Content-Aware DataGuides for Indexing Large Collections of XML Documents

Felix Weigel[1] Holger Meuss[2] François Bry[1] Klaus U. Schulz[3]

[1]Institute for Computer Science
University of Munich (LMU), Germany
{weigel,bry}@informatik.uni-muenchen.de

[2]European Southern Observatory
Headquarter Garching, Germany
hmeuss@eso.org

[3]Centre for Inform. & Language Processing
University of Munich (LMU), Germany
schulz@cis.uni-muenchen.de

Abstract

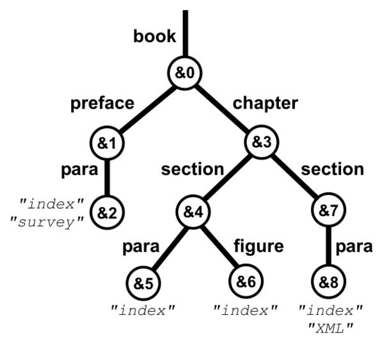
XML is well-suited for modelling structured data with textual content. However, most indexing approaches perform structure and content matching independently, combining the retrieved path and keyword occurrences in a third step. This paper shows that retrieval in XML documents can be accelerated significantly by processing text and structure simultaneously during all retrieval phases. To this end, the Content-Aware DataGuide (CADG) *enhances the well-known DataGuide with (1) simultaneous keyword and path matching and (2) a precomputed content/structure join. Extensive experiments prove the CADG to be 50-90% faster than the DataGuide for various sorts of query and document, including difficult cases such as poorly structured queries and recursive document paths. A new query classification scheme identifies precise query characteristics with a predominant influence on the performance of the individual indices. The experiments show that the CADG is applicable to many real-world applications, in particular large collections of heterogeneously structured XML documents.*

1. Introduction

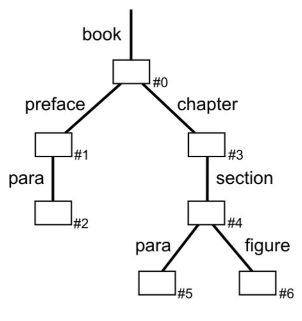
The *eXtensible Markup Language (XML)* [3] has established itself as the representation format of choice for semi-structured data [1]. Many modern applications produce and process large amounts of XML data, which must be queried with both structural and textual selection criteria. A typical example are digital libraries and archives, where either human users or management and mining tools search for papers with, say, a title mentioning *"XML"* and a section about *"SGML"* in the related work part. Obviously, both the query keywords (*"XML"*, *"SGML"*) and the given structural hints (title, related work) are needed to retrieve relevant papers: searching for *"XML"* and *"SGML"* alone would yield many unwanted papers dealing mainly with SGML, whereas a query for all publications with a title and a related work section selects virtually all papers in the library. Other applications include retrieval in structured web pages and manuals; collections of tagged textual content, such as linguistic or juridical documents; compilations of annotated scientific data, e.g. monitoring output in computer science or astronomy; e-business applications managing product catalogues; or web service descriptions. Novel Semantic Web applications will make the need for combined content and structure retrieval of XML metadata even more urgent.

All these applications have in common that they (1) query XML (i.e. semi-structured) data which (2) contains large portions of text and (3) requires persistent index structures for efficient processing. In addition, many of the aforementioned cases deal with rather static data, where updates are local and unfrequent. Despite the wealth of indexing techniques known from both Information Retrieval (IR) and database (DB) research, there are few approaches designed for this characteristic class of data. While IR indexing approaches tend to neglect the structure of documents, many approaches developed by the DB community disregard the textual content of XML data. The *DataGuide* [6, 9] as the ground-breaking approach to indexing XML is a pure structure index in its original form, and is used with a separate inverted keyword index in [9]. As a consequence, it suffers from an increased retrieval overhead even for selective queries, similar to inverted lists in multi-attribute search. More recent adaptations of the DataGuide give up the strict separation of content and structure, but still process both sequentially. The *Content-Aware DataGuide (CADG)*, introduced in this paper as an extension of the DataGuide, uses both structural and textual selection criteria simultaneously during all retrieval phases. Compared to the original DataGuide, this reduces the evaluation time by more than 50% in most cases, and up to 90% under favourable, yet highly realistic conditions.

This paper is organized as follows. The next section describes the DataGuide as the basis of the CADG, along with a simple query formalism to be used throughout the text. The following two sections introduce the Content-Aware DataGuide on different levels of abstraction: first section 3



(a) document tree



(b) index tree

#0	#1	#2	#3	#4	#5	#6
&0	&1	&2	&3	&4;&7	&5;&8	&6

(c) annotation table

"index"	"survey"	"XML"
&2;&5;&6;&8	&2	&8

(d) content table

Figure 1. Data structures of the original DataGuide

explains two abstract concepts of *content awareness*, one of which (the *structure-centric* approach) is shown to be superior. Section 4 elaborates on two concrete realizations of structure-centric content awareness, the *Identity CADG* and the *Signature CADG*. The following section is dedicated to the exhaustive experiments performed to compare both CADGs with the original DataGuide. The paper concludes with remarks on related work and future research.

2. Indexing XML with the original DataGuide

A well-known and influential approach to indexing semistructured data is the *DataGuide* [6, 9]. A DataGuide is essentially a compact representation of the document tree, in which all distinct label paths appear exactly once, as shown in figure 1. The tree on the left *(a)* depicts a small document collection. The corresponding DataGuide is shown in the middle *(b)*. A comparison of both trees reveals that multiple instances of the same document label path, like /book/chapter/section in *(a)*, collapse to form a single index label path in *(b)*. Therefore the resulting *index tree*, which serves as a path index, is usually much smaller than the document tree (although theoretically its size is linear in that of the document tree). Hence it is supposed to be held in main memory even for large document collections.

Without references to individual document nodes, however, the index tree only allows to find out about the existence of a given label path, but not its position in the collection. To this end, every index node is *annotated* with the IDs of those document nodes it represents (i.e. those reached by the same label path as the index node). For instance, the index node #4 in figure 1 *(b)* with the label path /book/chapter/section points to the document nodes &4 and &7, as they are accessible via this very path in the document tree *(a)*. The annotations of all index nodes are stored on disk in an *annotation table (c)*. Formally, the table represents a mapping $dg_a : i \mapsto D_i$ where i is an index node and D_i the set of document nodes reached by i's label path. Together the index tree and the annotation table encode nearly all structural information which is present in the document collection. Only parent/child relations between document nodes cannot be reconstructed from the DataGuide, due to the merging of multiple document paths in a single index path. For instance, from figure 1 *(b)* and *(c)* one cannot tell whether in *(a)* &8 is a child of &4 or &7, which are both referenced by the parent of &8's index node, #5.

