Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data

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

[1] Institute for Computer Science, University of Munich, Germany
[2] European Southern Observatory, Garching, Germany
[3] Centre for Information and Language Processing, University of Munich, Germany

Abstract. Not only since the advent of XML, many applications call for efficient structured document retrieval, challenging both Information Retrieval (IR) and database (DB) research. Most approaches combining indexing techniques from both fields still separate path and content matching, merging the hits in an expensive join. This paper shows that retrieval is significantly accelerated by processing text and structure simultaneously. The *Content-Aware DataGuide (CADG)* interleaves IR and DB indexing techniques to minimize path matching and suppress joins at query time, also saving needless I/O operations during retrieval. Extensive experiments prove the CADG to outperform the DataGuide [11,14] by a factor 5 to 200 on average. For structurally unselective queries, it is over 400 times faster than the DataGuide. The best results were achieved on large collections of heterogeneously structured textual documents.

1 Introduction

Many modern applications produce and process large amounts of semi-structured data which must be queried with both structural and textual selection criteria. The W3C's working drafts supporting full-text search in XQuery and XPath [4,1] illustrate the trend towards the integration of structure- and content-based retrieval [2]. Consider e.g. searching a digital library or archive for papers with, say, a title mentioning *"XML"* and a section about *"SGML"* in the related work part. Obviously, the query keywords (*"XML"*, *"SGML"*) as well as the given structural hints (title, related work) are needed to retrieve relevant papers: searching for *"XML"* and *"SGML"* alone yields many unwanted papers dealing mainly with SGML, whereas a query for all publications with a title and related work possibly selects all papers in the library. The same holds for retrieval in structured web pages or manuals, tagged linguistic or juridical corpora, compilations of annotated monitoring output in informatics or astronomy, e-business catalogues, or web service descriptions. Novel Semantic Web applications will further increase the need for efficient structured document retrieval. All these applications (1) query semi-structured data which (2) contains large text portions and (3) needs persistent indices for both content and structure.

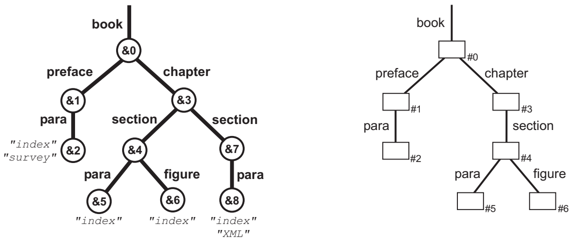
Ongoing research addresses this characteristic class of data by combining data structures from Information Retrieval (IR) and database (DB) research. Common IR techniques for flat text data are *inverted files* [9,24] and *signature files* [9,8]. The ground-breaking DB approach to indexing semi-structured data is the *DataGuide* [11,14], a compact summary of label paths. A mere structure index, it is used with an inverted file for text in [14], but has also been combined with signature files [5] and Tries [6] (see section 6). In these approaches structure and content are still separated and handled sequentially, usually requiring an expensive join at query time. We show that both inverted files and signatures can be tightly integrated with the DataGuide for simultaneous processing of structure and text in all retrieval phases. At the same time, part of the content information is shifted to main memory to minimize retrieval I/O. This improves performance by a factor 5 to 200 on average, and up to 636 under favourable, yet realistic conditions, namely few structural constraints but rather selective keywords. Excelling in poorly structured queries, our approach meets common demands, since users rarely explore the document schema before querying and tend to focus on content rather than structure, accustomed to flat-text web search engines.

This paper is organized as follows. Section 2 describes the DataGuide and a simple query formalism. Section 3 explains approaches to *content awareness*, one of which (the *structure-centric* one) is shown to be superior. Section 4 presents two structure-centric Content-Aware DataGuides, *Inverted File CADG* and *Signature CADG*. Section 5 reports on the experimental comparison of both with the DataGuide. The paper concludes with related work and future research.

