage (see e.g. the IndexFabric in section 6.4.5). Even for medium-size databases, the more compact representation of label paths provided by navigational indices outweighs by far the individual benefits or drawbacks of annotated path look-up indices. On the other hand, the latter are easy to implement, since the bulk of the evaluation work is left to the query engine, and fit any off-the-shelf relational database system.

6.3.2 Context Index

6.3.2.1 Abstract

The *Context Index* ([SM99], [MS99a]) is a path look-up index similar to the CIS index (see the previous section), but with an integrated structure filter. It helps to narrow down heuristically the retrieved keyword occurrences to those matching the structural part of the query. In the same fashion, the Context Index can be built as as a label index with structural filtering. In both cases it may serve as a secondary index in navigational approaches.

6.3.2.2 Index interface

Context Index	*Signatures* $\times$ *Keywords*	$\longrightarrow$	$\mathcal{P}($ *NodeIDs* $)$
	Signatures $\times$ *Labels*	$\longrightarrow$	$\mathcal{P}($ *NodeIDs* $)$

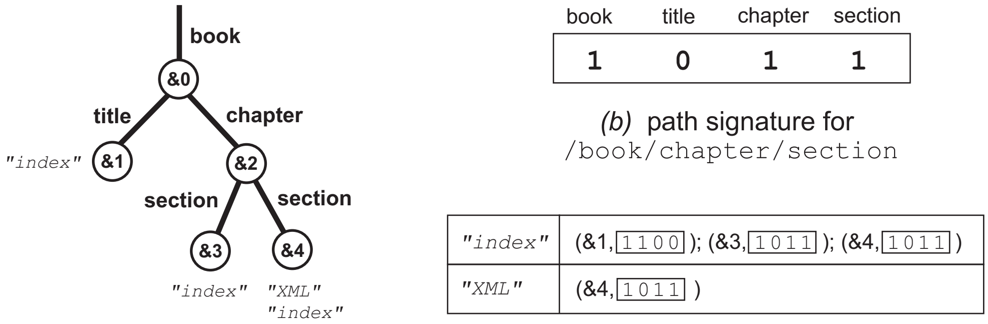
Table 6.4: Look-up patterns for the Context Index

As the two look-up patterns indicate, the Context Index can be used as a text or label index. In either case a path signature is given as input, too, which serves as a filtering criterion (see the next subsection for details).

6.3.2.3 Details

Path signatures. We describe path encoding for the Context Index without storage optimization first. Modifications of the algorithm for space reduction are covered below.

The Context Index owes its name to the fact that occurrences of a given keyword or label are filtered according to their structural context after retrieval. The index is used to retrieve those occurrences reachable via a specific label path (the query path) in the database. This path is encoded as a bit string, which we will call *path signature* here, equal in length to the number of labels in the database. Figure 6.4 *(b)* shows how path signatures are created: a 1 at any given position in the bit string indicates that the corresponding label is part of the query path, while a 0 means it is not.[18] For structural filtering to take place, each posting must bear some information about its structural context, i.e. the simple path leading to this occurrence. [SM99] introduces the term *linear context* to denote this path, which lists the labels of all nodes containing the occurrence, be it directly or indirectly. The linear context is encoded as a path signature, too. Each posting has its own path signature, as can be seen in figure 6.4 *(c)*.



(a) database tree

(b) path signature for /book/chapter/section

(c) Context Index for *(a)*

Figure 6.4: Context Index

Note that from the path signature alone one cannot see *where* exactly a label appears in the path. This is why the Context Index's filtering is inexact with respect to the query path.[19] The path signatures are used to isolate those occurrences which lack some of the labels in the query path, and therefore cannot possibly qualify as final hits. They are part of the recall noise (see

[18] Compare this to the keyword signatures discussed in section 3.3, which are arbitrary, non-compositional bit strings.

[19] Yet the filter is exact with respect to the given path signature, as explained in section 2.1.

definition 3 on page 16). However, among the occurrences which pass the filter because their linear context contains all the desired labels, some may still fail at the exact structure matching during verification, e.g. because the order of the labels in the path is scrambled. This part of the noise is not covered by the filter. In other words, the Context Index reduces the amount of recall noise slipping back to the query engine, but cannot guarantee to suppress it entirely.

Look-up procedure. During filtering, all occurrences returned by the index are compared to the query's path signature using bit string operations. Only those passing the test are handed over to the query engine, the others are discarded. From a logical point of view, a bitwise implication $\sigma_q \rightarrow \sigma_d$ is performed, where σ_q is the path signature of the query and σ_d the one of the keyword or label occurrence in question. Consider the trivial case where only one bit in the query signature is set. Assuming that the label book corresponds to the single bit set to 1, this may have either of the following two meanings, depending on the semantics of the query language[20]:

1. *Either* only occurrences reachable by the label path /book are acceptable. In other words, the query expression implies direct containment (see definition 16 on page 25).

2. *Or* only occurrences reachable by label paths beginning with /book are acceptable. This means the query expression implies indirect containment (see definition 17 on page 25).

In the first case, it would certainly be most effective to compare the signatures using strict equality: $\sigma_q = \sigma_d$ or with bit string operations, INV (σ_q XOR σ_d). This would rule out all false hits from the look-up results. On the other hand, exact matching is too restrictive for the second case, where paths with additional labels besides book are also tolerated. Only those paths lacking book altogether must be discarded. In terms of signatures, this means that while bits set to 1 in σ_q require the corresponding bit in σ_d to be set, too, unset bits in σ_q do not create any constraints. In other words, 0 is a "don't care" symbol in the query's path signature. Hence the corresponding comparison is $\neg\sigma_q \vee \sigma_d$, or $\sigma_q \rightarrow \sigma_d$. It can be implemented as (INV σ_q) OR σ_d, or else INV (σ_q ANDNOT σ_d).

Of course the same is true for more complex query signatures. We will meet the same comparison operation again in section 6.4.2.3 where the Signature File Hierarchy's look-up procedure is examined.

Selectivity and coverage. There are several optimization techniques for the Context Index (see the next paragraph) which are all based on the notion of *label selectivity*. Informally, a label is considered highly selective when it covers only few keyword occurrences in the database. When such a label is part of the query path, many occurrences can be dropped as recall noise by the Context Index because their linear context lacks the label. Thus highly selective labels make the filtering process more efficient. However, computing label selectivity on a per-keyword (or per-label) basis is quite an effort as it involves counting the pertinent and non-pertinent occurrences of every keyword (or label). Even though selectivity is calculated only once for every label when creating the index, for large databases with o occurrences in l labels, the $O(o \cdot l)$ procedure will quickly become infeasible.

As a remedy, [MS99a] suggests estimating selectivity based on a random sample of occurrences only. This is a good approximation if a normal distribution of occurrences over all labels is assumed. Alternatively, a label's selectivity may be computed based on its *coverage*, which is just the fraction of all occurrences appearing under any node with this label. Intuitively, one can regard the coverage of a label l as summing up the sizes of all regions governed by a node labelled l. Since occurrences are counted collectively rather than per keyword (or label) here, complexity is reduced to $O(o)$.

[SM99] proves that the selectivity η of a label is just the complement of its coverage γ, i.e. $\eta = 1 - \gamma$, provided that every keyword (or label) is equally distributed over all labels. Since this

[20] Or, to be more exact, on the semantics of the formalism used to express index look-up queries. Section 2.2 explains the difference between the two.

is normally not the case, coverage can only approximate selectivity. Hence faster index creation comes at the cost of suboptimal filtering. Especially for dynamic databases, however, it seems that this can hardly be avoided.

Storage reduction. [SM99] discusses three techniques for saving storage space with the Context Index, distinguishing bit string compression from label elimination methods. Both contribute to storage reduction by decreasing the size of path signatures. Furthermore, all techniques rely on the notion of label selectivity (see the previous paragraph). When each label's selectivity has been determined, any combination of the following methods may be chosen to create concise path signatures.

Bit string compression. For various reasons, only a fraction of all possible signatures actually occur in an ordinary database. Elements may only appear in certain places of a document, some of them being restricted by the underlying schema. Moreover, a number of signatures do occur in the database, but only rarely, mostly because they contain labels which either cover only a small region in the document or are scarcely used at all.

These observations suggest that there are efficient compression techniques, favouring the more frequent signatures at the expense of the rarer ones. [SM99] explores one way to encode such clusters in the signature space efficiently without losing information. It is based on the fact that the order of labels in a signature does not matter as long as all signatures use the same. If the label positions are arranged in order of decreasing selectivity, then signatures tend to start with a more or less long prefix of zeros. For signatures of length l, $\lceil \log_2 l \rceil$ bits suffice to hold the number of zeros in the prefix, thus encoding a significant part of each signature. The remaining bit strings vary in length.

This compression technique, also called *front compression*, can be applied analogously to the suffix of path signatures, whose rightmost bits tend to be set to 1. Contrary to front-compression, this method relies on the least selective labels to identify clusters in the signature space. Alternatively they may be excluded from the filtering process altogether, as described in the following paragraphs.

Label elimination. As explained above, selective labels in the database are most valuable for efficient filtering. As a consequence, the least selective labels can be disregarded at low risk since they do not contribute much to optimizing the index performance. There are two ways to carry out this label elimination: one may either consider each label in isolation or find pairs of labels where one can be subsumed under the other.

In the first case, a threshold t determines which labels are dropped in filtering. For instance, if $t = 85\%$, then any label containing more than 15% of the keyword (or label) occurrences in the database is not associated with a signature bit.

The second method for label elimination is actually a generalization of the first one, as shown in [SM99]. The idea is to identify labels which always appear as descendants of some other label and cover most of its occurrences. For example, consider a document schema in which chapter elements always occur inside a book element. The book element may also have a single title child but no other immediate descendants. (The database shown in figure 6.4 *(a)* conforms to this schema.) Obviously there will be hardly any keyword or label occurrences in book nodes which are not part of a chapter at the same time. Filtering out occurrences in books instead of chapters therefore adds only little recall noise. In other words, when the query restricts the search for some keyword or label to chapter nodes, the Context Index may instead check whether an occurrence has a book in its linear context, even though this will allow a few false hits in title nodes to escape filtering.

As can be seen from this example, labels which are dependent on others as just described may safely be neglected in filtering, thus saving bits in the path signatures. Yet incoming queries need to be adjusted to this elimination. Whenever the label to be removed appears in a query path, it is replaced by its subsuming ancestor before the query signature is created. Notice also that for this optimization to be applied, some knowledge about the database schema is required.

6.3.2.4 Features

Data representation. Similar to the CIS index (section 6.3.1), the Context Index is restricted to tree databases, and consists of a single flat look-up table. Path signatures, which serve for structural filtering, are represented as bit strings of fixed length in case of the plain Context Index, or of variable length when storage reduction techniques are applied (see section 6.3.2.3).

Navigation. Like all path look-up indices, the Context Index cannot be navigated, lacking a compositional path concept.

Path templates. As a feature of navigational indices, path templates are incompatible with the Context Index.

Node identification. The Context Index can be used with any node identification scheme.

Index update. While the Context Index without storage reduction supports incremental index updates, the performance of the space-optimized Context Index is affected by database modifications. Being based on statistical properties of individual labels in the database, its behaviour is sensitive only to major changes in the document collection, and can be said to degrade gracefully. But once selectivity has changed for a significant number of labels, only reorganizing the filter can restore its efficiency. This involves changing the query preprocessing as well as all signatures, and thus entails a complete reindexing.

Storage. Path signatures, being represented as bit strings, are amenable to various optimization techniques as discussed above, so the Context Index's space requirements should be comparably modest. However, each keyword (or label) occurrence in the database has got its own signature, which for rich databases must not be too short if a certain retrieval precision is desired (see section 3.3). But even if the index structure grows too big to be held in main memory, the table-based Context Index can take advantage of the sophisticated algorithms for secondary storage included in modern relational database systems.

6.3.2.5 Remarks

In [SM99] as well as [MS99a], which is an abridged version, the Context Index is presented as a filter add-on for path look-up indices, called *Context Filter*. The most prominent applications for this filter being content and label indices, we treat it as an index structure for either keywords or labels here. The name "Context Index" has been suggested by the authors of the above-mentioned papers.

While path signatures for the occurrences are created when the index is being built, i.e. before processing queries, the query path can only be encoded at query time. It might be adapted to suit regular and other path expressions (see section 4.1), although this is not covered in the literature. For instance, if a disjunction (`section|paragraph`) appears in the query path, both section and paragraph should be marked with a 1 in the signature. For numerical nesting constraints, such as `section`^{3}, path signatures could be expanded to hold more than one bit per label. Given the database in figure 6.4 *(a)*, the signature $\boxed{\texttt{1013}}$ would then select all paths containing a book label, a chapter label, and three section labels. Special symbols could indicate the minimum or maximum number of times a given label must appear in the linear context. This would allow to handle the $^+$ and * constructs.

The Context Index has not yet been evaluated experimentally. As far as filter compression is concerned, [SM99] reports that the total size of all signatures for a given keyword could be reduced by 70-80% in a case study, although these results have not yet been verified for larger databases with varying document types.

6.3.3 BitCube

6.3.3.1 Abstract

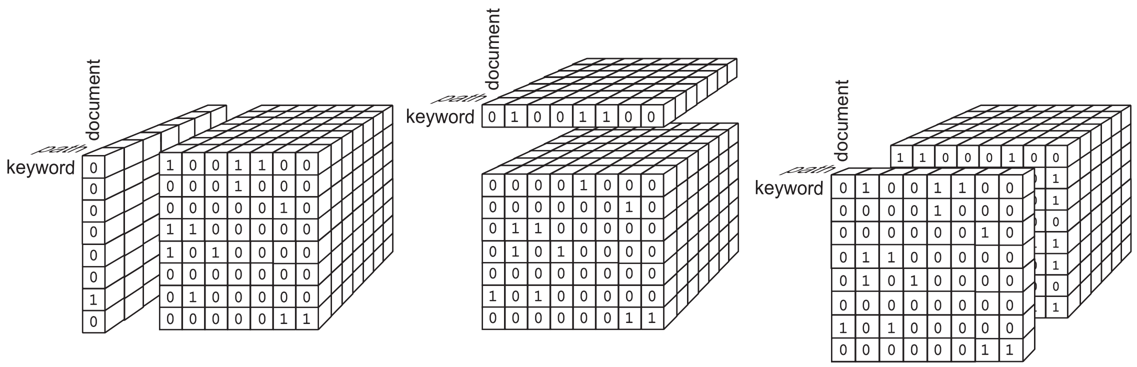
[YRCK01] and [YRC01] present the *BitCube*, a three-dimensional bitmap index. It adds a third dimension to the document/term matrix (see section 3.4) for the simple paths in every document. Clustering techniques are proposed to avoid sparse coding. The BitCube is confined to retrieval at document level.

6.3.3.2 Index interface

BitCube	DocIDs	$\longrightarrow$	$\mathcal{P}(\mathit{SimplePaths} \times \mathit{Keywords})$
	$\mathit{Keywords}$	$\longrightarrow$	$\mathcal{P}(\mathit{DocIDs} \times \mathit{SimplePaths})$
	$\mathit{SimplePaths}$	$\longrightarrow$	$\mathcal{P}(\mathit{DocIDs} \times \mathit{Keywords})$
	$\mathit{DocIDs} \times \mathit{Keywords}$	$\longrightarrow$	$\mathcal{P}(\mathit{SimplePaths})$
	$\mathit{DocIDs} \times \mathit{SimplePaths}$	$\longrightarrow$	$\mathcal{P}(\mathit{Keywords})$
	$\mathit{SimplePaths} \times \mathit{Keywords}$	$\longrightarrow$	$\mathcal{P}(\mathit{DocIDs})$

Table 6.5: Look-up patterns for the BitCube

Being accessible by any combination of its three dimensions, the BitCube supports six different look-up patterns. The first three can be viewed as extracting a single slice from the cube. In other words, each of these *primitive operations* corresponds to a different projection from a three-dimensional to a two-dimensional matrix, as illustrated in figure 6.5 below. The resulting data structure is a two-dimensional BitCube.



(a) path/keyword retrieval *(b)* document/path retrieval *(c)* document/keyword retrieval

Figure 6.5: Primitive operations on the BitCube

The first look-up pattern retrieves all keyword/path combinations from a given document *(a)*. The resulting matrix is in fact a path look-up table for a single document. As stated in [YRCK01], this operation may be helpful when browsing individual web pages in the database. Note, however, that the BitCube ignores the notion of a document node, subsuming all occurrences in nodes with the same label path under one index entry per document. Hence the resulting paths cannot be used to determine relationships between individual nodes.

Second, one can retrieve all simple paths from all documents where a given keyword occurs. This resembles the look-up patterns of the Element Locator Scheme and CIS index (see section 6.3.1), except that these two return not only simple paths, which may be ambiguous, but also a path annotation for identifying individual nodes in the database. Again, the BitCube does not distinguish individual instances of the same label path (see section 6.3.1.3 for an example of a query requiring a retrieval granularity at node level). Since the BitCube inherently supports multiple document trees, the index access results not in a one-dimensional path list, like with the other two indices, but in a two-dimensional matrix mapping document IDs to simple paths and vice versa, as shown in *(b)* above.

The third look-up pattern takes a simple path and yields all keywords reachable by this path, along with the documents hosting these occurrences. Again a two-dimensional matrix is returned, with document IDs in the first and keywords in the second dimension *(c)*. Note that this is like a document/term matrix from classic Information Retrieval (see section 3.4), narrowed down to those occurrences with a specific label path.