A third data structure indexes the textual document content. The DataGuide described above, as a pure path index, ignores textual content altogether. *Keywords* occurring in the document collection are indexed in a separate *content table*, which is stored on disk like the annotation table. The content table is an inverted list mapping a keyword to the set of document nodes where it occurs, as shown in figure 1 *(d)*. More formally, it implements a mapping $dg_c : k \mapsto D_k$ where k is a keyword and D_k the set of document nodes which contain an occurrence of k. Note that although figure 1 shows both the content table and the annotation table in non-first normal form (NF^2), this is not mandatory.

Our data model for tree queries, which covers the core XPath constructs, distinguishes between *structural* and *textual* query nodes. While structural query nodes are matched by document nodes with a suitable label path, textual query nodes correspond to certain keywords occurring in such a document node. As shown in figure 2 for the query tree `/book[.//*["XML"] and ./preface/para["index"]]`, each query path consists of structural nodes, linked by labelled edges, and possibly a single textual leaf node containing a non-empty set of query keywords, which are logically con- or disjoined. Edges to structural query nodes may be either *rigid* (solid line), which means that the two linked query nodes only match a parent/child pair of document nodes, or *soft* (dashed line), corresponding to XPath's `descendant` axis. Similarly, a textual query node is either reached by a rigid edge, indicating that its keywords must occur in the text contained in any document node matching the parent query node, or a soft edge, in which case the keywords may also be nested deeper in the subtree of the document node. For a formal language definition, see [10].

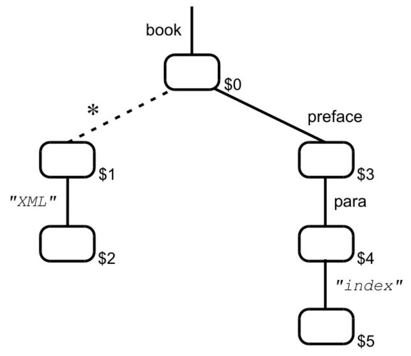


Figure 2. Query tree

Query processing with the DataGuide is divided into the following four *retrieval phases*.

1. *Path matching:* the query paths are matched separately against the index tree.
2. *Occurrence fetching:* annotations of the index nodes found in phase 1 are fetched from the annotation table; query keywords are looked up in the content table.
3. *Content/structure join:* for each query path, the sets of annotations and keyword occurrences are intersected.
4. *Path join:* the results of all query paths are joined to form hits matching the entire query tree.

While phases 1 and 3 are accomplished in main memory, phase 2 involves two disk accesses. Phase 4 may, but need not, require further I/O operations.

As an example, consider the pseudo-XPath query $/\texttt{book}// * [\texttt{"XML"}]$. It selects all document nodes below a tree root labelled book which contain an occurrence of the keyword *"XML"* (note that the shorthand $[\texttt{"XML"}]$ is not part of the XPath language). In phase 1, the query path $/\texttt{book}//*$ is searched in the index tree shown in figure 1 *(b)*. All nodes in the index tree except the root #0 qualify as structural hits. This illustrates that unselective query paths featuring the $//$ and $*$ constructs may cause multiple index nodes to be retrieved during path matching. In phase 2, the annotations (i.e. IDs of matching document nodes) of all index nodes from the previous retrieval phase are fetched from the annotation table. In our example, looking up the index node IDs #1 to #6 in the table yields the six annotation sets $\{\&1\}$, $\{\&2\}$, $\{\&3\}$, $\{\&4;\&7\}$, $\{\&5;\&8\}$, and $\{\&6\}$, respectively. Besides, a look-up of the query keyword *"XML"* in the content table identifies $\{\&8\}$ as the singleton set of document nodes where this keyword occurs. In phase 3, the content/structure join, each annotation set retrieved during the previous phase is intersected with the occurrence set for *"XML"* to find out which of the document nodes with a matching label path contain an occurrence of the query keyword. Here almost all candidate index nodes are discarded, their respective annotation set and the singleton occurrence set being disjoint. Only #5 references a document node which meets both the structural and textual selection criteria of the given query path. The document node in the intersection $\{\&5;\&8\} \cap \{\&8\} = \{\&8\}$ is returned as the only hit. Since the example is a path query, we are finished. If there were more paths in the query tree, &8's ancestors would need to be retrieved, too, in order to join the hits of all query paths (retrieval phase 4).

In the above example a lot of false positives (all index nodes but #0 and #5) are retrieved during path matching and kept in the fetching phase, to be finally ruled out in retrieval phase 3. Not only does this make the path matching step unnecessarily complex; it also causes needless disk accesses in phase 2. The reason why the false positives are not discarded during the first two retrieval phases is that structural and textual selection criteria are handled separately. While both are satisfied when considered in isolation, the join of content and structure in phase 3 reveals the mismatch. Note that a reverse matching order – first keyword fetching, followed by navigation and annotation fetching – is no good, unless keyword fetching fails altogether (in which case navigation is useless, and the query can be rejected as unsatisfiable right away). Moreover, it results in similar deficiencies for queries with selective paths, but unselective keywords. In other words, the DataGuide faces an inherent defect, keeping structural and textual selection criteria apart during the first two retrieval phases. Therefore we propose a *Content-Aware DataGuide* which combines structure and content matching from the very beginning of the retrieval process. This accelerates the evaluation process especially when querying selective keywords and unselective paths.[1]

3. Two approaches towards a Content-Aware DataGuide (CADG)

As stated in the previous section, the objective of the Content-Aware DataGuide (CADG) is to integrate content matching with both path matching and annotation fetching (retrieval phases 1 and 2, respectively), thus saving an explicit content/structure join (phase 3). In short, one can say that the CADG enhances the original DataGuide with a materialized content/structure join and a keyword-aware path

[1] The integrated content information of a CADG can also be used to facilitate schema browsing with the index tree, as proposed for the DataGuide in [6]. Creating *keyword-specific views of the document schema* in this way is not explored here for reasons of brevity.

matching procedure. More specifically, we propose two different techniques (an exact and a heuristic one) to prune index paths which are irrelevant to a given query keyword. This *content-aware navigation* not only reduces the number of paths to be visited during phase 1, but also excludes false positives from the expensive annotation fetching in phase 2. Besides, the two table look-ups in phase 2 (annotation and keyword occurrence fetching) are integrated within a single *content-aware annotation fetching* step, which again reduces the number of disk accesses by up to 50%. The idea is to precompute the content/structure join (phase 3) at indexing time such that document nodes can be retrieved simultaneously by their label path and the keywords they contain. This also avoids the intersection of possibly large sets of document node IDs at query time.

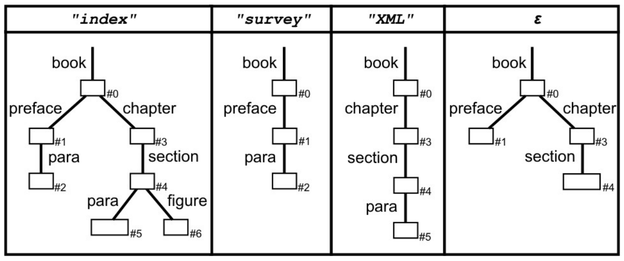
We examine two symmetric approaches to meeting the above objectives. The *content-centric approach* (see section 3.1), being simple but inefficient, only serves as starting point for the more sophisticated *structure-centric approach*, which is pursued in the sequel. Section 3.2 presents it from an abstract point of view. Two concrete realizations, as mentioned above, are covered in section 4.