2 Indexing XML with the original DataGuide

A well-known and influential DB approach to indexing semi-structured data (in this paper, we focus on tree-shaped XML data) is the *DataGuide* [11,14], a summary of the document tree in which all distinct label paths appear exactly once. Figure 1 *(b)* shows the DataGuide for the document tree in *(a)*. Both differ in that multiple instances of the same document label path, like /book/chapter/section in *(a)*, collapse to form a single index label path *(b)*. Hence the resulting *index tree*, which serves as a path index, is usually much smaller than the document tree (though theoretically it can grow to the same size). Thus it typically resides in main memory even for large corpora.

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 corpus. To this end, every index node is *annotated* with the IDs of those document nodes reached by the same label path as itself. 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. The annotations of all index nodes are stored on disk in an *annotation table (c)*. Formally, it represents a mapping $dg_a : i \mapsto D_i$ where i is an index node ID and D_i the set of document nodes reached by i's label path. Together index tree and annotation table encode nearly the whole structure of the document tree (except parent/child node pairs).



(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

Fig. 1. Data structures of the original DataGuide

The DataGuide, as a path index, ignores textual content. In [14], it is combined with an inverted file mapping any keyword k to the set D_k of document nodes where k occurs, $dg_c : k \mapsto D_k$. The file resides in a separate *content table* on disk, as figure 1 *(d)* shows. (Non-first normal form (NF^2) is not mandatory.)

In our tree query language [15], there are *structural* query nodes, matched by document nodes with a suitable label path, and *textual* ones containing keywords. A query path consists of structural nodes, linked by labelled edges, and possibly a single textual leaf holding a con- or disjunction of query keywords (see figure 2). Edges to structural query nodes may be either *rigid* (solid line), corresponding to XPath's `child` axis, or *soft* (dashed line, `descendant`). Similarly, if a textual query node has a rigid edge, its keywords must occur directly in a document node matching the parent query node. With a soft edge, they may be nested deeper in the node's subtree. Query processing with the DataGuide requires four *retrieval phases*:

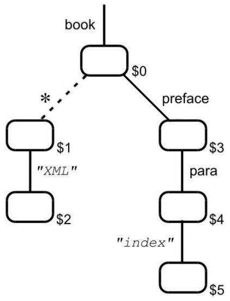


Fig. 2. Query tree

1. *Path matching:* the query paths are matched separately in the index tree.
2. *Occurrence fetching:* annotations for 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 joined (i.e. in the easiest case, intersected).
4. *Path join:* path results are combined to hits matching the entire query tree.

While phases 1 and 3 take place in main memory, phase 2 involves at least two disk accesses. Phase 4 may, but need not, require further I/O operations. The following example shows drawbacks of separate structure and content matching.

Example Consider the query /`book`// $*$ [`"XML"`] (left path in figure 2), selecting all document nodes below a `book` root which contain the keyword *"XML"*. In phase 1, the query path /`book`//$*$ is searched in the index tree from figure 1 *(b)*. All index nodes except the root qualify as structural hits. In phase 2, fetching the annotations of the index nodes #1 to #6 in *(c)* yields the six annotation sets {&1}, {&2}, {&3}, {&4; &7}, {&5; &8}, and {&6}. Obviously unselective query paths featuring // and ∗ may cause multiple index nodes to be retrieved during path matching. According to *(d)* the occurrence set of *"XML"* is {&8}. In phase 3, it is intersected with each annotation set to find out which of the structural matches contain the query keyword. Here almost all candidate index nodes are discarded, their respective annotation set and occurrence set being disjoint. Only #5 references a document node meeting both criteria. The document node in the intersection $\{\&5; \&8\} \cap \{\&8\} = \{\&8\}$ is returned as the only hit for the query node $1. The second path in figure 2 is processed analogously. Path joining (phase 4) is omitted here. Soft-edged textual nodes entail exhaustive index navigation and a more complex join (see [21] for details and optimizations).