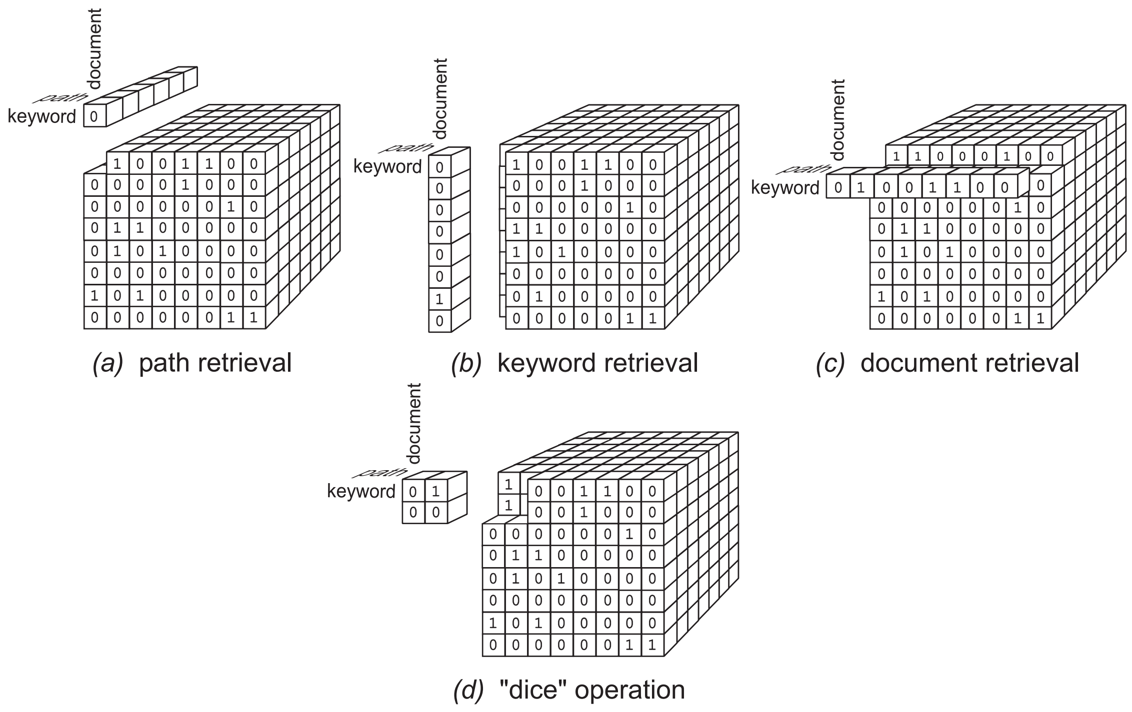


Figure 6.6: Complex operations on the BitCube

In addition to these primitive "slice" patterns, various *complex operations* are supported, such as searching all occurrences of a given keyword reachable by a given label path. Note that with the BitCube, paths are compared inside the index engine, whereas the other path look-up indices rely on the query engine for structure matching. The complex operations are represented by the last three look-up patterns. Figures 6.6 *(a)-(c)* above illustrate their effect on the BitCube. Moreover, [YRCK01] introduces a "dice" operation taking a set of triples (*docID*, *simplePath*, *keyword*) where each component comes from a given subset of documents, paths, or keywords, respectively. This is equivalent to cutting out smaller dice from the BitCube. Figure 6.6 *(d)* shows the special case where the three subsets supplied in the query are represented as contiguous ranges in the index structure, producing a single die. The "dice" operation can prove useful for creating document views, e.g. when only certain levels in documents may be accessed.

6.3.3.3 Details

Clustering. A major contribution of [YRC01] are the proposed clustering techniques for the BitCube, which is often very large though sparsely populated. Dividing a single index structure into multiple smaller ones obviously complicates the query plan because now the query engine needs to find out which of these indices is relevant for a given query. For perfect recall it may even be necessary to run look-ups against more than one of them. Although a secondary index structure can be charged with these segmentation issues, clustering may be costly.

On the other hand, maintaining multiple index structures is worth while if the sheer size of a single index slows down look-ups regardless of the complexity of their results. For a collection of non-trivial documents, the BitCube is likely to do so. Consider e.g. a small database of $d = 1000$ documents, containing occurrences of $k = 4000$ different keywords in $p = 200$ simple paths. A BitCube for this sample database has a total of $d \cdot k \cdot p = 800 \cdot 10^6$ cells of one bit each, consuming 100 Megabytes of memory. Clearly such an index structure cannot be held in main memory for significantly larger databases.

Rather than relying on the operating system's paging mechanism to partition the data for storage on secondary devices, the BitCube takes advantage of the fact that vast regions of this giant cube are populated by zeros. As an extreme case, let us assume that a single paragraph in one of the d documents contains a specific keyword not occurring in any other document. Then there is only one bit set in the whole slice of $d \cdot p = 2 \cdot 10^5$ bits representing this keyword in the BitCube. In other words, nearly 25 Kilobytes of memory are filled with zeros for a single keyword occurrence. All of the following clustering techniques strive to build separate indices for groups of documents such that together they consume less space than one huge BitCube for all documents. A set of documents can be partitioned according to either the paths or the keywords they contain. The resulting subsets induce multiple disjoint BitCubes, which are generally more dense because slices containing only zeros can be removed.

Similarity. [YRC01] introduces the notion of *similarity* as a means to partition the set of documents in the collection. Two documents are considered structurally similar if only few simple paths appear in one of them, but not the other. Analogously, content-based similarity declines with the number of keywords occurring in only one of a pair of documents. Either of the two measures can contribute to space reduction: When an index only covers documents where a given keyword does not appear at all (and which are in this sense similar), it need not reserve a slice for this word. Analogously, there is no point in storing nothing but unset bits for a simple path which occurs in none of the documents in a cluster. Potentially a lot of space can be saved when groups of documents lacking a specific keyword or path are kept together in the same data structure. From the opposite point of view, grouping documents which are similar because they have many keywords or paths in common is recommended for two reasons. First, it helps to reduce storage since collecting documents featuring a given keyword or path is dual to collecting those where the keyword or path is missing. Second, when the documents with a given keyword or path are kept together, query evaluation may be more efficient due to locality (see section 5.7).

Obviously not all keywords or paths from the entire database can be taken into account when grouping similar documents. There would be just too many similarity checks and, consequently, far too many separate indices. Therefore only some keywords and paths are considered in clustering, namely those with the greatest *popularity*, i.e. number of occurrences throughout the database. Furthermore, given two documents d_1 and d_2 and the set of most popular keywords or label paths, there are numerous degrees of similarity in between the one extreme, where d_1 and d_2 do not have any popular keyword or path in common, and the other, where d_1 and d_2 share all popular keywords or paths. Thus similarity is defined to take any value in the range $[0;1]$, as suggested in [YRCK01].

Clustering criteria: structure vs. content. The two criteria for similarity of documents we have isolated in the previous paragraphs are *structure* on the one hand and *content* on the other. Accordingly, there are two algorithms for clustering a set of documents. The structural one disregards the documents' textual content altogether, clustering them according to which popular label paths they contain. Since this takes place before the entire index is assembled (again, the content has been ignored so far), [YRC01] calls this way of partitioning a BitCube *early clustering*.

The second, keyword-based algorithm waits for the index to be built first, albeit in a two-dimensional representation where document/path pairs are mapped to keywords. Hence [YRC01] refers to it as *delayed clustering*. For the set of the most popular keywords the similarity between any two document/path pairs is then computed, inducing content-based document clusters. For instance, assume that document d_1 contains the popular paths /book/title and /book/chapter, among others, whereas d_2 contains only /book/title, but not /book/chapter, and d_3 neither of them. Then d_1 and d_2 are likely to end up in the same cluster, being relatively similar, whereas d_3 does not resemble any of them and therefore belongs to another cluster. Note that for the content-based clustering algorithm, $d \cdot p$ document/path pairs must be tested against k popular keywords, compared to only d documents and p popular label paths for the structural clustering algorithm.

Clustering direction: top-down vs. bottom-up. Another decision to be made arises when considering the way clusters are actually compiled from similar documents. No matter if structural or content-based similarity is preferred, the partitioning can be performed *top-down* or *bottom-up*. Top-down means the whole database is regarded as one huge cluster in the beginning. It is recursively divided into smaller clusters, namely one containing all documents exceeding an upper similarity threshold t_1, a second for those documents failing to reach a lower similarity threshold t_2, and a third comprising those documents with a degree of similarity between t_1 and t_2.

In bottom-up clustering each document is treated as a cluster in its own right first. Successively the existing clusters are merged, based on their similarity (in terms of structure or content), until a reasonable degree of fragmentation is reached. Similarity between clusters is computed by assessing the distance of their respective centres, as defined in [YRC01]. Those clusters whose distance falls below a threshold t together form a new one.

Performance. [YRCK01] compares the execution time for the three basic BitCube operations to those of other index structures. In all but one case the alternative approaches seem to be less efficient, although the high variance in their results might hint at inexact measuring. [YRC01] features performance tests for the BitCube with and without clustering, suggesting that much larger document collections can be handled when clustering is used. Sample storage data are not reported.

6.3.3.4 Features

Data representation. As a path look-up index, the BitCube has a (three-dimensional) table as main data structure. Additionally a secondary index may be used when the BitCube is clustered. Like the other path look-up indices, the BitCube is best used with tree databases. Individual document nodes are not distinguished in the index. As a consequence, multiple look-up results cannot be joined and the retrieval granularity is fixed to the document level, which is its main weak-spot. Although the index could certainly be extended to accommodate node identifiers, this would seriously deteriorate its storage behaviour. Unlike most other approaches reviewed in this work, the BitCube has been designed to accommodate multiple separate document trees without merging them to form a single database tree. As stated in section 5.1.2, however, this is of no practical importance in most applications.

Navigation. Lacking a compositional path concept, path look-up indices do not use internal navigation.

Path templates. As discussed in section 5.3, path templates only make sense with a compositional structure representation, not with the atomic path concept of the BitCube and other path look-up indices.

Node identification. Individual document nodes are not distinguished by the BitCube, so the question of node identification does not arise.

Index update. A single BitCube indexing the whole database can be updated incrementally. Adding a new label path, keyword, or document creates a new slice in the index, which is simply removed when the corresponding path, keyword or document disappears. Nevertheless, there may be obstacles to an incremental update when the BitCube is clustered. As shown in section 6.3.3.3, the clustering algorithms are based on the documents' similarity, which in turn depends on how often a set of keywords or simple paths occur in them. Regardless of whether a new document is added or an existing one is modified, changing the database may affect the popularity of both keywords and paths. As a consequence, not only does the partitioning of the document collection fail to reflect locality, as desired when clustering. Modifications of the database may also cause individual clusters to grow needlessly in size, adding very sparse slices to a BitCube when a keyword or path is added which did not exists in the cluster before. To take full advantage of clustering and make the index structures more compact again, one needs to repartition the updated index structure. Paths and keywords which have become more popular meanwhile may then be considered in clustering, grouping documents so as to avoid sparse BitCubes.

Storage. Section 6.3.3.3 demonstrates that even for small databases in-memory indexing with a single BitCube soon becomes infeasible. Using the clustering techniques proposed in [YRC01], it can be replaced by multiple smaller BitCubes for locality-based document grouping on secondary storage devices. However, a meta-index is needed to find out efficiently which BitCube to access at query time. Besides, since only one criterion (structure or content) is leveraged in clustering, querying the other may require look-ups against multiple BitCubes. How useful a given partition is depends largely on the chosen parameters controlling the clustering algorithms, of course.

6.3.3.5 Remarks

The BitCube as proposed in [YRCK01] and [YRC01] is an exact index. It can easily be adapted to achieve a higher compression rate with less precision, however, by dropping keywords or paths which hardly ever occur. As another optimization, the popularity measure used for clustering could be based on the document frequency instead of the collection frequency (see definitions 6 and 7 on page 17).

Being amenable to clustering, the BitCube is apt for huge static databases where secondary storage strategies are indispensable. Dynamic databases with unstable keyword and path frequencies would cause the index to be repartitioned too often. BitCube's major drawback, however, is the coarse granularity of its data model which ignores individual document nodes, reporting keyword occurrences only at the document or label path level. When both segmentation and high resolution matter, navigational approaches with paging strategies may be preferrable (e.g. the IndexFabric, section 6.4.5).

The presentation of clustering techniques in [YRC01] and [YRCK01] is somewhat confusing. The equivalence we establish between early and structure-based clustering on the one hand, as opposed to delayed and content-based clustering on the other hand, is not taken from the literature, but seems to be implied. Similarly, we conceive the clustering direction (top-down vs. bottom-up) as orthogonal to the clustering criterion (structure vs. content, i.e. early vs. delayed), although this is not mentioned in the two papers.

6.4 Navigational Indices

6.4.1 BUS

6.4.1.1 Abstract

[SJJ98] describes the *BUS index* (for *B*ottom-*U*p *S*cheme), which supports combined structure and content queries with result ranking. Among all nodes matching the structural part of the query, those where the query keyword occurs most often are ranked highest. The BUS index supports most, though not all frequency-based weighting techniques known from Information Retrieval.

6.4.1.2 Index interface

BUS index	*SimplePaths* × *Keywords* $\longrightarrow$ $\mathcal{P}$(*DocIDs* × *NodeIDs* × *Weights*)

Table 6.6: Look-up patterns for the BUS index

A look-up in the BUS index takes a simple path and a keyword as input and returns a set of (*docID*, *nodeID*, *weight*) triples, where *docID* is a document ID, *nodeID* a node ID, and *weight* a ranking value. Such a hit indicates that the keyword has been found in *docID*'s node *nodeID*, which can be reached by the given simple path, and that this occurrence should be ranked with a value of *weight* among all occurrences retrieved.

6.4.1.3 Details

Data structures. The BUS index consists of three data structures, depicted in figure 6.7 below, namely the schema tree *(a)*, which serves as structure index, a content index *(b)*, and a table of *accumulators (c)* storing a given keyword's term frequencies in all document nodes. The schema tree is augmented by *element type numbers*, preceded in *(a)* by the # sign. Each node in the schema tree is assigned a unique identifier so that nodes at different positions in the schema can be distinguished, even if they have the same label. One can think of these element type numbers as identifiers for simple paths in the schema tree.

As most navigational approaches with secondary content indices, the BUS index locates keyword occurrences in the lowest containing nodes only. Indirect textual containment (see definition 17 on page 25) is not supported by the content index and therefore needs to be deduced from direct containment (see section 16 on page 25). This type of inverted index is known from [LYYB96] as *leaf nodes only*, or *LNON*. (See section 4.4 for a brief review of inverted indexing schemes.) [SJJ98] explicitly precludes documents with mixed content, although this seems unnecessary (see subsection 6.4.1.5 for a discussion of this topic).

A posting in the content index consists not only of the document ID and node ID of the keyword occurrence in question, but also the corresponding level in the document tree and the element type number of the containing node. This additional information is needed both for joining structural and textual matches, and for computing weights (see section 3.2). Since weighting is based on how often a keyword occurs in a node, each posting also contains the term frequency of the indexed keyword in the corresponding document node. For instance, an index entry "XML" $\mapsto$ (&25, &802, 3, #7, 4) indicates that node &802 in document &25, which is located at level 3 and of element type #7, contains four occurrences of the keyword "XML".

According to [SJJ98], the accumulator table contains a counter for each document node, which can be uniquely identified by its document and node IDs. During a look-up these counters are incremented whenever the content index retrieves a keyword occurrence, as explained in the next paragraph. When all occurrences have been processed, the accumulator table serves two purposes. First, ranking weights can be calculated from the accumulated term frequencies for those nodes reachable by the query path. Second, from the number of counters with a value greater than zero, one can deduce the keyword's document frequency, which is needed by some weighting algorithms.

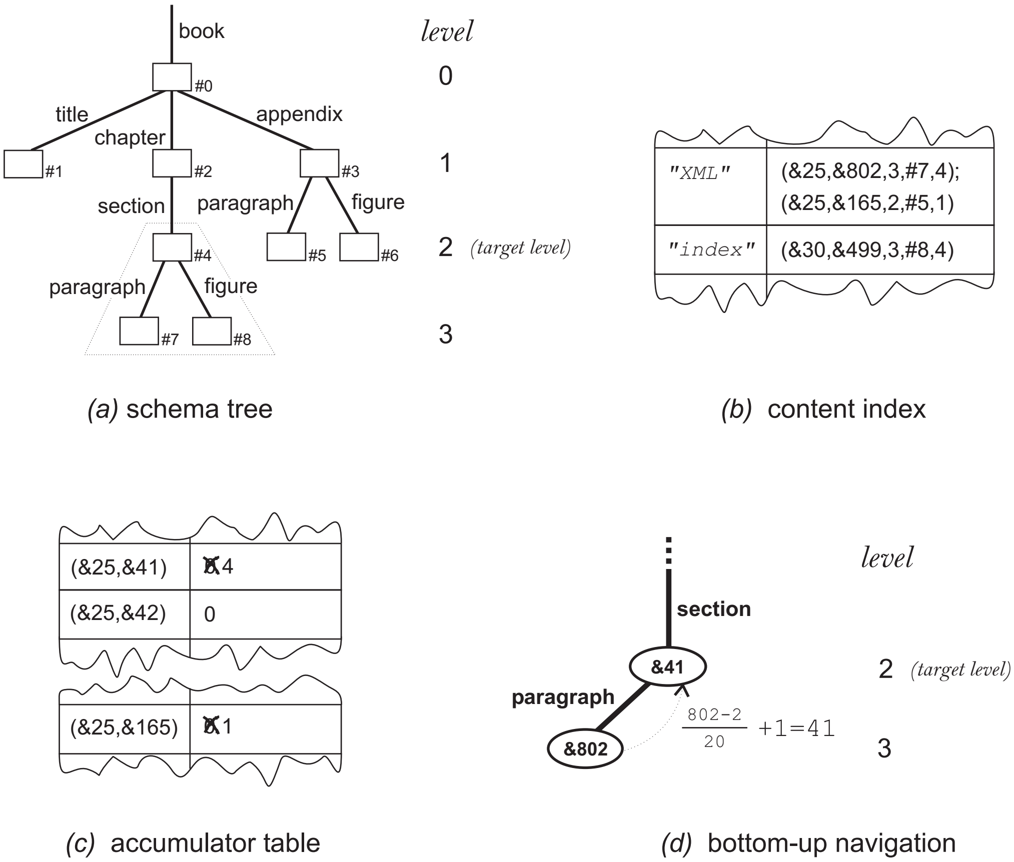


Figure 6.7: BUS index

Look-up procedure. The structural part of a look-up query, i.e. the simple path indicating where keyword occurrences are expected in the document tree, is matched against the schema tree in top-down direction. Unless a path is given that does not exist in the database, in which case the empty set is returned, the structure matching uniquely identifies a node in the schema tree. For instance, consider the query `/book/chapter/section["XML"]`. Evaluated against the database schema from figure 6.7 *(a)*, it selects the schema node with the element type number #4. Since only nodes from the subtree rooted at this node are relevant for content matching, their element type numbers are used to filter out keyword occurrences reachable by other label paths. Assume we store them in a set *Types* which now holds the two element type numbers {#7, #8}. The subtree root's level, called *target level* in [Shi01] (or *user level* in [SJJ98]), is stored for later use. It can also be deduced from the length of the query path. In our example, let $l_t = 2$ be the target level.