3.1. Naive content-centric approach

One way to restrict path matching to *relevant* index nodes, i.e. those referencing one or more document nodes in which a given query keyword occurs, is to create multiple keyword-specific index subtrees. Figure 3 depicts four such index subtrees, each of which indexes only those paths in the document tree from figure 1 *(a)* where a specific keyword occurs. Document nodes without textual content are associated with the empty word, ε. For instance, the *"XML"* index subtree in the third column ignores all but a single document path, /book/chapter/section/para, which is the only one leading to an occurrence of the keyword *"XML"* in figure 1 *(a)*. Analogously, the annotation fetching step becomes content-aware when partitioning the annotation table into keyword-specific subtables, which is equivalent to precomputing the content/structure join from retrieval phase 3: a right outer join[2] $dg_c \ltimes dg_a$ of the DataGuide's content and annotation tables in first normal form (1NF) produces a *content/annotation table*, shown in NF^2 in figure 3 *(b)*, which replaces the original tables. Formally, it represents a mapping $cadg_{cc} : (k, i) \mapsto D_{k,i}$ where k is a keyword, i is an index node ID, and $D_{k,i}$ is the set of document nodes where k occurs and which are referenced by i.

In terms of classic database systems, each index subtree is built over a keyword-specific *view* of the data. Consequently, the appropriate index subtree can only be chosen at query time, after the desired keywords have been specified. When processing the sample query from section 2, e.g., the

[2] with dg_c's document node column eliminated



(a) content-centric index trees

"index"			"survey"	"XML"	ε			
#2	#5	#6	#2	#5	#0	#1	#3	#4
&2	&5;&8	&6	&2	&8	&0	&1	&3	&4;&7

(b) content-centric content/annotation table

Figure 3. Content-centric CADG approach

"XML" subtree is selected and used during path matching. This narrows down the search space to the path reaching index node #5, excluding false positives right from the start. Occurrence and annotation fetching is accomplished by a single look-up in the content/annotation table, where the entry for #5 and *"XML"* is selected and returned as query result. Note that no explicit content/structure join is required at query time.

Obviously the content/annotation table may easily take up more space on disk than the original DataGuide's content and annotation tables together. In figure 3 *(b)* there are e.g. multiple columns referring to the index nodes #2 or #5, whereas column headers in figure 1 are unique. This redundancy, which is due to the Cartesian product underlying the join of the content and annotation tables, increases with the number of *path-unselective* keywords, i.e. those occuring under a variety of different label paths. Although unselective keywords, being of restricted use as selection criteria, are sometimes disregarded in indexing, it is true that the faster query processing provided by content awareness comes at the price of increased storage consumption (see section 5.2 for experimental results). Yet this trade-off is common to most indexing techniques. A much more important drawback of the content-centric approach is that not only the content/annotation table but also the index subtrees reside in secondary storage. Since the complete set of index subtrees (which has the same cardinality as the set of indexed keywords) cannot be held in main memory, selecting the right index subtree for a query keyword thus entails an additional disk access for loading the appropriate index subtree. When processing queries with more than one keyword, multiple index subtrees need to be fetched from disk, which is clearly prohibitive at query time.

3.2. Structure-centric approach

The naive approach presented in the previous subsection is *content-centric* in the sense that the indexed keywords determine the structure of both the index tree and the content/annotation table. A second, more viable approach to content awareness preserves the original DataGuide's index tree in its integrity, grouping the indexed keyword occurrences by their label paths. This *structure-centric* approach allows path matching to be performed without loading the index tree from disk. It resides in main memory like the index tree of the DataGuide. However, each node of the CADG carries a small amount of additional content information necessary to prune irrelevant index paths during phase 1. Dedicated data structures to be presented in the next section encode (1) whether an index node i references any document node where a given keyword k occurs, and (2) whether any of i's descendants (including i itself) does. In the remainder of this paper, we refer to the former relation between an index node i and a keyword k as *containment* (i contains k), and to the latter as *government* (i governs k). For instance, in figure 1 *(a)* and *(b)*, the index node #4 does not contain any keyword, although it governs both *"index"* and *"XML"*. Note that by definition, containment implies government, but not vice versa. During retrieval phase 1, the government relation is examined for any index node reached during path matching, in what we call the *government test*. A *containment test* takes place in retrieval phase 2 to avoid needless disk accesses. Both of these *relevance tests* are integrated with the original DataGuide's retrieval procedure (see section 2) to enable content-aware navigation and annotation fetching, as follows. During path matching, whenever an index node i is being matched against a structural query node q_s, the procedure $governs(i, q_s)$ is performed. It succeeds if and only if for each textual query node q_t below q_s containing a keyword conjunction $\bigwedge_{u=0}^{p} k_u$, condition (2) above is true for i and all keywords k_u (or at least one keyword in case of a disjunction $\bigvee_{u=0}^{p} k_u$). In this case path matching continues with the descendants of i; otherwise i is discarded along with its entire subtree. During retrieval phase 2, before fetching the annotations of any index node i matching the parent node of a textual query node q_t, the procedure $contains(i, q_t)$ is called to verify condition (1) for all of q_t's keywords (in case of a keyword conjunction, or at least one if they are disjoined). Upon success, i's annotations are fetched from disk; otherwise the node i is ignored.

The realization of the $governs()$ and $contains()$ procedures depends on how content information is represented in the index node given as first parameter (see section 4). In any case, however, the query node handed over as second parameter must bear content information, too, namely about the query keywords attached to its outgoing query paths. To avoid repeated exhaustive keyword searches in the query tree, every query node accomodates keyword information in a compact representation suitable for fast content matching during retrieval phases 1 and 2, similar to an index node. This is accomplished in a preliminary *query preprocessing step*, taking place in an new retrieval phase 0 (see below).

#0	#1	#2		#3	#4	#5		#6
ε	ε	"index"	"survey"	ε	ε	"index"	"XML"	"index"
&0	&1	&2	&2	&3	&4;&7	&5;&8	&8	&6

Figure 4. Structure-centric CADG approach

The referenced document nodes to be fetched in phase 2 are stored on disk in a combined content/annotation table, as shown in figure 4. It is almost identical to the content-centric one in figure 3 *(b)*, except that the corresponding mapping $cadg_{sc} : (i, k) \mapsto D_{k,i}$ takes its two arguments in reverse order, thus reflecting the structure-centric character of the approach. In fact, the NF^2 table in figure 4 can be considered as consisting of seven index-node specific content tables (labelled #0 to #6), each built over a label-path specific view of the document collection. Note that this conceptual difference between the content-centric and structure-centric content/annotation tables vanishes on the physical level, both tables being identical in $1\mathrm{NF}$. Index node ID and keyword together make up the primary key to support combined content/structure or pure structure queries (during phase 2) as well as pure content queries (during phase 0). Accordingly, both tables are equal in size.