In the above example many false positives (all index nodes except #5) survive path matching and fetching, to be finally ruled out in retrieval phase 3. Not only does this make phase 1 unnecessarily complex; it also causes needless I/O in phase 2. Since structural and textual selection criteria are satisfied when considered in isolation, only phase 3 reveals the mismatch. Note that a reverse matching order – first keyword fetching, then navigation and annotation fetching – has no effect, 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 phases 1 and 2. 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 in case of selective keywords and unselective paths.

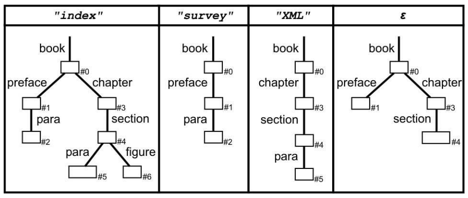
3 Two approaches towards content awareness

Content awareness strives to exploit keyword information during both path matching and annotation fetching in order to suppress needless I/O and joins at query time. It enhances the DataGuide with a materialized content/structure join and a keyword-driven path matching procedure. We propose an exact and a heuristic technique to prune index paths which are irrelevant to a given set of query keywords. This *content-aware navigation* not only reduces the number of paths to be visited, but also excludes false positives from annotation fetching. Precomputing the content/structure join at indexing time allows document nodes to be retrieved simultaneously by their label path and content, avoiding the intersection of possibly large node ID sets at query time. A single *content-aware annotation fetching* step replaces the two table look-ups in phase 2.

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 of restricting path matching to relevant index nodes is to use multiple index trees, each covering the document paths with a specific keyword, as in figure 3 *(a)*. Similarly, annotations are grouped by keyword to materialize the content/structure join in phase 3: replacing the DataGuide tables, a *content/annotation table (b)* maps a keyword k and an index node ID i to the set $D_{k,i}$ of i's annotations containing k, $cadg_{cc} : (k, i) \mapsto D_{k,i}$. Obviously the content/annotation table easily takes up more space on disk than the DataGuide's tables. Its size increases with the number of keywords occurring under many different label paths. This storage overhead is subject to a time/space trade-off common to most indexing techniques. The main drawback of the content-centric approach is that not only the content/annotation table but also the typically large and redundant set of index trees must reside on disk. For each query keyword the right tree needs to be loaded at query time.



(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

Fig. 3. Content-centric CADG

3.2 Structure-centric approach

The approach just presented is *content-centric* in the sense that keywords determine the structure of the index tree and table. A more viable approach to content awareness preserves the tree in its integrity, grouping the keyword occurrences by their label paths. This *structure-centric* approach allows path matching to be performed without loading index trees from disk. The tree resides in main memory like the one of the DataGuide. It resembles figure 1 *(b)*, except that each index node holds enough content information to prune irrelevant 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. Henceforth we refer to the former relation between i and k as *containment* and to the latter as *government*. While government is examined for any index node reached during phase 1 in what we call the *government test*, a *containment test* takes place in phase 2. Both are integrated with the DataGuide's retrieval procedure to enable content-aware navigation and annotation fetching, as follows. In phase 1, whenever an index node i is being matched against a structural query node q_s, the procedure $governs(i, q_s)$ is called. It succeeds iff 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 (at least one in case of a disjunction $\bigvee_{u=0}^{p} k_u$). If so, path matching continues with i's descendants; otherwise i is discarded with its entire subtree. In phase 2, before fetching the annotations of any index node

i matching the parent of a textual query node q_t, the procedure *contains*(i, q_t) is called to verify condition (1) for i and all of q_t's keywords (at least one in case of a disjunction). If it succeeds, i's annotations are fetched; otherwise i is ignored. The realization of *governs*() and *contains*() depends on how content is represented in index nodes—but also in query nodes, which must hint at the keywords below them to avoid repeated exhaustive query tree scans. To this end, suitable *query preprocessing* takes place in a new retrieval phase 0 (see below). Keyword occurrences and annotations are combined in a *content/annotation table* as in figure 4. It can be considered as consisting of seven index-node specific content tables (#0 to #6), each built over a label-path specific view of the data. In first normal form, it is identical to the one in figure 3 *(b)*. Index node and keyword together make up the primary key, enabling content/structure queries as well as pure structure or pure content queries.