Once the relevant element type numbers have been collected, content matching takes place. As illustrated in figure 6.7 *(b)* above, a look-up in the content index yields a set of quintuples (*docID*, *nodeID*, *level*, *type*, *freq*), where *docID* is the ID of the document hosting a given keyword occurrence, *nodeID* is the ID of the document node where the keyword occurs, *level* is its level in the document tree, *type* its element type number, and *freq* indicates how many times the keyword was found in the node. In the following, only postings whose element type numbers have been selected during structure matching are considered. The others are discarded because they do not match the query path. If the keyword's document frequency is required for computing occurrence weights, however, the term frequencies associated with these mismatches must be added to the accumulator table first. The right accumulator can be identified using the pair (*docID*, *nodeID*). For instance, looking up the keyword "XML" in the content index yields two postings, both for occurrences in document &25: one has the element type number #7 $\in$ *Types*, which is a hit, and another has #5 $\notin$ *Types*, which is discarded after its accumulator (&25, &165) has been incremented by 1.

For each of the remaining occurrences, its ancestor at the target level needs to be retrieved, which is the one to be returned in the end. Let $\delta_l = level - l_t$. Then $\delta_l \geq 0$ is guaranteed[21] because when no mixed content is admitted, only leaf nodes contain keywords. The difference between *level* and l_t may be zero, however, in which case the query path reaches down to the leaves of the schema tree. Then $Types = \{type\}$, and all remaining postings are returned as look-up hits, their weights being computed from *freq*. Otherwise the node ID of the ancestor at the target level can be computed from *nodeID*. Recall from section 5.5.3 that when the Virtual Nodes encoding is employed, a uniform arity a is assumed for all nodes in the schema tree, and for a given node with ID *childID*, its parent's ID is $parentID = \left\lfloor \frac{(childID-2)}{a} + 1 \right\rfloor$. Iterating this computation δ_l times gives the ancestor's node ID. Its document ID is *docID*, too, of course. Consequently, the accumulator identified by (*docID*, *parentID*) is incremented by *freq*. In our example, the only hit from content matching is the occurrence in the document node &802 at level 3. Since $\delta_l = 3 - 2 = 1$, a single computation (shown in figure 6.7 *(d)* above) suffices to determine its ancestor at the target level, which is the document node &41. Given that the keyword occurs four times in node &802, the accumulator (&25, &41) is incremented by 4, as shown in *(c)*.

After the last occurrence has been processed this way, the accumulator table contains the term frequencies of the query keyword for a certain number of document nodes: those corresponding to the selected schema tree node, which are all at the target level, have been updated in the bottom-up step just described. (The first updated accumulator in *(c)* is such a case.) Furthermore, leaf nodes with irrelevant element type numbers have had their accumulators incremented when their occurrences were discarded. (This is illustrated by the second updated accumulator in *(c)*.) The number of non-zero accumulators with distinct document IDs now reflects the keyword's document frequency. It may be used for weighting the queried nodes, together with the term frequency from their accumulators. Finally, their document and node IDs and weight are returned as the look-up result.

6.4.1.4 Features

Data representation. For structure matching, the schema tree is used. It is enriched with element type numbers for all nodes. The other two indices, for retrieving content and accumulators, are table-based and can be implemented as hash tables or B-trees, for instance. Due to the Virtual Nodes identification scheme, which is based on a breadth-first tree traversal, only tree-shaped documents are suitable for use with the BUS index. The look-up pattern as implied by [SJJ98] is designed to accommodate multiple document trees separately in the database.

Navigation. The BUS index follows a hybrid look-up strategy. First structure matching is performed top-down from the root to the target level. Then a content index locates keyword occurrences in leaf nodes of the database tree. References to the relevant ancestor nodes at the target level are computed from the leaf IDs returned by the content index. Logically, this computation works its way upwards through the database tree, although physically it need not access the disk. Occurrence weights are calculated during this second step in a bottom-up manner, too (hence the name *B*ottom *U*p *S*cheme).

[21] This is why content matching is always performed bottom-up, and the whole look-up in a hybrid fashion.

Note that since the schema tree does not contain references to document nodes, the structure matching procedure cannot be given a specific start node other than the database root, as is the case with the ToXin index, for instance (see section 6.4.3.3). In other words, navigation with the BUS index is restricted to absolute paths. Besides, queries without keywords are not supported because there is no way to get hold of node IDs apart from keyword look-ups. Compare this to the Signature File Hierarchy (section 6.4.2), DataGuide (section 6.4.3), or T-Index (section 6.4.4) which can all be used to select document nodes based on structural constraints only. Pure content queries, by contrast, are supported by the BUS index.

Path templates. No template-based path look-ups are supported by the original BUS index. However, since the structure index is used to collect the relevant element type numbers only, the task of precompiling specific path queries boils down to setting up another secondary index which maps query paths to element type numbers. This can be accomplished before building the context index, and may be useful for deeply nested documents where navigating the schema tree is tedious.

As an example, consider the template /book/chapter/section $x_1 \boxed{\mathsf{P}} x_2$. Queries instantiating this template, when applied to the database in figure 6.7 *(a)*, can only select the following schema paths: /book/chapter/section with $\boxed{\mathsf{P}} = \epsilon$, /book/chapter/section/paragraph with $\boxed{\mathsf{P}}$ = paragraph, and /book/chapter/section/figure with $\boxed{\mathsf{P}}$ = figure. Their relevant element type numbers for content matching are (&7, &8), (&7), and (&8), respectively. The template index has the entries /book/chapter/section $\mapsto$ (&7, &8), /book/chapter/section/paragraph $\mapsto$ (&7), and /book/chapter/section/figure $\mapsto$ (&8).

In terms of the query evaluation described above, a single look-up in the template index thus yields the set *Types* of element type numbers to be considered in content matching, replacing the whole top-down part of the evaluation. Recall that the target level needed for the bottom-up step can be computed from the query and does not require schema navigation. If the binding for $x_1, x_2, \ldots$ must be returned, too, there are in fact multiple target levels. The level differences necessary to compute the corresponding node IDs can be deduced from the query path.

While a table-based index stores path prefixed redundantly, as observed in section 6, it accelerates query evaluation. The template index is updated incrementally whenever the database schema changes. If a new path in the schema matches any of the templates, a new entry in the template index is created mapping the path to its relevant element type numbers.

Node identification. The BUS index relies on the Virtual Nodes encoding scheme presented in section 5.5.3. As a consequence, only tree-shaped documents can be indexed and incremental updating is limited, as explained in the next paragraph.

Index update. As all approaches based on Virtual Nodes, the BUS index is not recommended for frequently changing databases unless updates tend to preserve a maximum node arity. Section 5.5.3 discusses how the Virtual Nodes identification scheme affects incremental index updates. As long as no node in the database exceeds the assumed uniform arity a, new nodes can be assigned a node ID, causing further entries to be added to the content index and accumulator table. Of course the schema tree may need to be modified to accommodate new paths. When any document node has more than a children, however, all IDs in the database must be adjusted, which means that both the content index and the accumulator table have to be rebuilt from scratch.

Storage. Ignoring node IDs, the structure index does not grow with the number of nodes in the database. It only depends on the database schema. The content index behaves just like an ordinary text index except that its postings contain more than a node ID and are therefore larger, although fixed in size.

As far as the accumulator table is concerned, information given in the literature diverges. [Shi01] confines accumulation to the target level, whereas [SJJ98] declares that each document node is assigned an accumulator. The first solution implies that a new accumulator table is created for every query since the target level may change. However, not all document nodes at target level need to be weighted but only those with the right element type number. Besides, computing document frequencies for weighting requires accumulators at least for all leaf nodes in the database which are not covered by the target level accumulators. Alternatively, one may maintain a separate document counter.

On the other hand, if every document node has its own accumulator, the space required by the BUS index is considerable, leading to an indexing overhead of up to 150% in one of the experiments. Compressing the postings from the inverted index reduces the overhead to about 30% on average. Of course this affects the retrieval latency.

6.4.1.5 Remarks

We briefly mention some minor enhancements of the BUS index, most of which are not covered in the literature so far. First, there is a way to leverage specific classes of queries using path templates, as illustrated above. Second, documents with mixed content seem to be amenable to BUS indexing, too. When the inverted index retrieves a node with mixed content below the target level, the bottom-up step takes place just as before. Occurrences above the target level are discarded anyway because their element type number cannot possibly appear in the subtree spanned by the query path. Third, to optimize structure matching, the element type numbers could be assigned according to the interval encoding scheme (see section 5.5.2). This way no set *Types* needs to be maintained and searched to filter out keyword postings. Instead each posting's element type number is compared to the interval of the matching schema tree node to find out which postings lie outside the subtree it spans. This is done in constant time, no matter how large the subtree is.

[Shi01] describes how to store keyword occurrences from attributes in a database rather than an inverted index, leaving the look-up procedure almost unaffected. While it takes longer to build the index structure this way and the storage overhead increases, retrieval is faster. Moreover most database systems support sophisticated selection operators, turning the content index for attributes into a value index, as discussed in section 6.2.3.2.

While [SJJ98] features experimental results illustrating the space requirements and retrieval latency of the BUS index approach, no details are given about the quality of the ranking.

6.4.2 Signature File Hierarchy

6.4.2.1 Abstract

The *Signature File Hierarchy* ([CA98], [CA99]) combines signature files and Tries to index structured documents. Based on the schema tree, it has a built-in content filter allowing to discard during structure matching some nodes which do not contain the query keywords. A modified Trie is used for efficient signature matching.

6.4.2.2 Index interface

Signature File Hierarchy	$\mathit{SimplePaths} \times \mathit{Signatures}$	$\longrightarrow$	$\mathcal{P}(\mathit{NodeIDs})$
	$\mathcal{P}(\mathit{SimplePaths} \times \mathit{SignaturePaths})$	$\longrightarrow$	$\mathcal{P}(\mathit{NodeIDs})$

Table 6.7: Look-up patterns for the Signature File Hierarchy

According to the first look-up pattern a simple path and a keyword signature are mapped to a set of node IDs (alternatively, the whole ID path may be returned). The retrieved nodes are reachable via document paths matching the given simple path. Each node on such a document path has a signature which in turn matches the given keyword signature. Signature ambiguity taken into account, not all of the retrieved nodes really contain the keywords encoded in the signature. Textual verification in the query engine is needed to rule out these false hits. Note that with respect to the given signature, the index is exact, i.e. performs with optimal precision and recall, as discussed in section 2.1.

Tree queries, as opposed to path queries, could be processed using the first look-up pattern, too, namely by joining the results of multiple path look-ups. In this scenario the query engine is responsible for dividing the query tree into simple paths, obtaining a set of ID paths for each of them from the index, and comparing these sets to figure out which paths together form a subtree of the database tree. As shown in [CA98], however, tree queries can also be handled internally by the index. The algorithm, which uses the second look-up pattern, is discussed in the next subsection.

6.4.2.3 Details

Signature propagation. The Signature File Hierarchy is used not only as a structure index but also as a textual filter. As its name suggests, it relies on *signatures* which are used to represent individual keywords as well as the whole textual content of a node in the database (see section 3.3). Based on signature examination, entire subtrees can be safely ignored during structure matching because they lack the desired keywords, and therefore cannot possibly qualify as hits (see below).

Before the hierarchy can be built the keywords from the documents need to be identified using standard IR methods (see section 3.2). Every leaf of a document tree contains a (possibly empty) subset of representative words, each of which is assigned a signature, i.e. a bit string of fixed length. The set of keywords appearing in a given leaf node can now be encoded as the bitwise disjunction of their signatures. The result is a signature for this leaf node. Similarly, any inner document node is assigned a signature by recursively disjoining the signatures of its descendants, as depicted in figure 6.8 *(a)* (for the signatures of the keywords "index" and "XML", see figure 3.2 on page 17). Just like for leaves, the signature indicates which representative words can be found in this node. We call this way of creating new signatures from existing ones *signature propagation*.

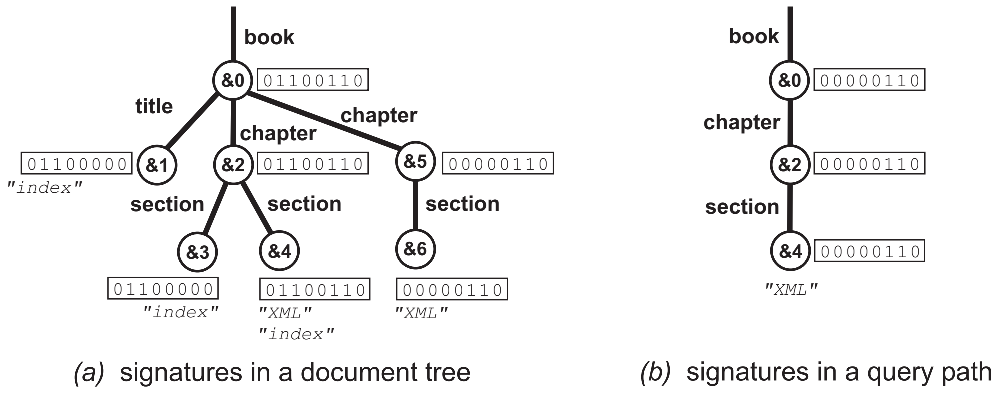


Figure 6.8: Signature propagation

Note that the signatures of higher-level nodes may well differ from those of their descendants. The reason is that nodes lower in the hierarchy cover a smaller subset of the keywords in the document, while their ancestors' signatures represent the content of a more extensive region. Due to the bitwise disjunction, no bit set in the signature of any document node can ever be unset on one of the higher levels. Hence signatures of nodes in the upper part of the hierarchy are generally less selective. As an extreme case, consider a node whose signature consists entirely of ones, and therefore lacks any information for content filtering. This does not necessarily mean that the node contains occurrences of all representative words, but rather that signature propagation saturated on the way up to this node. We observed the same phenomenon in section 3.3 where multiple combinations of keyword signatures all produced identical bit strings when disjoined. Because of this ambiguity signature-based filtering is inherently inexact and at the same time conservative. In case of the Signature File Hierarchy, this means that any branch pruned during filtering does indeed lack the keyword in question; however, there may be false hits among those nodes which pass the filter.

Data structures. Two particular index structures are part of this approach: the *Signature File Hierarchy* proper, and a *Signature File Index* as secondary index (see below). Other auxiliary data structures include a parent/child index, which may be incorporated in the main index structure (see the paragraph on query evaluation below), and a text index. Besides, the preprocessed query is represented in a dedicated data structure, too.

Query Signature Hierarchy. In much the same way as signatures are assigned to document nodes, they can be computed for the nodes in the query tree. The resulting data structure is called Query Signature Hierarchy in [CA98]. When the first look-up pattern is used, the hierarchy is a single path whose higher nodes all have the same signature as the only leaf (see figure 6.8 *(b)* above). In the more general case, when a non-trivial tree is given, the Query Signature Hierarchy looks just like a small document tree with signatures *(a)*.

Signature File Hierarchy. Consider the schema tree, in which all document nodes accessible via a given simple path are collectively represented by a single node, as explained in section 4.2. When reaching such a node during structural query evaluation, we would like to know immediately which of the corresponding document nodes contain the right keywords at all. There is no point in matching the rest of the query tree against those which do not, because even fulfilling the structural requirements they will fail the textual check anyway. The whole subtree rooted at such a document node can be pruned off safely. For this procedure to work, a node in the schema tree needs to store not only the IDs of the document nodes it represents, but also their signatures. Adding a list of signatures (a *signature file*) to every schema tree node turns the schema tree into a Signature File Hierarchy, as shown in figure 6.9 below. Together with the Query Signature Hierarchy this index structure helps to reduce the search space as described in the next paragraph.

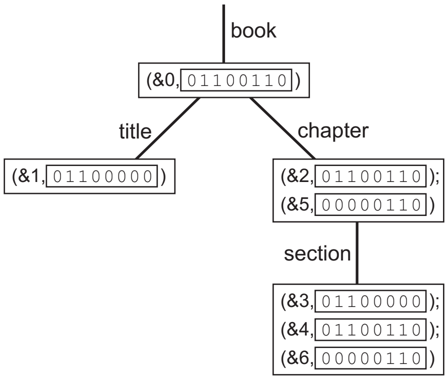


Figure 6.9: Signature File Hierarchy

Look-up procedure. Queries are evaluated in top-down direction. The Query Signature Hierarchy is matched against the Signature File Hierarchy in a depth-first traversal. When a node from the Signature File Hierarchy matches the label of the current query node, all entries in its signature file are compared to the query node's signature, except those excluded in previous steps. Document nodes whose signature fails in the comparison do not contain the desired keywords, so their descendants need not be considered in the matching process. (By the same argument, their parent and sibling nodes could be excluded from further matching, although [CA98] and [CA99] do not address this issue.) The children's and parents' IDs can be obtained using a parent/child index (see section 6.2.3.5). Alternatively, signature file entries may be grouped by parent ID which makes it easy to single out irrelevant signatures.