4. Two realizations of the structure-centric approach: Identity and Signature CADG

The concept of a structure-centric CADG, as discussed in the previous section, does not specify data structures and algorithms for integrating content information with the index and query trees. In this section we propose two alternative content representations, along with a suitable query preprocessing as well as containment and government tests (formal proofs of correctness are omitted). The first approach, which exploits index node IDs (see section 4.1), is guaranteed to exclude all irrelevant index nodes from path matching and annotation fetching. A second, signature-based realization of the structure-centric CADG (see section 4.2) represents keywords in an approximate manner, possibly mistaking some irrelevant index nodes as relevant during query processing. A final verification, performed simultaneously with the annotation fetching step, eventually rules out these false positives. Hence both variants of the CADG produce exact results, regardless of whether their content awareness is based on heuristic techniques or not.

4.1. Identity CADG

The *Identity CADG* relies on index node IDs to enable content-aware path matching and annotation fetching, as introduced in section 3. The idea is to prepare a list of *relevant index nodes* for each path in the query tree, comprising the IDs of all index nodes which contain the query keywords of this path. (Refer to section 3.2 for a definition of the containment and government relations between index nodes and keywords.) Assembled in a query preprocessing step (retrieval phase 0) to be described next, these lists are attached to the query tree (see figure 5). In the index tree, by contrast, content information is only represented implicitly by means of index nodes IDs. Unlike the Signature CADG presented below, the Identity CADG has no dedicated data structures for storing keyword information in the index tree. During retrieval phase 1, only ancestors of relevant index nodes are considered, while other nodes are pruned off. Similarly, only annotations of relevant index nodes are fetched during phase 2. Ancestorship among index nodes is tested by navigating upwards in the index tree (which requires a single backlink per node) or else computationally, by means of numbering schemes like e.g. interval encoding [8, 17]. Alternatively, ancestor IDs could be determined during phase 0 and stored in the query tree.

Query preprocessing. During retrieval phase 0, each node q of a given query tree is assigned a set I_q of sets of relevant index node IDs, as illustrated in figure 5. This second-order set is used for the relevance tests as described below. Let us consider first textual, then structural query nodes. Any textual query node q_t with a single keyword k_0 is associated with the set I_{k_0} of IDs of index nodes containing k_0, i.e. $I_{q_t} := \{I_{k_0}\}$. I_{k_0} is the set of all index nodes associated with k_0 in the content/annotation table. If the query node represents a conjunction $\bigwedge_{u=0}^{p} k_u$ of multiple keywords, their respective sets I_{k_u} are intersected, $I_{q_t} := \{\bigcap_{u=0}^{p} I_{k_u}\}$, because the conjoined keywords must all occur in the same document node, and hence be referenced by the same index node for the query to match. Analogously, a query node representing a disjunction $\bigvee_{u=0}^{p} k_u$ of keywords is associated with the union $I_{q_t} := \{\bigcup_{u=0}^{p} I_{k_u}\}$ of sets of relevant index node IDs. If $I_{q_t} = \{\emptyset\}$ the query is immediately rejected as unsatisfiable (without entering retrieval phase 1), because no index node references any document node where all keywords of the current query path occur.

Each structural query node q_s inherits sets of relevant index nodes (contained in a second-order set I_{q_v}) from each of its children q_v $(0 \leq v \leq m)$, i.e. $I_{q_s} := \bigcup_{v=0}^{m} I_{q_v}$. Thus the textual context of a whole query subtree is taken into account while matching any single query path. It is important to keep the member sets of all children's sets I_{q_v} separate rather than intersect them like in the case of a keyword conjunction discussed above (hence the second-order construct for textual query nodes in the previous paragraph): after all the children q_v are not required to be all matched by the same document node, which simultaneously contains occurrences of all their query keywords. Consequently, the government test for an index node i matching q_s is supposed to succeed as soon as there exists for each child query node q_v one descendant of i containing the keywords below q_v, without demanding that it be the same for all q_v. Therefore the sets contained in the I_{q_v} sets are not merged. In case q_s has no children, $I_{q_s} := \emptyset$ is used as a "don't care" symbol, ensuring that the government test for such query nodes always succeeds (see below). Note that all children q_v are treated alike, regardless of whether they are structural or textual query nodes. As a consequence, this preprocessing procedure also copes with mixed-content query trees.

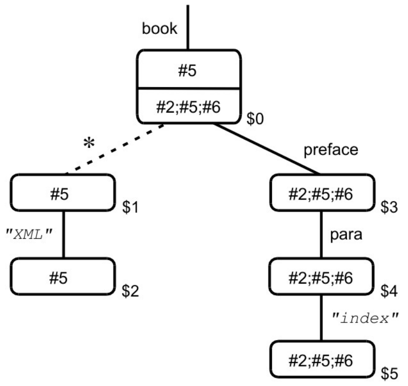


Figure 5. Identity CADG: adapted query tree

Figure 5 illustrates the preprocessed tree query /book[.//*["XML"] and ./preface/para["index"]]. To each query node q the member sets of I_q have been attached (one per row). For instance, the root of the query tree, $0, is associated with the set $I_{\$0} = \{\{\#5\}; \{\#2; \#5; \#6\}\}$. All sets of index node IDs have been computed using the content/annotation table shown in figure 4.

Relevance tests. As described in section 3.2, the government test $governs(i, q_s)$ is performed whenever an index node i is matched against a structural query node q_s during retrieval phase 1. In each set $I_{q_v} \in I_{q_s}$, a descendant of i is searched (e.g. using binary search, if I_{q_v} is ordered), with i counting as its own descendant, too. The test $governs(i, q_s)$ succeeds if and only if there is at least one (reflexive) descendant of i in each set I_{q_v}. Note that as a special case, the condition is satisfied for $I_{q_s} = \emptyset$.

The containment test $contains(i, q_t)$ is performed when processing a textual query node q_t during retrieval phase 2. It takes place for every index node i matching q_t's parent q_s, provided its government test succeeded. This ensures that i or any of its descendants references a document node relevant to q_t's keywords. To determine whether annotation fetching for i is justified, the index node is searched in the only member set I_k of the singleton set I_{q_t}, which contains the index nodes relevant for q_t's keywords. Again, binary search may be applied if I_k is ordered. The test $contains(i, q_t)$ succeeds if and only if $i \in I_k$. Obviously $contains()$ is similar to $governs()$ except that it is based on an identity relation between the index node being examined and the members of the sets in I_q, rather than an ancestor/descendant relation. Hence the containment test is stricter than the government test, as claimed in section 3.2.

As an example of content-aware retrieval with the Identity CADG, consider the query tree in figure 5, whose left branch has already been discussed in section 2. Since the Identity CADG's index tree is identical to the DataGuide's, we also refer to figure 1 *(b)* in the following. The corresponding content/annotation table is given in figure 4. Beginning with the label `book`, path matching identifies the index node #0 as a structural match for the query node \$0. The relevance test $governs(\#0, \$0)$ succeeds because in both sets associated with \$0, $\{\#5\}$ and $\{\#2; \#5; \#6\}$, there is a descendant of #0 (namely #5). The two paths leaving \$0 are processed one after the other. \$0's left child \$1 is reached by a soft edge without label constraint. Hence all index nodes but #0 are structural matches for \$1, as observed in section 2. First, the root's left child #1 undergoes a government test for \$1. Since none of its descendants is in \$1's list, $governs(\#1, \$1)$ fails right away, excluding the whole left branch of the index tree from further processing. #2 never enters path matching, let alone annotation fetching. As the first node in the right index path, #3 passes the government test for \$1 (being an ancestor of #5), but fails in the containment test for \$2 since $\#3 \notin \{\#5\}$. Its child #4 satisfies the government test for the same reason as #3. Analogously, it fails in $contains(\#4, \$2)$. By contrast, #5 passes both tests, being itself a member of both \$1's and \$2's ID list, $\{\#5\}$. Consequently, #5's occurrences of the keyword *"XML"* are fetched from the content/annotation table, retrieving &8 as \$1's only hit. The last matching index node is ruled out immediately by the government test $governs(\#6, \$1)$, which reveals that #6 is not an ancestor of the only relevant index node, #5. Processing the second query path is similar, and omitted here for brevity.

The sample query above shows how content-awareness can save both main-memory and disk operations. Compared to the DataGuide (see section 2), two subtrees (rooted at #1 and #6, respectively) are pruned during retrieval phase 1, and only one disk access is performed instead of seven during phase 2. Another I/O operation is needed in phase 0 for looking up relevant index nodes. Note that this saves the whole evaluation for queries with non-existent keywords. The final results are identical for both index structures.

4.2. Signature CADG

The *Signature CADG* differs from the Identity CADG in several respects. Most importantly, keyword information for content-aware path matching and annotation fetching is represented only approximately. The resulting heuristic relevance tests are not guaranteed to rule out all index nodes which are irrelevant w.r.t. a given query keyword, some of them being recognized as false hits only when looked up in the content/annotation table. (Nevertheless the retrieval is exact, as explained below.) Unlike the Identity CADG, the Signature CADG relies on additional data structures (*signatures*) in the index tree, created at indexing time, to represent the keyword occurrences referenced by an index node. The query tree is prepared in a similar manner at query time. As a third difference, a precomputed cumulative content representation for entire index subtrees substitutes for the ancestor/descendant check in the government test.

Signatures. A common IR technique for the concise representation and fast processing of content information are *signatures*, i.e. bit strings of a fixed length. Every keyword to be indexed or queried is assigned a (preferrably unique and sparse) signature. Note that this does not require all keywords to be known in advance, nor to be explicitly assigned a signature before indexing. Instead a suitable signature may be created from the keyword's character sequence, e.g. using hash functions (which may produce a negligible quantity of ambiguous signatures).

Sets of keywords, e.g. in a document or query node, can be represented collectively by a single signature, resulting from the bitwise disjunction ($\sqcup$) of the individual keyword signatures. As shown in figure 6, this may cause ambiguities due to overlapping bit patterns. In fact, the heuristic nature of the Signature CADG's content awareness results from this concise, but lossy content representation. Other operations on signatures s_0, s_1 include the bitwise conjunction ($s_0 \sqcap s_1$), bitwise inversion ($\neg s_0$), and bitwise implication which we define as $s_0 \sqsubset s_1 := (\neg s_0) \sqcup s_1$.

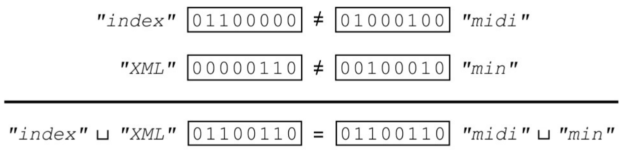

"index" 01100000 ≠ 01000100 "midi"
"XML" 00000110 ≠ 00100010 "min"
"index" ⊔ "XML" 01100110 = 01100110 "midi" ⊔ "min"

Figure 6. Ambiguous keyword signatures

Index tree. The Signature CADG's index tree closely resembles the one of the original DataGuide (see figure 1). The only difference is that each index node i has two signatures attached to it. A *containment signature* is created from the bitwise disjunction of the signatures of all keywords occurring in i's referenced document nodes. (If there are no such keywords, i's containment signature is set to `00000000`.) For content-aware navigation, a *government signature* encodes the keywords referenced by i or any of its descendants in the index tree. Inner index nodes inherit their children's government signatures and combine it with their own containment signature, again by bitwise disjunction. For leaf nodes, both signatures are identical. Figure 7 depicts the index tree of a Signature CADG built over the document collection from figure 1 *(a)*.

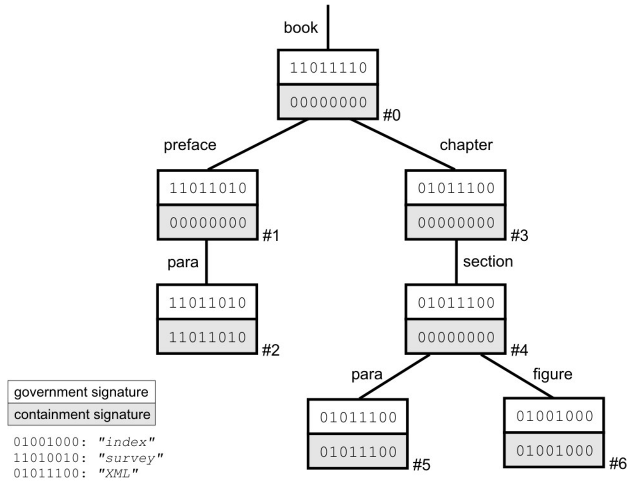


Figure 7. Signature CADG: index tree

Query preprocessing. Similar to the index tree, the query tree is prepared for content-aware navigation and occurrence fetching with the Signature CADG. Every textual query node q_t has a single signature s_{q_t} created from the keyword signatures s_{k_u} of all keywords k_u ($0 \leq u \leq p$) attached to this node (which are either stored in a *signature table* or created on the fly). If there is only one such keyword, say k_0, then $s_{q_t} := s_{k_0}$. In the case of a keyword conjunction $\bigwedge_{u=0}^{p} k_u$, s_{q_t} is set to be the bitwise disjunction of the keyword signatures, $s_{q_t} := \bigsqcup_{u=0}^{p} s_{k_u}$. The reason for this somewhat counterintuitive definition will become apparent when examining the containment test. Informally, disjoining the signatures allows each keyword to "leave its footprint" in the query node's signature, as required for a keyword conjunction. Analogously, $s_{q_t} := \bigsqcap_{u=0}^{p} s_{k_u}$ for a keyword disjunction $\bigvee_{u=0}^{p} k_u$ in q_t. A structural query node's signature s_{q_s} indicates which keywords are contained in the textual query nodes below q_s. To this end, the signatures s_{q_v} of its children q_v ($0 \leq v \leq m$) are disjoined, $s_{q_s} := \bigsqcup_{v=0}^{m} s_{q_v}$. As observed for the Identity CADG, this upward propagation of keyword information includes the textual context of a whole query subtree when matching any single query path. The signature of a childless structural query node is set to `00000000` which guarantees that any index node's government test will succeed, as one would expect for a query node without textual constraints.