Analogously to the Context Index's matching procedure (see section 6.3.2.3), the signature comparison is equivalent to a bitwise logical implication $\sigma_q \to \sigma_d$, where σ_q is the signature of the current query node and σ_d the one of the document node being checked (i.e. a single entry in a schema node's signature file). This is because if a keyword appears in the query node (setting some of σ_q's bits to `1`), then it is required to appear also in the document (so the same bits in σ_d need to be `1`, too, otherwise the comparison fails). On the other hand, since keywords not present in the query do not affect the outcome of the comparison, there is no constraint regarding those bits in σ_d which are not set in σ_q. In other words, a `0` in σ_q acts as a "don't care" symbol. Hence the signature comparison can be implemented as the disjunction of σ_d and the bitwise inversion of σ_q: (`INV` σ_q) `OR` σ_d, or `INV` (σ_q `ANDNOT` σ_d).

For example, consider the Signature File Hierarchy shown in figure 6.9 above and the query `/book/chapter/section["XML"]` from figure 6.8 *(b)*. Their root labels book match, and comparing the query root's signature $\boxed{\texttt{00000110}}$ to the only entry in the schema root's signature file, $\boxed{\texttt{01100110}}$, as described above produces a match, too. For the second step in the query path, chapter, two signature file entries need to be checked: $\boxed{\texttt{01100110}}$ and $\boxed{\texttt{00000110}}$. Both pass the test, so the procedure continues with the section step. Among the three entries in the lowermost signature file, only the last two ($\boxed{\texttt{01100110}}$ for document node &4, and $\boxed{\texttt{00000110}}$ for document node &6) match the query node's signature. Hence there are two hits, &4 and &6. Compare this to figure 6.8 *(a)* to verify that these are the only two document nodes reachable via /book/chapter/section which contain the keyword "XML".

Enhancements. The signature comparison described in the previous paragraph has to be performed for all entries in the signature file of a schema tree node checked during structure matching. For a file containing n entries of b bits each, and without any optimization, this amounts to checking $n{\cdot}b$ bits. Unset bits being a "don't care", the task boils down to comparing only those positions where σ_q is set to `1`. Still the same position is checked n times, once for every signature file entry.

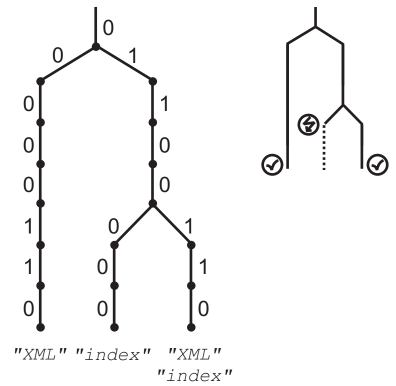


Figure 6.10: Signature File Index

A secondary *Signature File Index* is used to make the comparison procedure more efficient. [CA99] introduces a more compact, Trie-based representation of the signature file which requires much less than n comparisons per bit on average. To avoid scanning the whole signature file, all its entries are considered paths in a modified Trie as shown in figure 6.10, such that a traversal from the root to a leaf visits all bits of the corresponding signature from left to right. The search for matching signatures can now be carried out by following a couple of paths in the Trie, just like a text search. Unlike the original Trie, however, this data structure must support the bitwise logical implication $\sigma_q \to \sigma_d$ instead of equality $\sigma_q = \sigma_d$. As laid out above, the difference between the two is that in case of a `0` bit in σ_q, both a `0` and a `1` are accepted at that position in σ_d. Therefore a modified traversal algorithm for Tries is proposed which corresponds exactly to the "don't care" semantics of `0`. While a `1` at any given position in σ_q indicates that only the branch labelled "1" is to be followed in the Trie, a `0` requires both branches to be examined, as can be seen in the figure. To anticipate section 6.4.2.4, for a signature file containing n signatures and s bits set to `1` in σ_q, only $\mathcal{O}(n/2^s)$ comparisons are needed in the worst case.

6.4.2.4 Features

Data representation. The Signature File Hierarchy has been proposed for tree databases only, although the algorithms for creating signatures and for matching a query could be extended to general directed acyclic graphs. As described in section 4.2, there are two ways to build index structures based on the schema of a graph database. Accordingly, content filtering with signatures, as performed by the Signature File Hierarchy, can be applied to the two different navigational index structures dealing with graph databases, namely the DataGuide (see section 6.4.3) and the T-Index (see section 6.4.4).

The three major data structures discussed in this section are trees. Only the parent/child index, whether implemented as a separate data structure or fused with the signature files, is based on a table. The Signature File Hierarchy could easily be extended to accommodate multi-tree databases as described in section 5.1.2.

Navigation. It has been mentioned above that look-ups are performed in top-down direction. As a filter for pruning trees, the Signature File Hierarchy does not make sense for bottom-up searching. Hybrid strategies, e.g. with the bottom-up part replacing textual verification, are discussed neither in [CA98] nor in [CA99]. Note that unlike the BUS index (see section 6.4.1), one can query the Signature File Hierarchy for a label path without supplying any keywords to be searched. The reason is that its structure index contains references to document nodes, as mentioned above, whereas the BUS index is based on the schema tree without node IDs. If no keyword is given, navigation in the Signature File Hierarchy is reduced to ordinary tree matching without signature comparison. Just like the BUS index, however, the Signature File Hierarchy has not been designed for relative query paths, as e.g. the T-Index.

Path templates. There is no provision for template-based index look-ups.

Node identification. The Signature File Hierarchy does not rely on any special numbering scheme for document nodes. Supporting ancestor/descendant queries with structural node identification should be straightforward.

Index update. As described in [CA98], the Signature File Hierarchy can be updated incrementally. Since document node IDs are arbitrary, the index update after a structural or textual modification of the database is restricted to the document in question. When the content of a document node has been changed, its signature needs to be updated. This modification is propagated upwards to the root of the database tree. Note that for updating ancestors on the root path, the signatures of all their child nodes (whether modified or not) must be read, whereas their lower-level descendants (the "cousins" of the nodes already updated) need not be accessed. This ensures that the time required for refreshing the Signature File Hierarchy is proportional to the depth of the database tree. Of course, the secondary indices on all modified signature files have to be rebuilt, too.

Storage. Apart from the database contents, the size of the Signature File Hierarchy depends mainly on how long an individual signature is. The signature size in turn affects the retrieval precision, as explained in section 3.3. Thus it is difficult to assess the storage requirements of the Signature File Hierarchy in general. However, the fact that the ID and signature of each document node is stored in the hierarchy, and that for each signature file there is a secondary index structure suggests that in case of a huge document database, the whole index will not fit main memory. Storage on secondary devices is not addressed in [CA98] or [CA99].

6.4.2.5 Remarks

Section 2.2 states that the Signature File Hierarchy and the Context Index (section 6.3.2) can be regarded as dual in the following sense: while the former uses signatures for content filtering during structure matching, the latter, when built over keywords, integrates a signature-based structure filter into a content index. Their path-based look-up patterns reflect this duality, as can be seen in table 6.2.2 on page 41. However, the semantics of keyword signatures and path signatures differ, as mentioned in section 6.3.2.3. Moreover, since the Context Index retrieves a new set of path signatures for any keyword, no static secondary index like the Signature File Index can be used to enhance signature comparison.

While both look-up patterns of the Signature File Hierarchy can be extended to accommodate multiple keywords per label path and Boolean operations on them, regular expressions on the content would require additional constraints on the keyword signatures (see below). Regular path expressions are not supported either, including relative paths. [CA98] suggests the use of path variables for unknown path fragments of arbitrary or fixed length. As an alternative, it seems straightforward to apply textual filtering based on signature files to other index structures appropriate for such extended path concepts. For example, one could attach signature files to the nodes of a 2-index (see section 6.4.4) in much the same way as they are used with the Signature File Hierarchy. This would yield a structure index for relative path expressions with keyword-based pruning. Other features such as label wildcards could be reconciled with the original Signature File Hierarchy data structure using a slightly modified evaluation algorithm.

[CA98] features an update algorithm as well as a performance analysis and comparison for three retrieval methods (*Nested Loop Top-Down Retrieval*, *Yong's Method*, and *Yong's Method combined with Top-Down Hierarchy Retrieval*). The Signature File Index as a secondary index for signature matching is presented in [CA99].

6.4.3 DataGuide

6.4.3.1 Abstract

The *DataGuide* ([GW97b], [GW97a]) is an index structure for both tree- and graph-shaped documents. Based on the database schema, it serves query evaluation as well as schema browsing. Here we concentrate on the first aspect, which is tightly coupled with the more specific concept of a *strong DataGuide*. It supports deterministic navigation during structure matching, at the cost of exponential growth in the worst case. For trees, the strong DataGuide and the 1-Index (see section 6.4.4.3) are equivalent, coinciding with the schema tree defined in section 4.2.

6.4.3.2 Index interface

DataGuide	$\mathit{SimplePaths} \longrightarrow \mathcal{P}(\mathit{NodeIDs})$

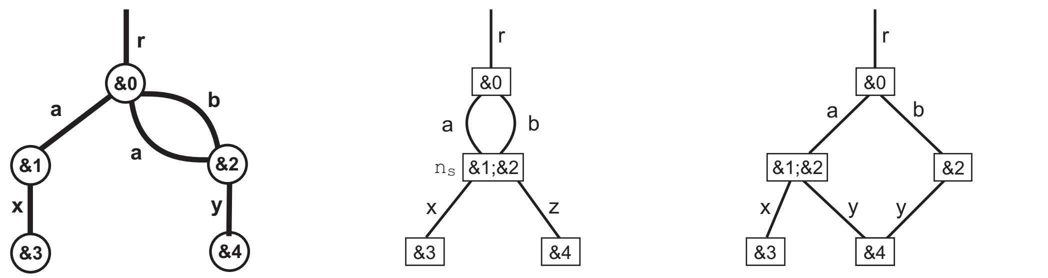
Table 6.8: Look-up patterns for the DataGuide

The DataGuide has a single look-up pattern, taking a simple path and returning a set of node IDs. No relative paths can be given. As with most navigational index structures, the DataGuide's look-up procedure can be modified to evaluate any regular path expression, but they are not inherently supported, as e.g. by the 2-Index (see section 6.4.4).

6.4.3.3 Details

Concept of a DataGuide. In the most general sense, a *DataGuide* for a given database is a graph such that every simple path from the database is represented exactly once, and no other paths appear in it. This resembles closely our definition of a schema tree in section 4.2. The only difference is that DataGuides are not required to be trees. However, when using them as structure indices with annotations as we have shown for the schema tree, cycles and node sharing may cause simple paths to be associated with references which are not in their target set. Hence not all DataGuides are suitable for indexing. We say that a DataGuide is *weak* when navigating certain of its paths leads to wrong references, in contrast to *strong DataGuides* which guarantee exact retrieval (see below).[22]

For instance, assume a graph database as the one shown in figure 6.11 *(a)*. Its two a edges below the root need to be merged in the DataGuide because each path must only be represented once. There are two possibilities to do this, depicted in *(b)* and *(c)*, respectively. In the first DataGuide, which is a weak one, the paths /r/a and /r/b share the same node n_s. Consider the document node &1 which can be accessed via /r/a but not /r/b. Obviously &1's ID must be attached to n_s, as there is no other place to put references to nodes reached by /r/a. But then &1 could be mistaken to be accessible via /r/b, too. In other words, the node annotation of such a DataGuide is ambiguous. In *(c)* this problem does not arise because a separate node is reserved for the /r/a path which can now be annotated with a different target set than /r/b.



(a) database graph *(b)* weak DataGuide for *(a)* *(c)* strong DataGuide for *(a)*

Figure 6.11: DataGuide vs. strong DataGuide

Strong DataGuide. To capture these subtleties of index annotation, [GW97a] introduces the notion of a *strong DataGuide*. Strong DataGuides guarantee that whenever one of their nodes n_s can be reached by multiple paths p_j in the index, like in figure 6.11 *(b)* above, these paths have identical target sets in the database, i.e. they reference the same set of document nodes. Hence all document nodes referenced in n_s can be reached by traversing any of the paths p_j, and no ambiguities arise as to which reference belongs to which path. Conversely, simple paths with the

[22] Both [GW97b] and [GW97a] use the term "strong" in the same sense as we do, whereas "weak" DataGuides are not attributed at all. However, since only strong DataGuides can be used for indexing, most authors refer to them simply as DataGuides, unless stated otherwise.

same target set in the database always lead to the same node in a strong DataGuide, such that no space is wasted in needless node duplication. The graph shown in figure 6.11 *(c)* is a strong DataGuide for the database in *(a)*.

More formally, let P denote the set of simple paths in a database DB, N the set of nodes in a strong DataGuide for DB, and $\equiv_t$ an equivalence relation on $P \times P$ such that $p_1 \equiv_t p_2$ *iff* p_1 has the same target set as p_2, for any two $p_1, p_2 \in P$. Then there exists a bijective mapping m from N to the set of equivalence classes $[p]_{\equiv_t}$. Each $n_i \in N$ is reachable via all $p_j \in m(n_i)$, and annotated with the target set of any such p_j.

Since the equivalence classes $[p]_{\equiv_t}$ form a partition of P, any simple path in the database is represented exactly once in the DataGuide, i.e. the index is deterministic. It can be shown that for each database graph there is a unique strong DataGuide. Moreover, recalling section 4.2 it is easy to verify that when the collection contains only tree-shaped documents, $\equiv_t$ and $\equiv_p$ coincide. In other words, the strong DataGuide of a tree database is just its the schema tree.

[GW97a] explains in detail how to derive a strong DataGuide from a graph database. [NUWC97] proves that this procedure is equivalent to converting a non-deterministic finite automaton (NFA) into a deterministic one with the powerset construct.[23] Thus the strong DataGuide can be regarded as the standard powerset automaton of the NFA corresponding to the database graph, as [MS99b] states.

Space requirements. As mentioned in section 4.2, the deterministic schema graph of a database with cycles and node sharing may be of exponential size. More precisely, there are cases where it contains exponentially many nodes, edges, and database references, compared to the number of nodes, edges, and database references in the database. Unlike the 1-Index (see section 6.4.4), which relaxes the unique path requirement in order to remain compact, the strong DataGuide incurs this storage explosion to keep structure matching deterministic. The analogy to the determinization of an NFA mentioned above explains why in the worst case, time and space required for creating and storing a strong DataGuide grow exponentially with the size of the database, while the non-deterministic 1-Index stays linear.

We give two examples for the exponential growth of a strong DataGuide. Far from being typical for realistic document databases, they both suggest that the cases in which the DataGuide's performance declines are not among the most frequent in practice. Often only few node IDs need to be duplicated for annotating different simple paths, which already meet at the next level to share the same subgraph, as shown for &2 in figure 6.11 *(c)*.

Figure 6.12 shows a cyclic database DB_1 with extensive node sharing *(a)* and the corresponding strong DataGuide *(b)*. The database graph is given in [Sch01] as an example of an NFA whose minimal powerset automaton has exponentially many states. As one can guess from the figure, if there are n nodes in a database like the one in *(a)*, the DataGuide contains $n_{DG} = 2^{n-1}$ nodes, $2^{n-1} + 1$ edges, and even more references to document nodes. We sketch a proof for the first conjecture to illustrate why there are so many paths annotated with different target sets in the DataGuide.

Proof. *($n_{DG} = 2^{n-1}$)* We successively construct the sets T_h of target sets annotating the DataGuide when only paths up to level $0 \leq h \leq n-1$ are considered in the database DB_1, and then show by induction over h that $n_{DG} = |T_{n-1}| = 2^{n-1}$. Let $L = \{a, b\}$ be the set of labels and I the set of node IDs in DB_1. To formalize the traversal of an edge in DB_1, we propose a function $f\colon L \times \mathcal{P}(I) \longrightarrow \mathcal{P}(I)$ which maps a label $l \in L$ and a set $\{i_0, \ldots, i_k\}, 0 \leq k < n$, of references in a DataGuide node to the set of references in its l child:

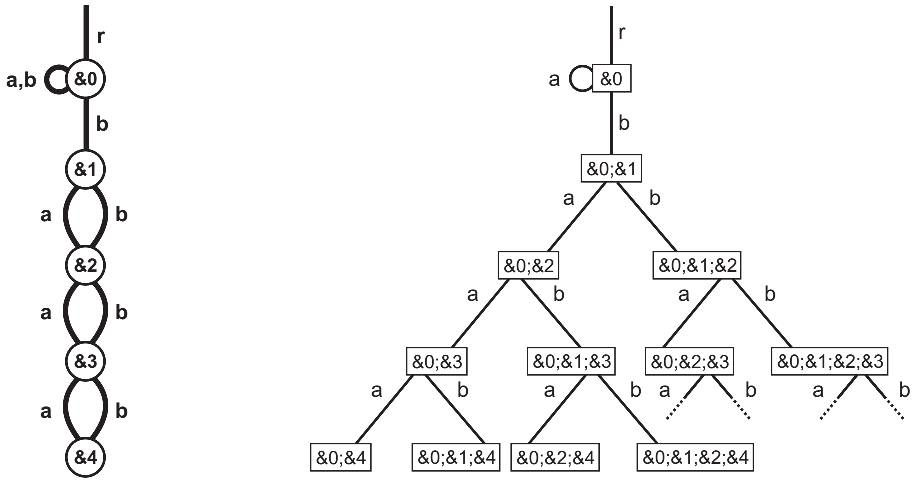
$$f(l, \{i_0, \ldots, i_k\}) = \begin{cases} \{i_0, i_1+1, \ldots, i_k+1\}, & l = \mathsf{a} \textit{ and } i_0 = 0; \\ \{i_0, i_0+1, i_1+1, \ldots, i_k+1\}, & l = \mathsf{b} \textit{ and } i_0 = 0; \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

[23] Actually the proof is given for *Full Representative Objects* (FRO), which subsume DataGuides.