Figure 8 illustrates the tree query from figure 5, this time preprocessed for the Signature CADG. The keyword signatures are the same as in figure 7. They are either fetched from a *signature table*, where they have been stored at indexing time, or created on the fly. Although the former solution requires some extra disk space and an additional I/O operation at query time, it proved superior to dynamic signature creation in our experiments (see section 5).

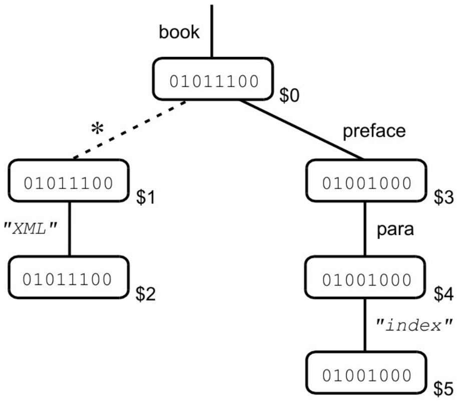


Figure 8. Signature CADG: adapted query tree

Relevance tests. The government test $governs(i, q_s)$ for an index node i and a structural query node q_s simply consists of the bitwise implication of q_s's signature and i's government signature, $s_{q_s} \sqsubset s_g$. This means that those bits set in s_{q_s} because of the query keywords below q_s must also be set in s_g, which is definitely the case when each such query keyword occurs in some document node referenced by i or its descendants. However, the converse is not always true: $s_{q_s} \sqsubset s_g$ may also hold when i does not govern all the keywords in q_s's subtree. Recall from figure 6 that the disjunction of different sets of keyword signatures can produce identical results. Likewise, other keywords than the ones responsible for s_{q_s} can make s_g look as if i were relevant w.r.t. q_s, although it is not. In this case, path matching continues in the subtree rooted at i, ignoring the fact that occurrence fetching for any of its nodes is doomed to fail.

Analogously to the government test just described, the containment test $contains(i, q_t)$ for an index node i and a textual query node q_t is just the bitwise implication of q_t's signature and i's containment signature, $s_{q_t} \sqsubset s_c$. It succeeds when q_t's keywords occur in document nodes referenced by i itself (regardless of its descendants), where they cause the same bits to be set in s_c as in s_{q_t}. But as with the government test, i might also pass the containment test without actually containing all keywords in s_{q_t}, in which case occurrence fetching for this index node (including a database access) is performed in vain.

For instance, reconsider the query from figure 8, /book[.//*["XML"] and ./preface/para["index"]], and the index tree in figure 7 whose corresponding content/annotation table looks like the one in figure 4. The query tree's root label book leads to index node #0, whose government signature is $\boxed{\texttt{11011110}}$. Since the test $governs(\#0, \$0) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{11011110}}$ succeeds, path matching continues with the query node \$1. The $//*$ step is matched by all index nodes except the index root, as observed in section 4.1. #0's left child #1 is immediately discarded in the government test for \$1, $governs(\#1, \$1) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{11011010}}$, since the antepenultimate bit is set in \$1's signature, but not in #1's. Therefore the left branch of the index tree is pruned, and path matching continues with #3. It passes $governs(\#3, \$1) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01011100}}$, but not $contains(\#3, \$2) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{00000000}}$. The same is true for #3's only child #4. Yet the remaining index nodes behave differently: while #5 passes both $governs(\#5, \$1)$ (same as $governs(\#3, \$1)$) and $contains(\#5, \$2) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01011100}}$, contributing the document node &8 to the result, #6 fails already in the first test, $\boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01001000}}$ (the fourth and sixth bits are missing in its government signature). Accordingly, #6 is excluded from further navigation (which cannot take place anyway, #6 being a leaf node) and annotation fetching, thus saving a look-up in the content/annotation table. The second query path is processed similarly.

In this example, the number of disk accesses compared to the DataGuide is reduced from seven to two (including query preprocessing), like with the Identity CADG above. Moreover, signatures are more efficient data structures than node ID sets (in terms of both storage and processing time), and make the relevance checks and preprocessing easier to implement. Note, however, that if another keyword with a suitable signature occurred in the document node referenced by #6, e.g. the keyword *"query"* with the signature $\boxed{\texttt{00011100}}$, then #6 would be mistaken to be relevant for \$5's query keyword *"XML"*. The reason is that both #6's government and containment signatures would then be the bitwise disjunction of the two signatures representing *"index"* and *"query"*, $\boxed{\texttt{01001000}} \sqcup \boxed{\texttt{00011100}} = \boxed{\texttt{01011100}}$, which equals the keyword signature for *"XML"*. Hence both $governs(\#6, \$1)$ and $contains(\#6, \$2)$ would succeed. Only after a content/annotation table look-up would #6 turn out to be a false hit. This illustrates how the Signature CADG trades off pruning precision against navigation efficiency.

5. Experimental evaluation

This section gives an overview of the extensive experiments that were carried out in order to evaluate the Content-Aware DataGuide. Besides content-awareness, various optimizations have been tested with both realizations of the CADG as well as with the original DataGuide. A detailed report on the evaluation can be found in [17].

5.1. Experimental set-up

The tests have been performed on three document collections with different characteristics: *Cities* is a small collection (16,000 nodes, 1.3 MB) describing German cities. It is a homogeneous collection, comprising 19,000 distinct keywords in 253 different label paths (with a maximal length of 7) which are not recursive (i.e., no label appears twice on a path). The second collection, *XMark*, is a medium-sized (417,000 nodes, 30 MB) synthetically generated collection [18] with 515 different label paths and 84,000 different keywords. This collection is slightly more heterogeneous than the *Cities* collection and contains some recursive paths. The third collection (*NP*, 510 MB), containing syntactically analyzed German noun phrases [13], was the most challenging collection, not only due to its size: *NP* is a strongly recursive and heterogeneous collection (2,349 different label paths of maximal depth 40). In total, 130,000 different keywords appear in 458,000 different nodes.

We tested four basic index structures, namely the original DataGuide, the Identity CADG, the Signature CADG (with 64-bit signatures and 3 bit set in each keyword signature), and a variation of the latter, each equipped with all possible combinations of four optimizations. Thus a total of 64 different index configurations were compared to each other. For lack of space, we only report on the first three index structures without optimizations here. Both hand-crafted and synthetic path query sets were evaluated against the different document collections, resulting in four test suites: unlike *CitiesM* with 166 manually created queries on the *Cities* collection, *CitiesA* (191 queries), *XMarkA* (163 queries), and *NpA* (160 queries) consist of automatically generated queries on the *Cities*, *XMark*, and *NP* collections, respectively. For a systematic analysis of the experimental results, the queries of all test suites were classified according to seven *query characteristics*, which are summarized in table 1. Each characteristic of a given query is encoded by one of seven bits in a *query signature* determining which class the query belongs to. A bit value of 1 indicates a more restrictive nature of the query w.r.t. the given characteristic, whereas 0 means the query is less selective and therefore harder to evaluate. Hand-crafted queries were classified manually, whereas synthetic queries were assigned signatures automatically during the generation process. Two groups of query characteristic turned out to be most interesting for our purposes. First, the bits 2, 3, and 4 concern the navigational effort during evaluation: queries with `--000--` signatures (read "`-`" as a "don't care" symbol), being structurally unselective, cause many index paths to be visited. Second, the bits 1 and 0 characterize keyword selectivity, a common IR notion which we have generalized to structured documents: A keyword is called *node-selective* if there are few document nodes containing that keyword, and *path-selective* if there are few index nodes referencing such document nodes (for details on the collection-specific selectivity thresholds, see [17]). For instance, the query classes `-----10` contain queries whose keywords occur often in the documents, though only under a small number of different label paths. Not all 128 query classes were used in every test suite, e.g. because specific combinations of path- and node-selective keywords hardly ever occurred in the data. This is particularly the case for *XMarkA* where only 60% of all query classes were populated, whereas the query classes in the other test suites are nearly complete.

6	`1------`	*query result*	mismatch
5	`-1-----`	*branching*	few path joins
4	`--1----`	*soft structure*	few soft-edged struct. nodes
3	`---1---`	*label selectivity*	highly selective labels
2	`----1--`	*soft text*	few soft-edged textual nodes
1	`-----1-`	*path selectivity*	highly path-select. keywords
0	`------1`	*node selectivity*	highly node-select. keywords

Table 1. Query classification scheme

The index structures were integrated into the XML retrieval system X^2 [11] that computes a *Complete Answer Aggregate* (CAA) [10] for a query. The advantage of using this data structure in our case is that a CAA is a minimal representation for the set of all answers to a query. Query evaluation time as a performance measure in all experiments includes CAA construction. Since large parts of the query evaluation algorithms and even index algorithms are shared by all basic index structures, the comparison results are not polluted with implementational artefacts.

All tests have been carried out sequentially on the same computer (AMD Athlon XP 1.33 GHz, 512 MB, running SuSE Linux 7.3 with kernel 2.4.16). The PostgreSQL relational database system (with a disabled database cache), version 7.1.3, was used as relational backend for storing the index structures. To compensate for file system cache effects, each query was processed once without taking the results into account. The following iterations of the same query (between three and ten, depending on the test suite) were then averaged.

5.2. Results

Figure 9 shows the performance results for four selected sets of path query classes ([17] provides similar results for tree queries). Each plot covers the queries in all classes with certain characteristics, as indicated by the plot title. For instance, the upper right plot assembles all query classes with mismatch queries (`11-----`, see table 1), i.e. 32 classes altogether. By contrast, the lower left plot narrows down to those queries with few path joins, many soft-edged structural and textual nodes, and unselective labels (`-1000--`, 8 query classes). As labelled on the abscissa, every test suite comprises three boxes, each of which represents the performance of a single index, i.e. Identity CADG (IC, □), Signature CADG (SC, ■), or DataGuide (DG, ▨). The relative performance compared to the DataGuide, which determines the height of each box, was computed as follows. First, the time needed to evaluate any given query with a specific index was averaged over all iterations of that query. Next, the average time was normalized w.r.t. the DataGuide's average value for that query (which produces 100% for the DataGuide, being compared to itself). Then the relative times of all queries in a single query class were averaged. Finally, the average over the relative evaluation times of all selected classes was plotted. The step-wise averaging ensures that query classes of different cardinality are equally weighted. Normalization w.r.t. to the DataGuide takes place before averaging to make fast and slow queries mutually comparable (otherwise long-evaluating queries would predominate the result).

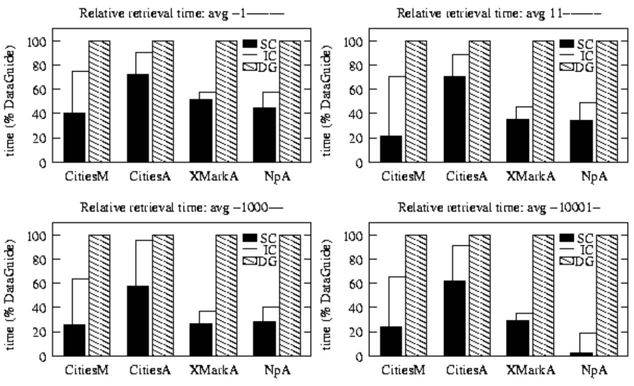


Figure 9. Retrieval time CADG vs. DataGuide

As figure 9 shows, the Signature CADG outperforms the original DataGuide by more than 50% on average for all query classes in the *CitiesM*, *XMarkA*, and *NpA* test suites, and much more for selected classes. In *CitiesA*, the performance gain lies between 30 and 40%, probably due to a higher CAA construction overhead (see section 7).

A comparison of the four plots above reveals that certain additional characteristics of path queries (upper left plot) further increase the relative performance gain of the Signature CADG by a factor 2 to 20 (except for *CitiesA*). Mismatch queries (upper right plot) are usually favourable to the CADG, which detects misplaced or non-existent query keywords early during retrieval. The relative performance gain still grows for queries with fairly unspecific structure (lower left plot), causing more index nodes to be visited and more annotations to be fetched from disk. Here the benefits of content awareness reduce the Signature CADG's evaluation time to well below 30% of the DataGuide's in three out of four test suites. Finally, when considering only those poorly structured queries with path-selective keywords (lower right plot), the Signature CADG's performance on the *NP* collection once again rises dramatically to a gain of 97.5% compared to the original DataGuide. This proves that the Signature CADG is particularly effective for large amounts of data. Moreover, its preference for queries with little structure but selective keywords makes it most suitable for realistic applications, for three reasons: users (1) tend to use selective keywords to reduce the number of hits, (2) often ignore the document schema, unwilling to explore it before querying, and (3) are likely to focus on content rather than structure, being accustomed to the flat keyword search facilities of today's WWW search engines. The same is true for synthetic queries, which often neglect structure to support different document schemata.