The reader may verify that f is equivalent to a transition function of the database interpreted as an NFA. f is only defined on sets including &0. An a edge increments all IDs but &0. A b edge does the same, but additionally includes &1 in the set. Compare this to the database, where both a and b link to the next level, except at the root which can only be left via b. The two cycles on the root node cause f to reproduce &0 given any set where this ID already occurs. It is easy to see that f is injective.

Now let T_h be the set of all path annotations in the DataGuide up to, and including, level h. Each element in T_h is a set of node IDs from DB_1. Following our definition of the function f, which models edge traversals in DB_1, we state that $T_{h+1} = f(L \times T_h)$. Moreover, the DataGuide obviously has the same number of levels as DB_1, namely $n - 1$. Thus T_{n-1} is the set of all DataGuide nodes, and $n_{DG} = |T_{n-1}|$.

To prove that $|T_{n-1}| = 2^{n-1}$, first we verify that $|T_0| = |\{0\}| = 2^0$. For any T_h, which by induction contains 2^h elements, we obtain $|T_{h+1}| = |f(\{a, b\} \times T_h)| = 2 \cdot 2^h = 2^{h+1}$, where the next to last equation holds because f is injective. □



(a) database graph *DB₁*   *(b)* strong DataGuide for *(a)*

Figure 6.12: Exponential growth of the strong DataGuide for a cyclic database

There are also acyclic databases whose strong DataGuide grows exponentially. We briefly illustrate such a case in figure 6.13. Let DB_2 be a triangular node matrix with h rows and columns, as shown in the figure. Adjacent nodes in each column are linked by both a and b edges. Additionally, all nodes $n_{i,j}, 0 \le i, j < h$, on level i project to $n_{i+1,i+1}$ via b only. Obviously DB_2 contains $\frac{h(h+1)}{2}$ nodes and $h(h-1)$ edges, i.e. the number of both nodes and edges is in $\mathcal{O}(\, h^2\,)$. We show that a strong DataGuide for DB_2 has at least $n_{DG} = 2^{h-1}$ leaf nodes, one for each label path from the root of the database graph to one of its leaves.

Proof. *($n_{DG} = 2^{h-1}$)* Every node $n_{i,j}, 0 \le i, j < h$, in DB_2 has outgoing edges for both a and b. Hence there are $|\{\mathsf{a}, \mathsf{b}\}|^{h-1} = 2^{h-1}$ root-to-leaf paths in DB_2. To verify that they all have different target sets, consider e.g. the path /a/a/b/a/a reaching the leaves in columns 0 and 3. Without the b step leading to level 3, the leaf in column 3 is not in the target set: the path /a/a/a/a/a only reaches the leftmost leaf node. Analogously, any b edge leading to level $0 < i \le h$ in a given path adds the leaf node in column i to the path's target set. The second extreme case is the path /b/b/b/b/b whose target set contains all six leaves in DB_2. It is easy to see now that no two root-to-leaf paths in DB_2 have the same target set, so that a strong DataGuide for DB_2 needs to represent at least 2^{h-1} different target sets in its leaves. $\square$

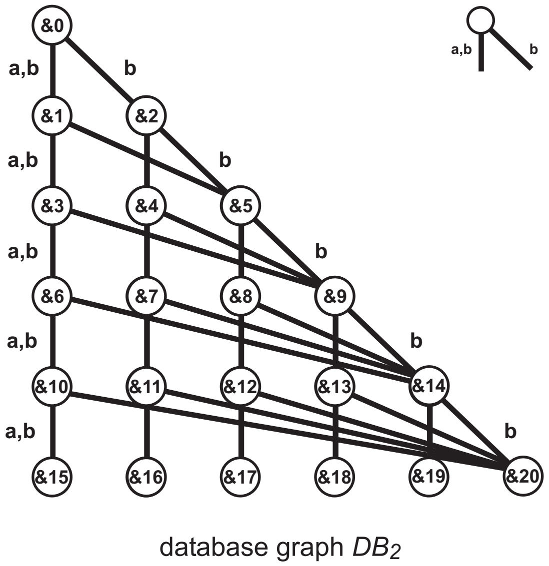


Figure 6.13: An acyclic database causing exponential growth of the strong DataGuide

Performance. Experimental results from [GW97a] show that in spite of its worst-case behaviour, the strong DataGuide performs quite well for various kinds of database. The best results were achieved for tree databases or regular acyclic graphs, where the storage overhead was well below 2% and the throughput during index creation ranged from nearly 3000 to more than 6000 database objects (nodes or links) per second. In the only experiment with a shallow cyclic graph, still regularly structured, the DataGuide performed almost equally well, with a throughput of about 2500 objects and storage overhead still less than 2%. Space requirements increased by an order of magnitude when the fan-out varied among the nodes in the database. In this case, throughput dropped to 168 database objects per second. For deeply nested graphs with backlinks, the DataGuide required up to 317% of the database size, indexing 250 objects per second.

Applications of the DataGuide. We briefly review two retrieval systems using the DataGuide as structure index.

LORE. LORE ([MAG+97], [MW97], [MWA+98]) is an acronym for *L*ightweight *O*bject *Re*pository. The system is based on the semistructured data model OEM (for *O*bject *E*xchange *M*odel, [PGMW95]). Meanwhile LORE has become a full-fledged database management system for semistructured data, including its own OQL-based query language, LOREL ([AQM+97]), a query plan generator, a query optimizer, import facilities, and a graphic user interface. [MWA+98] lists four indices in the system: the Vindex, which is a value index (see section 6.2.3.2), the Tindex, a text index (see section 6.2.3.1; not to be confused with the T-Index from section 6.4.4), the Lindex, which combines a label index (section 6.2.3.3) with a parent index (section 6.2.3.5), and the Pindex, which is a DataGuide. It also serves as a browsable database schema. Additionally, the Bindex ([MW97]) can be used to locate parent/child pairs by the linking edge's label.

ToX. ToX (for *T*oronto *X*ML *E*ngine, [BBM+01]) is a repository for XML data which supports multiple query languages, external queries, and different storage methods (relational, object-oriented, native storage systems such as LORE) depending on the properties of a document. For structure documents, an index structure based to the DataGuide and *Access Support Relations* ([KM92]) is used: the ToXin (for *TOX In*dex, [RM01], [Riz01]) is a weak DataGuide with a minimal number of nodes. Since simple paths with different target sets in the database are allowed to share the same index nodes, secondary parent/child indices are needed to reconstruct linked pairs of document nodes. To this end, so-called *instance tables* hold the IDs of parent/child pairs separately for each edge in the ToXin. Note that while the index is now linear in the size of the database even for cyclic documents, index paths no longer represent joins as described in section 6.1. Instead a separate look-up in the corresponding instance table is required for each edge traversal in the index graph. The instance tables collectively have the same functionality as the Lindex in the LORE system. The latter also allows to match paths starting at any level in the database tree, in much the same way as the instance tables. An in LORE, atomic content is handled by a separate text index.

6.4.3.4 Features

Data representation. As most navigational indices presented in this work, the DataGuide only takes into account the structural part of the database, leaving atomic content to a secondary index. For tree databases the strong DataGuide is equivalent to the schema tree presented in section 4.2. It is no longer restricted to a tree shape when cycles and node sharing occur in the documents. Simple paths sharing a node in the strong DataGuide are required to have the same target set in the database. No native support for multiple document graphs is provided.

Navigation. The DataGuide is designed for top-down query plans (see section 2.2) in conjunction with a secondary content index. It only supports top-down navigation starting from the root. Neither relative nor any other regular paths are admitted. As mentioned in section 4.2, ancestor/descendant relationships between individual document nodes cannot be identified either. Hence either a secondary index or structural node identification is recommended. The ToX index, which is based on the DataGuide, strives to enable bottom-up navigation as well as relative paths using multiple parent/child indices (see the end of the previous subsection).

Path templates. Although no template mechanism has been proposed for the DataGuide so far, the IndexFabric's *refined paths* (see section 6.4.5.3) actually are atomic templates on an index structure similar to the DataGuide. Templates containing placeholders, like the ones used with the T-Index in section 6.4.4, might be adapted, too. A suitable algorithm would then generate a DataGuide over the subset of paths in the database which match a given template expression, analogous to the T-Index. Again, this issue has not been mentioned in the cited literature. However, [CS01] describes a way to index relative paths with the IndexFabric. Although no placeholder templates are used, the structure of the resulting index is comparable to that of a 2-Index.

Node identification. The DataGuide is independent of – and therefore compatible with – any node identification scheme. In particular, to handle indirect containment without upward navigation, it can be combined with interval encoding (see section 5.5.2) and the corresponding join operations discussed in section 6.2.3.6. Alternatively, the Virtual Nodes scheme from section 5.5.3 may be used to identify parent/child pairs without a secondary index, as required by the ToX index. Both are useful e.g. when combining the results from multiple path look-ups to answer a tree query given to the query engine.

Index update. The update algorithm given in [GW97a] is very similar to the one used for building DataGuides. It allows to insert entire subgraphs into the database, which are then indexed recursively. This is valuable e.g. when a new document is added to the collection. Some persistent data structures initialized during index creation make sure that only those parts affected by the modification are re-examined. More specifically, adding or deleting an edge in the database graph may alter the partition induced by the path equivalence relation $\equiv_t$ (see section 6.4.3.3), causing new nodes and edges to be created and others to be removed from the DataGuide. Where this is not the case, however, nothing is changed.

Updates of the DataGuide, although incremental, may be costly because individual document nodes can be referenced at multiple points in the index structure. Hence more than one path may need to be examined at the risk of expensive I/O accesses to secondary storage devices. Notice that since content is not represented in the index, modifying atomic values in the database does not entail an update of the DataGuide (but maybe of the content index).

Storage. Although its worst case space requirements are exponential in the size of the database, the above-mentioned experimental results show that the DataGuide's performance depends dramatically on the structure of the documents. If they contain only few distinct paths and no cycles, the index structure may be held in main memory. On the other hand, deeply nested documents with long-distance backlinks deteriorate both storage overhead and indexing time.

When saving space is crucial, even at the cost of retrieval latency, the non-deterministic 1-Index may be a better choice. After all the disk accesses needed to query a fully grown DataGuide slow down the look-up procedure, too. A general T-Index can also be set up to index only predefined paths in the database, which further reduces the storage required. Alternatively, the IndexFabric may be considered, which is conceptually similar to the DataGuide and comes with a dedicated paging strategy (see section 6.4.5 for details).

6.4.3.5 Remarks

[GW97b] is a slightly abridged version of [GW97a], lacking the listing of the update algorithm as well as some details on the experimental results. As a database summary, DataGuides are based on *Representative Objects* which are introduced in [NUWC97]. The most prominent example of a retrieval systembased on a DataGuide as structure index is LORE (see section 6.4.3.3 above), which is described extensively in [MAG+97], [MW97], and [MWA+98].

We have hinted at how to use DataGuides together with node identification schemes. Different kinds of path template could be supported, too. So far no formal criteria or heuristics have been proposed to assess the complexity of the DataGuide resulting from a given database.

As already mentioned, several other approaches are closely related to the DataGuide, which itself resembles the Trie known from Information Retrieval (see section 3.4). While the IndexFabric (see section 6.4.5) and ToXin index (see subsection 6.4.3.3 above) have been designed to enhance the DataGuide's storage and navigation, respectively, the T-Index can be considered an alternative solution to the schema graph problem, which is discussed in section 4.2.

6.4.4 T-Index

6.4.4.1 Abstract

The *T-Index* ([MS99b]) is a non-deterministic structure index for both tree and graph databases. Tailored to fit queries matching a given path template, it supports only a specific class of look-ups, but with a minimum of storage wasted. Special cases of the T-Index include the *1-Index*, for indexing all absolute paths, and the *2-Index*, which covers relative paths. The 1-Index is equivalent to the DataGuide when indexing tree databases, but even for non-trivial graphs grows only linearly in the size of the database.

6.4.4.2 Index interface

1-Index	*SimplePaths*	$\longrightarrow$	$\mathcal{P}(\,\mathit{NodeIDs}\,)$
2-Index	*RelativePaths*	$\longrightarrow$	$\mathcal{P}(\,\mathit{NodeIDs}\,)$
T-Index	*PrivilegedPaths*	$\longrightarrow$	$\mathcal{P}(\,\mathit{NodeIDs}\,)$

Table 6.9: Look-up patterns for the T-Index

The look-up patterns for the three indices differ in the type of path expressions they accept as input. While the 1-Index is restricted to simple paths, like the DataGuide, the 2-Index supports relative path queries. The T-Index can be given any regular path expression which matches the template used to build the index, as described in the next subsection. Other requests may be handled if they can be rewritten appropriately, as discussed by [MS99b].

6.4.4.3 Details

Data structures and look-up procedure. Section 5.3 explains how navigational index structures can be optimized for certain classes of queries (so-called *privileged queries*) using path templates of the form $\mathsf{T}_0\, x_0 \ldots \mathsf{T}_k\, x_k$. Some path fragments may be represented in a symbolic form, which not only makes the index smaller but also reduces the number of navigation steps when evaluating privileged queries. With the *T-Index*, [MS99b] introduces an index structure which is modelled entirely after a single path template given at creation time. According to the structure of the template, there are three kinds of index optimized for different query classes: the *1-* and *2-Index*, whose templates are fixed, and the T-Index proper which accepts any path template and therefore subsumes the former two from a structural point of view.

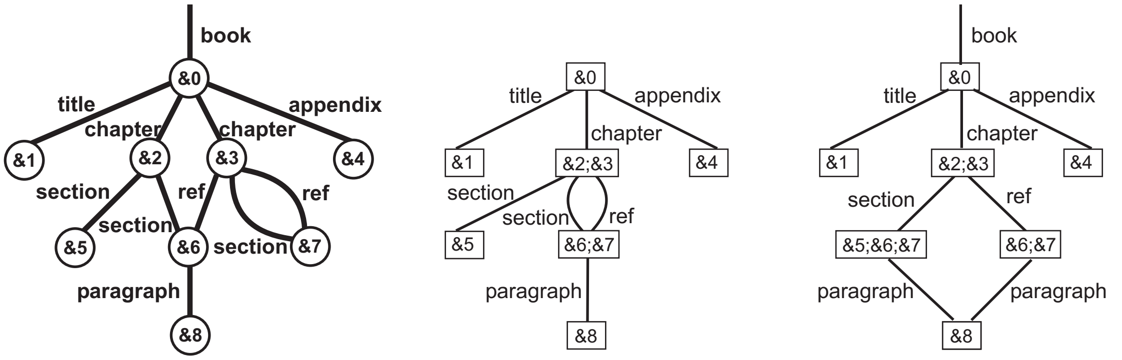
All T-Indices have in common that only label paths matching one of the privileged queries are represented in the index structure, while paths not conforming to the template are left out. During structure matching the variables in the template are bound successively in top-down direction. A set of node ID tuples is retrieved in the end, each referring to nodes within the same document which together match the query. The T-Index is non-deterministic in the sense that the same label path may occur multiple times in it. Hence the index graph must be traversed with backtracking to find all relevant path occurrences.

In the following we review the structure of the 1-Index, 2-Index, and T-Index in more detail, along with their respective look-up procedures and a simple example for each of the three. However, we omit the complete formalization which can be found in [MS99b].

1-Index. The 1-Index is based on the template $\boxed{\mathsf{P}}\, x$, which privileges all regular path queries. In other words, the 1-Index is not optimized for any specific query class. Hence all label paths in the database are represented in full length, starting from the database root. The 1-Index is therefore suitable for retrieving simple paths, although other queries (e.g. for relative paths) can be evaluated with a full index scan, as with most navigational indices. Since the path template contains only one variable, the 1-Index retrieves a set of 1-tuples (i.e. singleton IDs) locating the matching document nodes. Each path in the 1-Index is annotated with references to members of its target set in the database tree. The whole target set may need to be collected from multiple index nodes, however, due to non-determinism.

The structure of the 1-Index is based on an equivalence relation $\equiv_{p_1}$ defined on the set N of all document nodes. For any $n_1, n_2 \in N$, let $n_1 \equiv_{p_1} n_2$ *iff* n_1 and n_2 are accessible via the same set of simple paths. Each of the equivalence classes $[n]_{\equiv_{p_1}}$ is represented by a different node in the index. It is annotated with references to all members of the equivalence class and can be reached from the index root via the same set of paths as any of these document nodes (note that by definition the set is identical for all of them). Since the $[n]_{\equiv_{p_1}}$ together form a partition of N, each document node is referenced exactly once in the 1-Index. Note that for tree-shaped databases, the strong DataGuide's equivalence relation $\equiv_t$ and $\equiv_{p_1}$ coincide. Hence in this special case, the two index structures are equivalent.

Figure 6.14 below shows a database graph *(a)* with the corresponding 1-Index *(b)* and strong DataGuide *(c)*. Obviously there are three differences between the two index structures. First, the 1-Index (as all T-Indices) omits the root label root. Second, every document node ID (&0–&8) appears exactly once in *(b)*, whereas *(c)* contains &6 and &7 twice. Conversely, in the 1-Index the path /chapter/section is duplicated, while the DataGuide's path representations are unambiguous. In contrast to this duality, both have in common that hierarchical relationships between index nodes cannot be extrapolated to the document nodes they reference, as pointed out in section 4.2.



(a) database graph *(b)* 1-Index for *(a)* *(c)* strong DataGuide for *(a)*

Figure 6.14: 1-Index vs. DataGuide

As mentioned above, the 1-Index contains all label paths from the database. Hence it is possible to evaluate relative path queries, although the index has not been designed to do this efficiently. Such a look-up may involve scanning the whole 1-Index, whereas with a 2-Index the search is restricted to a single index path.

2-Index. A T-Index built over path expressions which satisfy the template $\_^*\, x_0 \boxed{\mathsf{P}}\, x_1$ is called 2-Index. Unlike the 1-Index, whose path template could also be written $\epsilon\, x_0 \boxed{\mathsf{P}}\, x_1$, the 2-Index privileges label paths at arbitrary depth. As a consequence, not only absolute but also relative paths are indexed. In this sense the 2-Index is more general than the 1-Index. The T-Index, by contrast, can be built with the same path template as the 2-Index or more restrictive ones, in which case it covers only a subset of the paths in the database.

One can think of the 2-Index as a path-based ancestor/descendant index, retrieving all pairs of nodes in the database which are linked by a given label path. Notice that relationships between individual document nodes need not be double-checked, as e.g. with the 1-Index or DataGuide, because the 2-Index returns pairs of IDs separately binding x_0 and x_1, i.e. the ancestor and descendant, respectively. Pairs representing absolute paths in the database can be recognized by their first component, which is &0.

Analogously to the 1-Index each node in the 2-Index represents an equivalence class $[(n_1, n_2)]_{\equiv_{p_2}}$ where $(n_1, n_2) \equiv_{p_2} (n_1', n_2')$ *iff* n_1 and n_2 are linked by the same set of label paths as n_1' and n_2', for any $(n_1, n_2), (n_1', n_2') \in N \times N$. The members of an equivalence class annotate the corresponding node in the index graph. Hence when a pair of node IDs is found after path traversal, we know that the two referenced document nodes are linked by this path in the database, and additionally by all others reaching this pair in the index. Again, finding all hits for a given label path may involve backtracking.

Figure 6.15 exemplifies this with the 2-Index for the database from figure 6.14 *(a)*. Any query path, be it relative or not, is matched top-down from the root to find the right pairs of node IDs. For instance, the annotation of the node at the bottom of the figure asserts that in the database, node &8 can be reached from the root &0 by traversing either /chapter/section/paragraph or /chapter/ref/paragraph. Likewise, the relative path section/paragraph is indexed as leading from &2 to &8 only. To retrieve all node pairs linked by a section edge in the database, we need to follow both section edges leaving the index root.

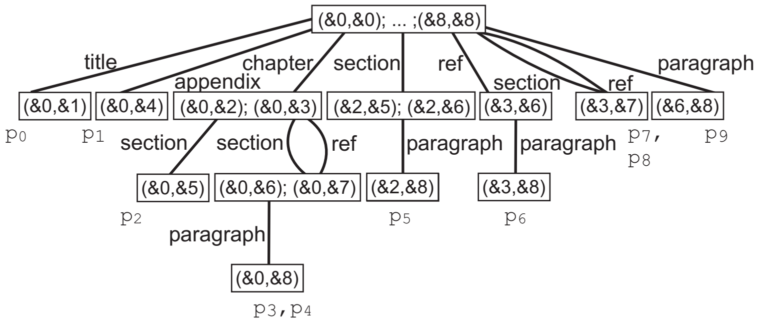


Figure 6.15: 2-Index

The root's annotation represents node pairs linked by the empty path ϵ. For later reference, we identify individual root-to-leaf paths in the index with names p_0 through p_9 as shown in the figure. Each of them represents a set of simple paths in the database instantiating the path template $\_^* \, x_0 \, \boxed{\mathsf{P}} \, x_1$. Grouping them according to which prefixes of the database paths matched the $\_^*$ part of the template during index creation, we see that for p_0 through p_4, $\_^*$ was matched by ϵ. As one can see from the first component of their annotating ID pairs, they stand for all absolute paths in the database. For p_5 through p_8, $\_^*$ was bound to /chapter, whereas for p_9, both /chapter/section and /chapter/ref matched $\_^*$. (Remember that all T-Indices omit the label of the database root.)

T-Index. The T-Index in its most general form accepts any template $\mathsf{T}_0 \, x_0 \ldots \mathsf{T}_k \, x_k$, where each T_i, $0 \leq i \leq k$, is either a constant regular path expression or one of the placeholders $\boxed{\mathsf{P}}$ and $\boxed{\mathsf{F}}$. As discussed in section 5.3, the placeholders span a subgraph in the index, covering all matching paths in the database from the nodes bound to the preceding variable (or the root if there is none). Constant regular path expressions, however, are represented symbolically as a single edge in the index graph. In a preprocessing step, the path query is compared to the template, and those parts matching a constant regular path expression step are replaced by the special symbol the corresponding edge is labelled with. Thus privileged queries traverse the whole regular path expression in a single navigation step, while queries not compatible with the path template fail. When large and complex expressions are replaced in this manner, the T-Index exhibits its true benefit, which is to enhance the evaluation of certain classes of path queries, albeit at the cost of generality.

Unlike the 1- and 2-Index, the nodes in the T-Index are annotated with partial bindings for the variables $x_0, \ldots, x_k$.[24] Hence the tuples annotating each path in the index contain between

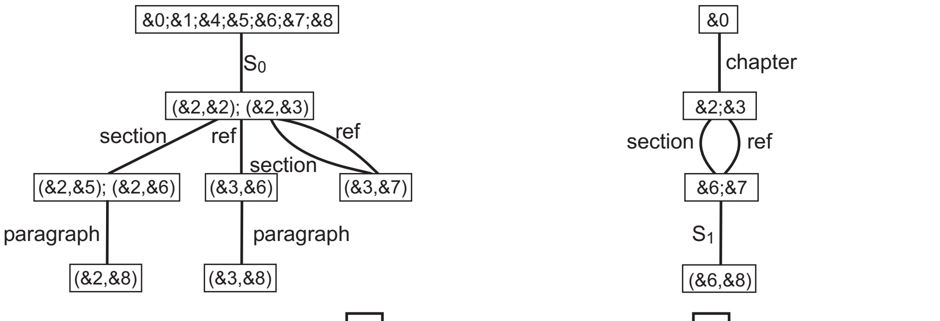
[24] Actually the 2-Index suppresses the step where singleton IDs would bind only the first of its two variables. Since the template $\_^* \, x_0$ matches any regular path expression, all node IDs from the database would annotate the corresponding index node, which is redundant.

one and k node IDs. Again the nodes in the T-Index represent equivalence classes induced by a relation $\equiv_{p_T}$ on the set of ID tuples, which is formalized in [MS99b]. Suffice it to say here that tuples annotating the same index node are equivalent in the sense that for each pair of successive variables in the template, the corresponding pairs of document nodes in these tuples are linked by the same set of label paths. Obviously $\equiv_{p_T}$ coincides with $\equiv_{p_1}$ or $\equiv_{p_2}$ when the templates $\boxed{\mathsf{P}}\,x$ or $\_^*\ x_0\,\boxed{\mathsf{P}}\,x_1$ are used, respectively.

The T-Index is also particular in that only nodes annotated with k-tuples contain query matches, i.e. those binding all variables in the path template. Consequently, all other nodes in the index need not be annotated at all. Yet in figure 6.16 below, where two T-Indices for the database from figure 6.14 *(a)* are shown, nodes binding less than k template variables also list the tuples from their corresponding equivalence class for reasons of clarity. In *(a)* this applies to the root node, whereas in *(b)* the three topmost nodes contain partial bindings. "Terminal" nodes are all those annotated with k-tuples of node IDs, i.e. pairs in both cases. We also omit some technical details irrelevant for the intuition of how the T-Index works, such as additional nodes needed to separate the variables' ranges in the matching path.

The T-Index shown in figure 6.16 *(a)* is based on the path template /chapter $x_0\,\boxed{\mathsf{P}}\,x_1$. From the root, which is annotated with the equivalence class of the database root &0, covering all nodes not reachable via /chapter, there is a single edge S_0 representing the regular path expression /chapter. In the query this prefix is replaced by S_0, too, so that x_0 is bound either to &2 or to &3, and matching continues with ID pairs. The subgraph below the second level is spanned by the placeholder $\boxed{\mathsf{P}}$. It encompasses just the paths p_5 through p_8 from figure 6.15, since these are the only ones representing document paths with the prefix /chapter. A query leading to the rightmost node, e.g., would yield (&3, &7) as the only pair of document nodes linked by both a section and a ref edge below the path /chapter.

The path template $\boxed{\mathsf{P}}\,x_0$ paragraph x_1 creates the T-Index in *(b)*, again for the database from figure 6.14 *(a)*. The subgraph spanned by $\boxed{\mathsf{P}}$ has been reduced to the only two paths reaching a "terminal" node, i.e. a full variable binding, via the S_1 edge representing paragraph.[25] The only absolute document paths satisfying this condition are p_3 and p_4 from figure 6.15. A look-up for `/chapter/section/paragraph`, translated to `/chapter/section/`S_1 in the preprocessing step, retrieves (&6, &8), where &6 is reached from the database root via /chapter/section and &8 from &6 via paragraph. By contrast, the query `/title/section/paragraph`, although conforming to the path template, yields no results.



(a) T-Index for **/chapter**$x_0\,\boxed{\mathsf{P}}\,x_1$ *(b)* T-Index for $\boxed{\mathsf{P}}\,x_0$ **paragraph**x_1

Figure 6.16: T-Indices for different path templates (simplified)

Enhancements. As pointed out above, all T-Indices are non-deterministic. To enhance navigation, a secondary index may be used, locating all outgoing edges of a given type for any node

[25] This minimization is part of the T-Index algorithm as proposed in [MS99b].

in the index graph.

[MS99b] stresses that while computing the equivalence relations $\equiv_{p_1}$, $\equiv_{p_2}$, and $\equiv_{p_T}$ is infeasible, there are refinements $\approx_{b_1}$, $\approx_{b_2}$, and $\approx_{b_T}$ based on *backwards bisimulation* which can be handled efficiently. Although they tend to produce more index nodes than the equivalence relations, which also entails needless edge duplication, each of the refinements coincides with its respective counterpart for databases where no node has multiple incoming edges of the same type. This means that in case of a tree database, e.g., using the refinements does not change the T-Index. Regardless of the structure of the database, the refinements do not affect the theoretical complexity results or any of the algorithms presented here.

Since any T-Index is restricted to paths matching a single template, the question arises whether queries involving other paths can still profit from the existing T-Indices in a retrieval system. In this context, [MS99b] discusses a partial solution to the *query rewriting problem*, called *prefix replacement*. Intuitively, queries with a prefix matching the path template of a T-Index can be rewritten so that the first part of the resulting path expression is supported by the index, while the remainder needs to be looked up in the database or other indices. Finding out whether a given query can be treated this way is a PSPACE-complete problem. If both the query and the template contain only constant regular path expressions with _, _*, and label names, it is even in PTIME.

6.4.4.4 Features

Data representation. All three variations of the T-Index are generally non-deterministic graphs with node sharing and cycles. Only for pure tree databases they become deterministic trees. The T-Index has been designed for use with a single database graph rather than separate document graphs.

Navigation. As most navigational indices, T-Indices are traversed in top-down direction during structure matching. Being non-deterministic, however, they require backtracking, unlike e.g. the DataGuide and its variations. A text or value index may be used in conjunction with any T-Index to support combined structure and content queries.

Hierarchical relationships between individual document nodes can be deduced for nodes binding subsequent variables in the same node ID tuple. For instance, in figure 6.15, while the annotation of the leaf in p_5 asserts that &2 reaches &8 via section/paragraph, the fact that the leaf's parent node is annotated with (&2, &5) does not imply that &8 reached from &5 via paragraph. In case of the 1-Index, paths in the index only reveal how certain document nodes can be reached from the database root, not from other document nodes. We have observed the same phenomenon in section 4.2 when examining descriptive schemata.

Path templates. The T-Index ows its name to the fact that it is built based on a single path template which determines its structure for a given database. The template is used to select a class of queries to be privileged during structure matching, as described in section 5.3. While the 1-Index and 2-Index are each restricted to a single fixed template, the general T-Index supports arbitrary combinations of constant regular path expressions and the two placeholders $\boxed{\mathsf{P}}$ and $\boxed{\mathsf{F}}$. Thus it can be customized to serve only the most frequent look-ups in a space-restricted environment. For example, if 15% of the queries are restricted to the paragraphs in a document, while 80% are absolute path expressions of any sort and the remaining 5% cover relative path queries, one may wish to create a T-Index for the template /chapter/section/paragraph x_0 $\boxed{\mathsf{P}}$ x_1 as well as a 1-Index. In another scenario, parts of the database which are fixed in structure may be indexed using standard DB techniques such as the B-tree, whereas queries concerning the semistructured part of the documents rely on one or several dedicated T-Indices, as suggested by [MS99b].

Node identification. Node identification schemes encoding hierarchical relationships between individual document nodes are useful with any T-Index, including the 2-Index, when the template's structure is too coarse to identify all desired nodes. For instance, searching for /chapter/section in a 2-Index built over a tree database involves a single look-up to retrieve pairs of nodes linked by this path. But a second look-up is needed to identify the intermediate node for each pair, i.e. the one reached by /chapter. Remember that each look-up may require backtracking. Eventually the results from both look-ups must be joined. With the Virtual Nodes encoding scheme, the second look-up and the join can be replaced by a trivial node ID computation. However, Virtual Nodes as well as interval encoding are restricted to tree databases. In case of general graphs, there are queries whose evaluation with a 2-Index requires almost as much look-ups and joins as with elementary indices. Of course, a T-Index for the template /chapter x_0 $\boxed{\mathsf{P}}$ x_1, like the one in figure 6.16 *(a)*, would bind x_0 to the intermediate node's ID right away. But unless this type of query is frequent, no such specialized index may be available. T-Indices modelled after certain other templates, e.g. /chapter/_ x_0 $\boxed{\mathsf{P}}$ x_1, might even be useless for this query.

Index update. [MS99b] does not discuss updating in detail. There seems to be no sophisticated algorithm for incremental maintenance of arbitrary T-Indices, but only a way to mark certain parts of the index as outdated. However, an incremental approach presented in [BKV91] is applicable to the T-Index if its template contains no placeholders, and then only for either insertions or deletions, but not both.

Storage. Obviously the size of a T-Index depends on the selectivity of its path template. Not surprisingly, the examples in figures 6.14 through 6.16 suggest that the 2-Index, as the most versatile of the three index structures, requires more space on average than the 1-Index or a T-Index with a sufficiently restrictive template. In the worst case, the size of the 2-Index is quadratic in that of the database, although [MS99b] argues that in two experiments, it consumed only up to two thirds of the memory needed by the database.

From a theoretical point of view, the 1-Index clearly beats the DataGuide in terms of storage consumption. However, as we have already pointed out in section 6.4.3.3, on average the DataGuide requires far less space than its worst-case behaviour indicates. Table 6.10 compares the space complexity of both 1-Index and DataGuide for different database structures. While the size of the former is linear in that of the database for any kind of graph, the latter may grow exponentially when node sharing or cycles are admitted. (See section 6.4.3.3 for examples.)

index	tree	acyclic graph	cyclic graph
1-Index	linear	linear	linear
DataGuide	linear	exponential	exponential

Table 6.10: Worst-case storage requirements of the 1-Index and strong DataGuide

[MS99b] states that under certain circumstances the size of both the 1-Index and the 2-Index no longer depends on the overall size of the database but merely on the maximum path length and the number of distinct labels. No details are given about the complexity of the general T-Index.

6.4.4.5 Remarks

The close relationship between the 1-Index and the DataGuide has already been mentioned in different places throughout this work. In case of a tree database, where both coincide, they can be regarded as Tries built over label strings (see section 3.4). The 2-Index, in turn, resembles a suffix tree where labels in a path are treated as symbols in a string. [MS99b] describes this analogy in more detail. Additionally, a connection between T-Indices and *Access Support Relations* (ASR) for the object-oriented data model is observed (see [KM92] for details on ASR).

Another navigational index structure involving path templates is the IndexFabric, described in the next section. Unlike the T-Index, it indexes privileged and non-privileged paths simultaneously. However, the IndexFabric only supports templates based on constant regular path expressions.

6.4.5 IndexFabric

6.4.5.1 Abstract

The IndexFabric ([CS01], [CSF$^+$01]) indexes both paths and content of tree databases in a balanced hierarchy of PATRICIA Tries. At each level the Trie is split into blocks suitable for efficient paging. A simple path look-up causes at most two blocks per layer to be fetched from disk. The IndexFabric supports path templates without placeholders.

6.4.5.2 Index interface

IndexFabric	*SimplePaths*	$\longrightarrow$	$\mathcal{P}(\,\mathit{NodeIDs}\,)$
	PrivilegedPaths	$\longrightarrow$	$\mathcal{P}(\,\mathit{NodeIDs}\,)$

Table 6.11: Look-up patterns for the IndexFabric

Look-ups in the IndexFabric may follow the typical pattern for navigational indices: a given simple path is mapped to a set of document node IDs. Additionally, queries instantiating one of a set of predefined regular path templates can be evaluated. In general, however, neither regular path expressions nor tree queries are supported by the index engine. As with other approaches, the query engine may evaluate them by joining the results of multiple path look-ups.