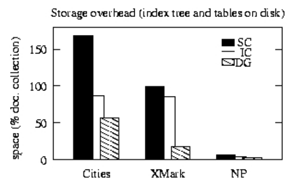


Figure 10. Index size CADG vs. DataGuide

Figure 9 also illustrates that heuristic content-awareness is much more effective than the Identity CADG's exact realization, despite a possible overhead caused by unpruned mismatches. The price to pay is an increased storage overhead due to the signature table. As depicted in figure 10, the Signature CADG grows to 150% of the size of the *Cities* collection, which is three times as big as the DataGuide. However, the storage overhead is reduced considerably for *XMark* and *NP*, again recommending the CADG for large-scale applications. Also note that the storage measurements include so-called *function words* (i.e. extremely unselective keywords without a proper meaning) and inflected forms, which may be excluded from indexing using common IR techniques like stop word lists and stemming. This further reduces the storage overhead. The resulting index, although inexact, is well suited for result ranking like in [14].

6. Related work

In this section we discuss approaches directly related to the CADG, focussing on XML index structures which incorporate the textual content of the documents. Work on index structures for XML in general is surveyed in [16].

An early approach targeted at integrating text retrieval (and relevance ranking) with an index structure for semi-structured data is the *BUS index* [15]. In contrast to our approach, a keyword is mapped to both the containing document nodes and index nodes. The latter are used to filter out document nodes which do not satisfy the path conditions related to the keyword. This corresponds to the DataGuide's content/structure join, although carried out at index node rather than document node level. Note that the CADG uses structural and content information simultaneously in an earlier retrieval phase (content-aware annotation fetching), thus saving the fetching of false positives as well as an explicit content/structure join.

The *Signature File Hierarchy* [4] is based on keyword signatures like the Signature CADG. However, these signatures are not propagated to index nodes. Instead document node signatures need to be fetched from disk during path matching. For realistic data collections, this entails a significant overhead owing to I/O operations.

The *IndexFabric* [5] enriches the DataGuide's index nodes with Tries representing textual content. This is equivalent to the materialized join of the CADG. In contrast to our work, the IndexFabric is equipped with a sophisticated layered storage architecture. Ignoring the notion of content-aware navigation, however, it is closer to the DataGuide.

A very simple approach to content indexing for XML documents is presented in [12]: designed as an extension for inverted-file based index structures mapping keywords to document nodes, the *Context Index* does not use a tree structure to summarize the document schema. Every keyword occurrence bears approximative information about the respective document node's structural context (e.g. its label path) in the form of a structure signature. It serves to discard structural mismatches early in the retrieval process.

Similar in spirit to the aforementioned approach is the work described in [2]. Here keyword occurrences are annotated with information about the structural context in the form of *Materialized Schema Paths*, a datastructure optimized for compact representation of schema and document paths. Flexible access is not only provided to individual keywords, but also to larger portions of content and to node labels, based on collection statistics. From another point of view, the approach resembles the content-centric CADG (see section 3.1), although using index paths (Materialized Schema Paths) instead of DataGuides to represent keyword-specific fragments of the document collection.

7. Conclusion

Results. In this paper, we have introduced the Content-Aware DataGuide (CADG) as an efficient index structure for XML documents. The CADG enhances the original DataGuide with content-aware navigation and annotation fetching. As a means of integrating content and structure matching during all retrieval phases, these optimizations save a join of the retrieved document node sets at query time as well as many needless I/O operations. Two concrete realizations of the CADG have been presented, the Identity CADG featuring exact and the Signature CADG heuristic content-awareness. Based on a novel query classification scheme, experiments prove that (1) the Signature CADG is faster than the Identity CADG, and (2) the Signature CADG outperforms the original DataGuide by more than 50% in nearly all test suites, when averaging all path query classes. For classes containing queries with little structure and selective keywords, which have been shown to be most important in real-world applications, the Signature CADG's retrieval time is only 3-40% of the DataGuide's. The highest performance gain, and lowest storage overhead (5% of the original data), is achieved for a large, heterogeneous document collection of several hundred MB.

Future Work. CAAs as representations of query results usually contain not only the IDs of the retrieved document nodes, but also of some of their ancestors. Currently these ancestor IDs are looked up in the database for each hit, causing many I/O operations especially for unselective queries. We plan to address this performance bottleneck using numbering schemes similar to the one described in [7].

Other possible investigations may concern incremental index updates as well as a generalization to graph databases. In particular, signature propagation in the Signature CADG needs to be reviewed when indexing graph-shaped documents. In special cases we expect the CADG for a graph database to be even smaller than the corresponding original DataGuide. (General complexity results for graph documents and queries can be found in [10].) Future research may also include new techniques for adaptively increasing the level of content-awareness based on query statistics. An on-going project tries to amalgamate DataGuide and CADG techniques with the rather limited indexing support for XML which are provided by commercial relational database systems, like e.g. IBM's XML Extender.

As far as the performance evaluation is concerned, we plan to refine the proposed query classification scheme and to generate queries based on actual documents rather than DTDs. Furthermore it would be interesting to assess the benefits of both content-aware navigation and the materialized content/structure join separately.

Acknowledgements. The authors thank Tim Furche for providing the query generator used in the experiments.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] M. Barg and R. K. Wong. A Fast and Versatile Path Index for Querying Semi-Structured Data. In *Proc. 8th Int. Conf. on Database Systems for Advanced Applications*, 2003.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, 2000.

[4] Y. Chen and K. Aberer. Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases. In *Proc. 10th Int. Conf. on Database and Expert Systems Applications*, 1999.

[5] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proc. 27th Int. Conf. on Very Large Data Bases*, 2001.

[6] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, 1997.

[7] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. *Proc. 1st ACM Int. Conf. on Digital Libraries*, 1996.

[8] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. 27th Int. Conf. on Very Large Data Bases*, 2001.

[9] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[10] H. Meuss, K. Schulz, and F. Bry. Towards Aggregated Answers for Semistructured Data. In *Proc. 8th Int. Conf. on Database Theory*, 2001.

[11] H. Meuss, K. Schulz, and F. Bry. Visual Querying and Exploration of Large Answers in XML Databases with X^2: A Demonstration. In *Proc. 19th Int. Conf. on Database Engineering*, 2003.

[12] H. Meuss and C. Strohmaier. Improving Index Structures for Structured Document Retrieval. In *Proc. 21st Ann. Colloquium on IR Research*, 1999.

[13] J. Oesterle and P. Maier-Meyer. The GNoP (German Noun Phrase) Treebank. In *Proc. 1st Int. Conf. on Language Resources and Evaluation*, 1998.

[14] T. Schlieder and H. Meuss. Querying and Ranking XML Documents. *JASIS Spec. Top. XML/IR 53(6):489-503*, 2002.

[15] D. Shin, H. Jang, and H. Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. In *Proc. 3rd ACM Int. Conf. on Digital Libraries*, 1998.

[16] F. Weigel. A Survey of Indexing Techniques for Semistructured Documents. Technical report, Dept. of Computer Science, University of Munich, Germany, 2002.

[17] F. Weigel. Content-Aware DataGuides for Indexing Semi-Structured Data. Master's thesis, Dept. of Computer Science, University of Munich, Germany, 2003.

[18] XML Benchmark Project. Benchmark suite for XML repositories. Available at `http://monetdb.cwi.nl/xml`.