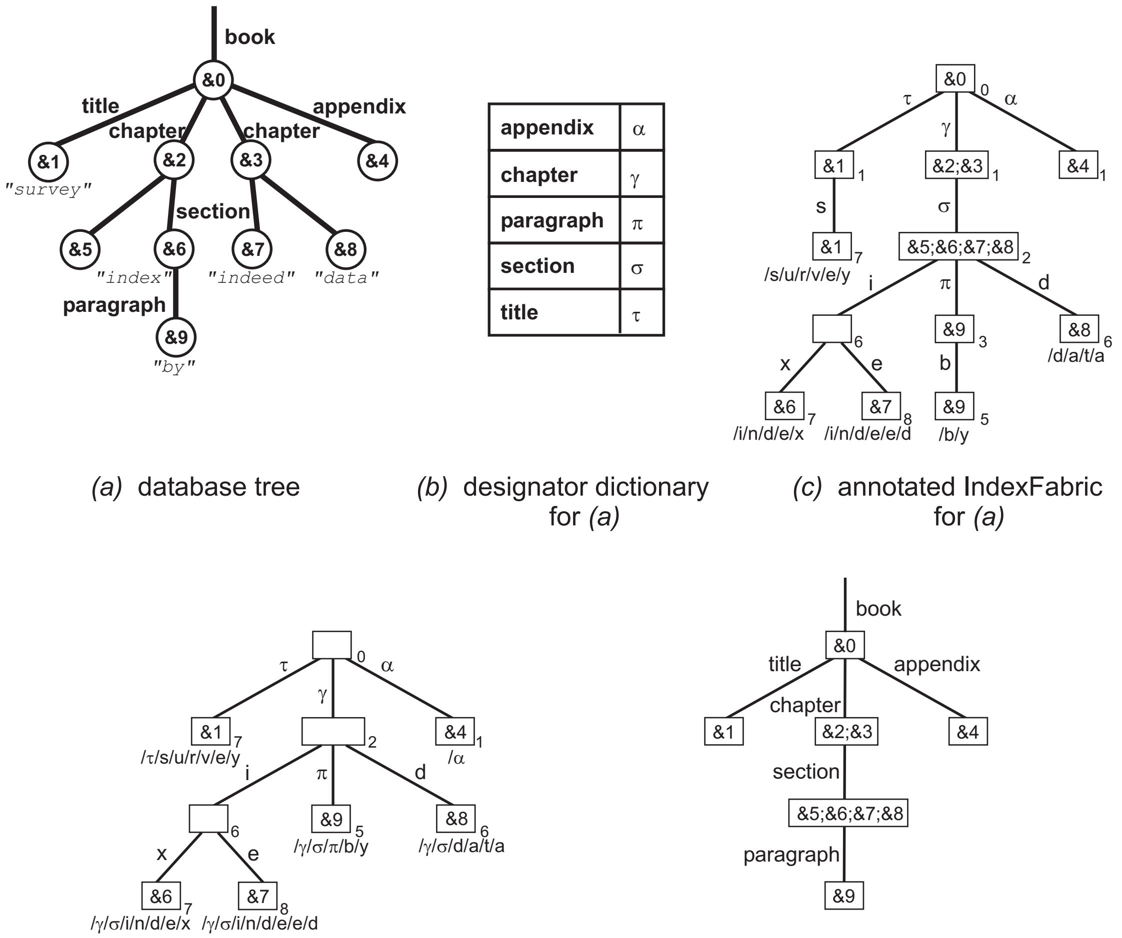
6.4.5.3 Details

The IndexFabric has been designed to index both the structure and the content of large databases. As shown in [CS01], it is compatible with a variety of data models, including the relational and object-oriented paradigm. In case of semistructured data, the IndexFabric closely resembles the DataGuide presented in section 6.4.3. In the cited literature, two features of the approach are claimed to make it amenable to very large data collections. First, it comes with a paging strategy which minimizes the number of disk accesses during a look-up. Second, being based on the PATRICIA Trie, the IndexFabric stores information in a compressed form. However, this may only be useful for the atomic database content, as we shall see below.

Data structure. The IndexFabric extends the concept of a PATRICIA Trie (see section 3.4) from character strings to simple path expressions. To this end, each label from the database is assigned a dedicated symbol, or *designator*, stored in a *designator dictionary.* The index structure is built over all simple paths with their textual content as follows.[26] Before inserting an individual document path into the IndexFabric, all its labels are replaced by their respective designators. If the last node in the path has textual content, the corresponding character string is appended to the designator-encoded label path. For instance, given the designator dictionary shown in figure 6.17 *(b)*, a document node reachable via /book/chapter/section which contains the keyword "data" would cause the path $/\gamma/\sigma/$d/a/t/a to be indexed. (For technical reasons, we have omitted the root label book.) If an intermediate node in the path has textual content besides its child node, which may occur when mixed content is allowed in the documents, then the path prefix up to this node is treated similarly as a separate path. Thus every path to be indexed (and, consequently,

[26] We only describe *prefix encoding* here, as opposed to the *infix encoding* mentioned by [CSF$^+$01].

every query expression to be evaluated) consists of a sequence of path designators, maybe followed by a sequence of ordinary characters. These strings of label and content symbols can be indexed in a PATRICIA Trie as shown in section 3.4. The result is a single-layer IndexFabric.



(a) database tree *(b)* designator dictionary for *(a)* *(c)* annotated IndexFabric for *(a)*

(d) IndexFabric for *(a)* *(e)* strong DataGuide for *(a)*

Figure 6.17: IndexFabric vs. strong DataGuide

As an example, consider the tree database in figure 6.17 *(a)* and the corresponding IndexFabric in *(c)*. Each simple path from the database is represented by exactly one index path. Additionally, when a document path reaches a node with textual content, then the content is appended to the corresponding index path in compressed form.[27] For instance, the document node &6 has mixed content: besides its paragraph child, it contains the word "index". &6 is accessible by the path /book/chapter/section in *(a)*, which becomes $/\gamma/\sigma$ in *(c)*, according to the designator dictionary shown in *(b)*. Hence the index node reached by $/\gamma/\sigma$ has, among others, one outgoing edge representing the paragraph child, labelled π, and another one representing the word "index" in PATRICIA Trie style, labelled i.

Note that since &6 and &7 have keywords with a common prefix of length 4 ("index" and

[27] Remember that the small numbers next to each index node indicate the level of this node in an uncompressed Trie.

"indeed", respectively), the i branch in the IndexFabric splits up at the fifth letter ("x" and "e", respectively). The common prefix is not represented in a PATRICIA Trie. As a result of this lossy compression, query paths not represented in the IndexFabric may still match, leading to node references which are not in their target set. Searching for `/book/chapter/section["inbox"]` in figure 6.17 *(c)*, e.g., one erroneously finds a reference to the document node `&6` because the IndexFabric stores only those symbols needed to disambiguate between "index", "indeed", and "data". To remedy this defect, which makes the IndexFabric inexact, node IDs are stored together with the full label path reaching the referenced node in the document. From this supplementary information the skipped keyword symbols can be reconstructed.

Actually the index structure shown in *(c)* is a variation of the original IndexFabric. According to [CS01] and [CSF$^+$01], only leaf nodes of the IndexFabric contain references to document nodes, like in *(d)*. However, this precludes the use of the IndexFabric for pure structure queries: while a query for `/book/chapter` does not return the desired IDs `&2` and `&3` because these nodes are represented by an inner node in the index, the last label in `/book/chapter/section` cannot even be found without accessing all leaf nodes below the γ edge (which still does not retrieve the right references). The reason is that due to PATRICIA Trie compression, the section step has been omitted in the indexed paths and can only be reconstructed from full paths stored with the leaf annotations. Hence there is no place (not even an inner node) to put a reference to the empty node `&5`. While compressing the structural part of the database this way makes the index graph more compact, compared to *(c)*, it is obvious that the resulting index is too restricted, lacking references to part of the database. Therefore we focus on the *annotated IndexFabric (c)* in the following, which compresses atomic database content but not structure, and stores node IDs in inner index nodes, too.

Figure 6.17 *(e)* depicts the strong DataGuide for the database in *(a)*. Apart from the root label, it is isomorphic to the upper part of the annotated IndexFabric *(c)* which represents hierarchical relationships, i.e. paths in the database. (Recall from section 6.4.3.4 that DataGuides do not index atomic database content.)

Enhancements. In the following paragraphs we review five optimization techniques proposed for the IndexFabric.

Precision optimization. Rather than storing full paths with each reference to a document node, one may consider labelling edges in the index with the full path fragment they represent, e.g. inde instead of i in figures 6.17 *(c)* and *(d)* above. The benefit of this approach is not so much that each label then stores a common prefix which is otherwise duplicated by all references below: unlike labels, leaf annotations usually reside on disk, hence storing full paths with the node references does not affect the retrieval performance. Yet the label-based disambiguation has the advantage that false hits can be detected before traversing the edge, which may avoid unnecessary disk accesses when the index is stored on a secondary device (see below).

Path templates. [CSF$^+$01] introduces a template mechanism called *refined paths* for the IndexFabric to enhance the evaluation of certain predefined queries. It can be regarded as a special case of the path templates presented in section 5.3, which are used by the T-Index (see section 6.4.4). In terms of this general mechanism, the IndexFabric supports only templates without placeholders, i.e. those consisting of constant regular path expressions, but lacking $\boxed{\mathsf{F}}$ and $\boxed{\mathsf{P}}$ steps. As explained in section 5.3, this subset of templates defines not query classes but only single privileged queries, and therefore is of limited use. Refined paths appear in the same IndexFabric tree as the non-privileged *raw paths*, but with reserved designators. During the look-up, queries recognized as privileged are translated to expressions with the right designator, which directs them to the appropriate refined path in the IndexFabric.

Relative path queries. [CS01] suggests indexing relative paths in addition to simple paths in much the same way as the 2-Index (see section 6.4.4). However, experimental results are needed to assess whether the resulting space requirements are acceptable, given that the IndexFabric also stores atomic database content.

Range queries. Unlike other navigational approaches, the IndexFabric does not rely on secondary indices to handle the textual content of the database. When an ordering relation over the content alphabet is given, range queries may be supported. For example, assuming the lexicographical order on characters, a search for `/book/chapter/section["data..indeed"]` on the IndexFabric in figure 6.17 *(c)* would first locate the reference to node `&8` below the `d` edge before backtracking to the next higher level in the Trie. In the end, `&7` would also be found, but not `&6` because the `x` edge does not fit the range.

Layer-based paging strategy. A major contribution of the IndexFabric is its layered approach to indexing large structured databases. Built over both structure and content of the database, the size of the IndexFabric easily exceeds main memory capacity. [CS01] proposes a paging strategy which can be applied to Tries in general. The original index structure stored on disk is supposed to be divided into multiple blocks of roughly equal size, each of which holds a smaller sub-Trie. The idea is to create an additional Trie indexing the blocks of the first one. Successively building such meta-indices results in a balanced hierarchy of Tries[28] (more particularly, IndexFabrics in our case) decreasing in size, the root of which fits a single block.

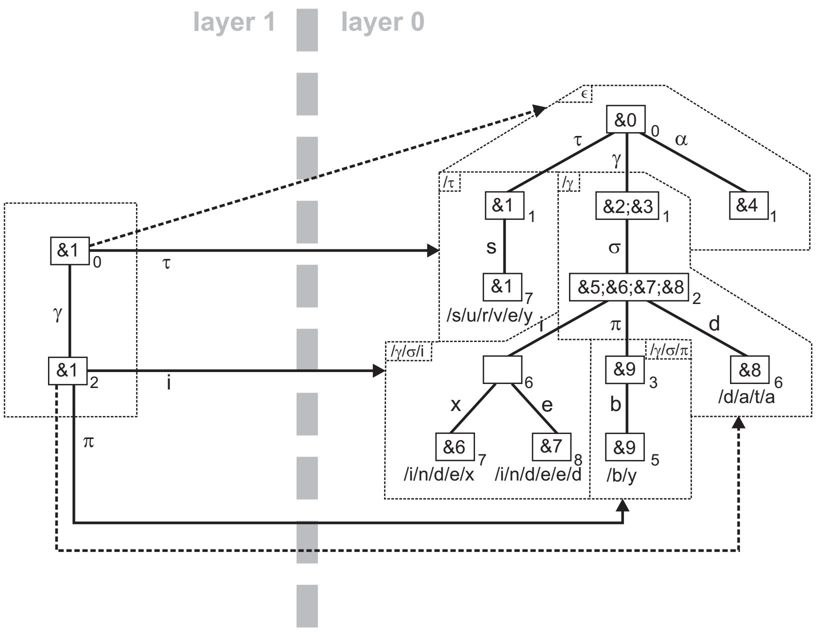


Figure 6.18: Layered IndexFabric

As figure 6.18 illustrates, the index hierarchy can be visualized as growing in the horizontal axis from right to left, i.e. orthogonal to the original IndexFabric, which resides at the bottom (i.e. rightmost) level (layer 0). For instance, the IndexFabric from figure 6.17 *(c)* may induce a two-layer hierarchy (though the actual size of the blocks determines how many layers are needed for a given index). On layer 0, each block contains a sub-Trie of the original IndexFabric. The

[28] Note that not the individual PATRICIA Tries on the different layers are balanced, but the overall hierarchy.

uncompressed path leading to the root of such a sub-Trie is attached to the containing block for disambiguation (see below). Since all paths represented in the sub-Trie begin with this string, it is called the *common prefix* of that block. The Trie on layer 1 represents those nodes of the original index with outgoing edges crossing a block border (so-called *split links*). They are copied to the next higher layer and point to the block they originate from via an unlabelled *direct link*. Direct links are shown as dotted arrows in the figure. In our example, there are two nodes on layer 0 to be copied to layer 1: the index root and the node below the σ edge. Their copies on layer 1 have direct links to the blocks with the common prefix ϵ (the empty path) and $/\gamma$, respectively. One may regard the copied nodes as representations of the two layer 0 blocks on layer 1. Additionally, for each split link, the copied node has a *far link* with the same label leading to the block whose border is crossed by the split link. Far links are depicted as solid arrows. For instance, the upper node on layer 1 (i.e. the copy of the index root on layer 0) is far-linked via τ to the block with the $/\tau$ prefix. Besides, it is connected to the second node on layer 1, which resides in the same block, via a *near link* labelled γ. This near link can be considered replacing the far link which would be necessary if the $/\gamma$ block were not represented by a dedicated node on layer 1.

To sketch query evaluation on the layered IndexFabric, suppose we are searching for paths matching `/book/chapter/section["index"]`, which translates to $/\gamma/\sigma/$`i/n/d/e/x`. First, the root node of the only block on the leftmost layer (layer 1) is examined. Of course it represents nodes at level 0 in the database, so we try to match its outbound edges to the first label in the query path. γ leads us to the lower node on layer 1. Now we examine its outgoing links to match the second label in the query path. Trying to find a σ edge fails, so we need to follow the direct link to the block with the common prefix $/\gamma$, which also is a prefix of the search path. The block's sub-Trie root represents document nodes at level 1, so again we look for the second edge in the query path, σ, and afterwards for one labelled i which leads us to the block prefixed $/\gamma/\sigma/$i. Its root node represent nodes at level 6 in the database, so we skip the next three steps in the query path and try to match the last one, x. Finally this yields a reference to the document node &6, which also matches the desired keyword, "index". Notice that searching for "inbox" would have led us here, too, to find out eventually that the reference to &6 is not valid for this keyword. By contrast, with the next-to-last edge labelled inde instead of i, as suggested above, we would have known before fetching the last block.

Performance. The layered IndexFabric is *balanced* in the sense that for query paths of arbitrary length, there is a fixed number of layers to be accessed. [CS01] explains that although usually only one block per layer needs to be examined, the PATRICIA Trie's lossy compression may cause a single *block miss*. Thus in the worst case two blocks per layer must be fetched. Nevertheless the number of I/O operations involved in querying an IndexFabric on the disk is linear in the number of horizontal layers, which in turn is not directly correlated with (and usually much less than) the size of the database. To give an idea of how large an IndexFabric might become, [CS01] estimates that a three-layer hierarchy can index 10^9 occurrences. Note that while the lowest layer is likely to be stored on disk, containing the most blocks, upper layers in the hierarchy may well fit main memory.

[CSF$^+$01] samples both the evaluation time and the number of disk blocks fetched for five different queries, which range from simple path expressions over ancestor/descendant to tree queries. Note that all queries access leaf nodes so that the problem of missing references in inner index nodes does not arise. The experiments are carried out on a corpus of about 180,000 documents. The results for the IndexFabric are compared to those achieved with B-trees for two different mappings to the relational data model. In the first case, a *basic edge mapping* creates two tables for document roots and edges; in the second case, regular parts of the document structure are extracted heuristically by the STORED system described in [DFS99]. As could be guessed, the navigational index performs better than the RDBMS when multiple joins on tables are required, and STORED's performance declines when data are irregularly structured. The observation that efficiency both in terms of evaluation time and I/O operations increases dramatically when refined paths are used is in some sense a foregone conclusion, because apparently the corresponding templates have been tailored to fit the given queries. For a brief discussion of how valuable query pre-evaluation using refined paths is, see section 5.3.

In neither of the cited papers the IndexFabric is compared to other navigational indices. In particular, assessing the performance gain over the DataGuide and T-Index for large databases would be interesting.

6.4.5.4 Features

Data representation. In its simplest form the IndexFabric it is a PATRICIA Trie indexing combined label and character strings. As such it is suitable for tree databases and possibly directed acyclic graphs in general, but not for cyclic graphs. As both structure and content are kept in the same data structure, there is no need for a secondary content index.

The layered IndexFabric is a hierarchy of IndexFabrics where each pair located at adjacent layers is interconnected by two different kinds of link: labelled far links, and unlabelled direct links. On each layer, the IndexFabric is divided into blocks which may correspond to disk blocks when the index is stored on a secondary device. Both the simple and the layered IndexFabric assume a single database graph rather than separate document graphs.

Navigation. As most navigational indices, the IndexFabric is designed for top-down navigation. Since no secondary context index is used, there is only a single combined look-up for structure and content. Compare this to the other navigational indices which embed two separate look-ups in a top-down query plan. Evaluating range or tree queries with the IndexFabric may involve backtracking and hence require provisions for upward navigation.

Path templates. The IndexFabric supports path templates without placeholders. In database terminology, they can be compared to materialized views on the document node references. The pre-evaluated hits are inserted as *refined paths* into the same IndexFabric as the non-privileged *raw paths*. Following [CSF$^+$01]'s argument, the missing generality of refined paths, compared to T-Index templates, can be compensated for by the sheer number of privileged queries (but see the discussion of storage issues below).

Node identification. Since the IndexFabric does not inherently support tree queries, structural node identification using numbering schemes such as interval encoding (see section 5.5.2) or Virtual Nodes (see section 5.5.3) may be valuable. They can enhance the evaluation of complex queries as described in section 6.2.3.6.

Index update. [CS01] describes in detail how to insert new paths into the layered IndexFabric. Since the hierarchy is balanced, in the worst case an insertion may cause all blocks in the horizontal path leading to the insertion point to be splitted. Analogously, removing paths from the index can result in block merging, which is not discussed in the paper. In any case, the IndexFabric can be adapted to database changes incrementally as long as no additional constraints are introduced by node identification schemes.

Storage. [CSF$^+$01] argues that thanks to PATRICIA Trie compression, the IndexFabric's storage requirements remain modest even for large databases, so that many refined paths can be indexed additionally at almost no expense. Indeed inserting long paths into the index does not immediately affect the index size and thus its performance, like with the DataGuide or T-Index, for example. Compression also keeps the hierarchy shallow, removing chains of nodes with a single child each. However, as we have shown above, annotating the IndexFabric with references in inner index nodes precludes PATRICIA Trie compression for the structural part of the index. Besides, in real-world databases deeply nested trees with only one node per level are rare, so path compression is unlikely to contribute much to an overall reduction of the index size anyway. When compression is restricted to the content part of the index paths, the structural part of the IndexFabric requires at least as much space as the corresponding DataGuide. Nevertheless, the sophisticated paging strategy of the IndexFabric (which might also be applicable to other navigational indices) is a considerable advantage when indexing huge document collections.

We have already stressed that the IndexFabric is conceptually similar to the DataGuide. In particular, both indices are deterministic, representing each simple path from the database only once. As shown in section 6.4.3.3, this may cause exponential growth for graph databases in the worst case. Hence the IndexFabric incurs the (albeit relatively low) risk of storage explosion, just like the DataGuide. While this does not mean that the IndexFabric, all the more the layered one, cannot lead to a significant efficiency gain in secondary storage, it may limit the use of additional features like refined paths or relative path indexing in a 2-Index fashion, as suggested by [CS01].

6.4.5.5 Remarks

Details on the IndexFabric's data structure, both simple and layered, are given in [CS01] and [CSF$^+$01]. The latter paper additionally introduces the concept of refined paths and presents the experimental results cited above. [CSS02] complements them with a setup for assessing parallelized query evaluation with the IndexFabric, which is not discussed here. Finally, [SCF$^+$02] is a short overview reviewing the basics of the layered IndexFabric.

Unlike the other indices presented in this work, the IndexFabric is a patented approach with a commercial background. Although not stated explicitly, it seems to be based on the JS data structure introduced in a technical report earlier, which is cited by [CSF$^+$01].

We have mentioned above that though PATRICIA Tries are usually defined for tree databases only, one could examine whether they can be generalized to arbitrary directed acyclic graphs. Furthermore, the verification mechanism described to compensate for the lossy PATRICIA Trie compression might be enhanced if labels in the index included the skipped path steps, as suggested above.

Finally, [CS01] describes a variety of applications of the IndexFabric in combination with relational or object-oriented data, such as indexing symmetric relations or relative paths. However, not all of them can be applied to semistructured data.

Chapter 7

Conclusion

In this work a variety of approaches to indexing semistructured data has been presented, covering the period from the mid-nineties through 2002. As a background for their description and comparison, we have reviewed models for structured documents and basics from Information Retrieval. Moreover a generic retrieval system architecture has been sketched along with a unified terminology for all indexing approaches. Based on these preliminaries we have inferred a set of comparison criteria. According to the underlying representation of structural hierarchies in the data – a novel feature compared to traditional index structures from both Information Retrieval and datase research – we propose to classify the approaches into *elementary indices*, *path look-up indices*, and *navigational indices*. Each class comes with a characteristic *index interface* which comprises a set of *look-up patterns* described in terms of input and output parameters. The individual index structures range from very simple tables to complex graphs. Most of the more sophisticated approaches make use of simpler ones as secondary indices, however. We have also seen cases where features of a given index (e.g. the T-Index's path templates, 6.4.4) can be applied to some of its "peers" as well. In fact, this observation holds for the whole field: while there are rarely completely new ideas at the heart of any approach, combinations of well-understood IR and DB techniques have turned out to be most promising. Examples of this synergy effect include, among others, the IndexFabric (6.4.5) which combines the PATRICIA Trie (3.4) with B-tree-style paging, or path templates (5.3) which resemble views in a traditional database management system.

The detailed analysis of this broad range of index structures has also revealed trade-offs between the benefits pursued by the individual approaches. While each of them makes a contribution to a specific issue regarding performance or query support, it is impossible to reconcile them all in a single index structure. For instance, enhancing a navigational index structure with either reduced storage overhead or deterministic navigation in mind produces such different, yet related approaches like the 1-Index (6.4.4.3) and the DataGuide (6.4.3), respectively. Others focus on weighting (BUS index, 6.4.1) or heuristics to optimize query evaluation (Signature File Hierarchy, 6.4.2; Context Index, 6.3.2). The 2-Index (6.4.4.3) strives to support powerful query expressions efficiently, at the expense of storage reduction. By contrast, other T-Indices need much less space, but are also less versatile. Accommodating both structure and content, the IndexFabric (6.4.5) needs a paging strategy and compression techniques to compensate for its storage demands, just like the BitCube (6.3.3).

We have mentioned several operative retrieval systems based on these indexing approaches. The XISS system (6.2.3.6) is built on elementary indices, while both ToX and LORE (6.4.3.3) make use of the DataGuide. In particular, the latter illustrates that for practical usage, worst-case complexities are not always decisive.

Of course this work is not exhaustive, in the sense that some approaches to indexing semistructured data have been omitted for lack of both time and space. For example, we have not covered the nagivational Proximal Nodes index ([NBY97]), the paging strategy of the NATIX repository ([KM00]), and a recent approach to updating based on Relative Region Coordinates ([KYU01]). Besides, schema-assisted indexing techniques have not been taken into account (with one exception, see section 6.3.2.3), as this topic is not yet well established. Hence the survey is still incomplete. Notwithstanding we estimate the most prominent, and interesting, techniques have been considered to give a detailed overview of the current state of the art. Future work could be dedicated to investigate alternative approaches from the fields of deductive databases and compilers.

Bibliography

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, pages 1–18, 1997.
WWW: http://citeseer.nj.nec.com/abiteboul97querying.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/abiteboul97querying.pdf

[AMFL$^+$95] T. Arnold-Moore, M. Fuller, B. Lowe, J. Thom, and R. Wilkinson. The ELF Data Model and SGQL Query Language for Structured Document Databases. In *Proceedings of the Australasian Database Conference*, pages 17–26, 1995.
WWW: http://www.mds.rmit.edu.au/ msf/papers/adc95.pdf
Local:

[AQM$^+$97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
WWW: http://citeseer.nj.nec.com/abiteboul97lorel.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/abiteboul97lorel.pdf

[BBM$^+$01] Denilson Barbosa, Attila Barta, Alberto O. Mendelzon, George A. Mihaila, Flavio Rizzolo, and Patricia Rodriguez-Guianolli. ToX – the Toronto XML Engine. In *Workshop on Information Integration on the Web*, pages 66–73, 2001.
WWW: http://citeseer.nj.nec.com/barbosa01tox.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/barbosa01tox.pdf

[BKV91] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter. A Data Structure for Arc Insertion and Regular Path Finding. In *Annals of Mathematics and AI*, volume 3, pages 187–211, March 1991.

[Bun97] Peter Buneman. Semistructured Data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.
WWW: http://citeseer.nj.nec.com/buneman97semistructured.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/buneman97ssd.pdf

[BY94] R. Baeza-Yates. An hybrid query model for full text retrieval systems. Technical Report DCC-1994-2, Dept. of Computer Science, Univ. of Chile, 1994.
WWW: ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/hybridmodel.ps.gz
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/baeza94hybrid.pdf

[CA98] Yangjun Chen and Karl Aberer. Layered Index Structures in Document Database Systems. In *Proceedings of the Seventh International ACM Conference on Information and Knowledge Management (CIKM)*, pages 406–413, November 1998.

WWW: http://xml.darmstadt.gmd.de/cc-publications.html#4
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/chen98layer.pdf

[CA99] Yangjun Chen and Karl Aberer. Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases. In *Database and Expert Systems Applications*, pages 473–484, 1999.
WWW: http://citeseer.nj.nec.com/390292.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/chen99combining.pdf

[CCB95] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal*, 38(1):43–56, 1995.
WWW: http://citeseer.nj.nec.com/clarke95algebra.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/clarke95algebra.pdf

[CS01] Brian Cooper and Moshe Shadmon. The Index Fabric: Technical Overview. Technical report, RightOrder Inc., 2001.
WWW: http://www.rightorder.com/Technology%5FWhitePaper.html
Local:

[CSF+01] Brian 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, 2001.
WWW: http://citeseer.nj.nec.com/cooper01fast.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/cooper01fast.pdf

[CSS02] Brian Cooper, Neal Sample, and Moshe Shadmon. A parallel index for semistructured data. 2002. Presented at the 17th ACM Symposium on Applied Computing (SAC).
WWW: http://www.rightorder.com/Technology%5FWhitePaper.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/cooper02parallel.pdf

[DFS99] Alin Deutsch, Mary Fernández, and Dan Suciu. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.
WWW: http://citeseer.nj.nec.com/deutsch98storing.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/deutsch99stored.pdf

[Die82] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 122–127, May 1982.

[DML98] D. Dervos, Y. Manolopoulos, and P. Linardis. Comparison of Signature File Models with Superimposed Coding. *Information Processing Letters*, 65(2):101–106, 1998.

[Fal85] C. Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. In *Proceedings of the 1985 SIGMOD Conference*, pages 63–82, May 1985.

[FFK+98] Mary F. Fernández, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 414–425, 1998.

WWW: http://citeseer.nj.nec.com/fernandez98catching.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/fernandez98boat.pdf

[FFLS97] Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3):4–11, 1997.
WWW: http://citeseer.nj.nec.com/article/fernandez97query.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/fernandez97strudel.pdf

[FK99] Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, May 1999.
WWW: http://citeseer.nj.nec.com/florescu99performance.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/florescu99performance.pdf

[FMN02] Mary Fernández, Jonathan Marsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, April 2002.
WWW:

[Fre60] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[GBYS91] G. Gonnet, R. Baeza-Yates, and T. Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, Centre for the New OED, University of Waterloo, February 1991.
WWW: http://citeseer.nj.nec.com/gonnet91lexicographical.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/gonnet91lex.pdf

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB)*, pages 25–30, 1999.
WWW: http://citeseer.nj.nec.com/goldman99from.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/goldman99mig.pdf

[GT87] G. H. Gonnet and F. W. Tompa. Mind Your Grammar: a New Approach to Modelling Text. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 339–346, 1987.
WWW:
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/clarke95algebra.pdf

[GW97a] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Technical report, Stanford University, Computer Science Department, Database Group, 1997.
WWW: http://dbpubs.stanford.edu:8090/pub/1997-50
Local:

[GW97b] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.
WWW: http://citeseer.nj.nec.com/126680.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/goldman97dataguides.pdf

[KM92] Alfons Kemper and Guido Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, 17(2):117–145, 1992.
WWW: http://citeseer.nj.nec.com/kemper92access.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/kemper92access.pdf

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, 1993.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference On Data Engineering (ICDE)*, page 198, 2000.
WWW: http://citeseer.nj.nec.com/article/kanne99efficient.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/kanne00efficient.pdf

[KYU01] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 313–320, 2001.
WWW: http://citeseer.nj.nec.com/391959.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/kha01xml.pdf

[LM01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, 2001.
WWW: http://citeseer.nj.nec.com/454193.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/li01indexing.pdf

[LS00] Hartmut Liefke and Dan Suciu. XMill: an Efficient Compressor for XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.
WWW: http://citeseer.nj.nec.com/liefke99xmill.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/liefke00xmill.pdf

[LYYB96] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index structures for structured documents. *Digital Libraries*, pages 91–99, 1996.
WWW: http://citeseer.nj.nec.com/lee96index.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/lee96index.pdf

[Mac90] I. Macleod. Storage and Retrieval of Structured Documents. *Information Processing and Management*, 26(2):197–208, 1990.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.
WWW: http://citeseer.nj.nec.com/mchugh97lore.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/mchugh97lore.pdf

[Meu00] H. Meuss. *Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents*. PhD thesis, Dept. of Computer Science, University of Munich, 2000.
WWW: http://citeseer.nj.nec.com/meuss00logical.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/meuss00logical.pdf

[Mor68] D. R. Morrison. PATRICIA: Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MS99a] H. Meuss and C. Strohmaier. Improving index structures for structured document retrieval. In *Proceedings of the 21st Annual Colloquium on IR Research (IRSG)*, 1999.
WWW: http://citeseer.nj.nec.com/meuss99improving.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/meuss99impr.pdf

[MS99b] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 277–295, 1999.
WWW: http://citeseer.nj.nec.com/milo97index.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/milo97index.pdf

[MW97] J. McHugh and J. Widom. Query Optimization for Semistructured Data. Technical report, Stanford University, Computer Science Department, Database Group, November 1997.
WWW: http://citeseer.nj.nec.com/477475.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/mchugh97query.pdf

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, Database Group, January 1998.
WWW: http://citeseer.nj.nec.com/mchugh98indexing.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/mchugh98index.pdf

[NBY97] Gonzalo Navarro and Ricardo A. Baeza-Yates. Proximal Nodes: A Model to Query Document Databases by Content and Structure. *Information Systems*, 15(4):400–435, 1997.
WWW: http://citeseer.nj.nec.com/navarro97proximal.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/navarro97prox.pdf

[NUWC97] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchial Data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 79–90, 1997.
WWW: http://citeseer.nj.nec.com/nestorov97representative.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/nestorov97repres.pdf

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogenous information sources. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, volume 11, pages 251–260, 1995.

[PH01] Leela Krishna Poola and Jayant R. Haritsa. SphinX: Schema-conscious XML Indexing, 2001.
WWW: http://citeseer.nj.nec.com/poola01sphinx.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/poola01sphinx.pdf

[Riz01] F. Rizzolo. ToXin: An Indexing Scheme for XML Data. Master's thesis, University of Toronto, Canada, January 2001.
WWW:
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/toxinThesis.pdf

[RM01] Flavio Rizzolo and Alberto Mendelzon. Indexing XML Data with ToXin. In *Proceedings of the Fourth International Workshop on the Web and Databases (in conjunction with ACM SIGMOD)*, 2001.
WWW: http://citeseer.nj.nec.com/492738.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/rizzolo01index.pdf

[SCF+02] Neal Sample, Brian Cooper, Michael J. Franklin, Gísli R. Hjaltason, and Levy Cohen. Managing Complex and Varied Data with the IndexFabric, 2002.
WWW: http://citeseer.nj.nec.com/497317.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/sample02managing.pdf

[Sch01] Uwe Schöning. *Theoretische Informatik - kurzgefaßt*. Spektrum Akademischer Verlag, 2001. In German.

[SDAMZ94] Ron Sacks-Davis, Timothy Arnold-Moore, and Justin Zobel. Database Systems for Structured Documents. In *Proceedings of the International Symposium on Advanced Database Technologies and Their Integration (ADTI)*, 1994.
WWW: http://citeseer.nj.nec.com/491144.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/sacks94db.pdf

[Shi01] Dongwook Shin. XML Indexing and Retrieval with a Hybrid Storage Model. *Knowledge and Information Systems*, 3(2):252–261, 2001.
WWW: http://win-www.uia.ac.be/u/s975420/local/indexhybrid.pdf
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/shin01hybrid.pdf

[Sig] ACM Special Interest Group on Information Retrieval. Web site.
WWW: http://www.acm.org/sigir/resources.html

[SJJ98] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. In *Proceedings of the 3rd ACM International Conference on Digital Libraries*, pages 235–243, 1998.
WWW: http://win-www.uia.ac.be/u/s975420/local/bus.pdf
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/shin98bus.pdf

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[SM99] Christian Strohmaier and Holger Meuss. A Filter for Structured Document Retrieval. Technical Report 99-123, Centrum für Informations- und Sprachverarbeitung (CIS), University of Munich, Germany, 1999.
WWW: http://citeseer.nj.nec.com/220821.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/strohm99filter.pdf

[ST92] A. Salminen and F. W. Tompa. Pat expressions: an algebra for text search. *Papers in Computational Lexicography (COMPLEX)*, pages 309–332, 1992.

[Suc98] Dan Suciu. An Overview of Semistructured Data. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29(4):28–38, 1998.
WWW: http://citeseer.nj.nec.com/160105.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/suciu98ssd.pdf

[Vir] VirtuGrade Course in Information Retrieval. On-line tutorial. University of Tübingen (Germany), in cooperation with the Katholieke Universiteit Brabant in Tilburg (Netherlands) and the University of Munich (Germany).
WWW: http://pi0959.kub.nl/Paai/Onderw/Vir/index.html

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.
WWW: http://www.dcs.gla.ac.uk/Keith/Preface.html

[YRC01] Jong P. Yoon, Vijay Raghavan, and Venu Chakilam. Bitmap Indexing-based Clustering and Retrieval of XML Documents, 2001.
WWW: http://citeseer.nj.nec.com/459292.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/yoon01bitmap.pdf

[YRCK01] Jong P. Yoon, Vijay Raghavan, Venu Chakilam, and Larry Kerschberg. BitCube: A Three-Dimensional Bitmap Indexing for XML Documents, 2001.
WWW: http://citeseer.nj.nec.com/yoon01bitcube.html
Local: http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/xmlindex/material/yoon01bitcube.pdf