#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

Fig. 4. Structure-centric content/annotation table

4 Two realizations of the structure-centric approach: Inverted File CADG and Signature CADG

The concept of a structure-centric CADG does not specify data structures or algorithms for integrating content information with the index and query trees. This section proposes two alternative content representations inspired from IR, along with suitable query preprocessing and government/containment tests. The first approach, which uses inverted files (see section 4.1), is guaranteed to exclude all irrelevant index nodes from path matching and annotation fetching. The signature-based CADG (see section 4.2) represents keywords approximately, possibly missing some irrelevant index nodes. A final verification, performed simultaneously with annotation fetching, eventually rules out false positives. Thus both CADGs produce exact results, no matter if their content awareness is heuristic.

4.1 Inverted File CADG (ICADG)

The *Inverted File CADG (ICADG)* relies on inverted index node ID files to enable content awareness. 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. Assembled in a query preprocessing step (retrieval phase 0), these lists are attached to the query tree (see figure 5). The index tree lacks explicit content information, just like the DataGuide in figure 1 *(b)*. During retrieval phase 1, only ancestors of relevant index nodes are examined, whereas 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 [13,20].

Query preprocessing In retrieval phase 0, each node q in a given query tree is assigned a set I_q of sets of relevant index nodes, as shown in figure 5. Any textual query node q_t with a single keyword k_0 is associated with the set I_{k_0} of index nodes containing k_0, i.e. $I_{q_t} := \{I_{k_0}\}$. I_{k_0} consists of all index nodes paired with k_0 in the content/annotation table (see figure 4). If the query node holds a conjunction $\bigwedge_{u=0}^{p} k_u$ of 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. Analogously, a query node representing a keyword disjunction $\bigvee_{u=0}^{p} k_u$ is associated with the union $I_{q_t} := \{\bigcup_{u=0}^{p} I_{k_u}\}$ of sets of relevant index nodes. If $I_{q_t} = \{\emptyset\}$, the query is immediately rejected as unsatisfiable (without entering retrieval phase 1), because no index node references document nodes with the right content. A 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 $(0 \le v \le m)$, $I_{q_s} := \bigcup_{v=0}^{m} I_{q_v}$. Thus the textual context of a whole query subtree is taken into account while matching any single query path. It is crucial to keep the member sets of all I_{q_v} separate in the higher-order set I_{q_s}, rather than intersect them like with keyword conjunctions. After all the children q_v are not required to be all matched by the same document node, which contains occurrences of all their query keywords at once. Hence the government test for an index node i matching q_s must succeed if 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. For a childless q_s, $I_{q_s} := \emptyset$ is used as a "don't care" to make the government test for q_s succeed (see below). Note that since all children q_v, whether structural or textual, are treated alike the preprocessing procedure also copes with mixed-content queries. Figure 5 depicts the preprocessed query tree from figure 2. Each query node q contains the member sets of I_q (one per row), e.g. $I_{\$0} = \{\{\#5\}; \{\#2; \#5; \#6\}\}$. All ID sets were computed using the content/annotation table in figure 4.

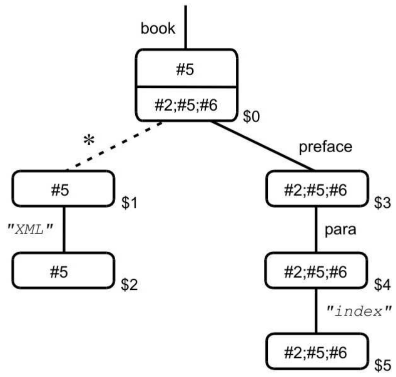


Fig. 5. ICADG query

Government/containment tests As described in section 3.2, $governs(i, q_s)$ is performed when matching an index node i to a structural query node q_s in phase 1. In each set $J \in I_{q_s}$, a descendant of i (including i itself) is searched. The test succeeds iff there is at least one in each J. As a special case, this is true for $I_{q_s} = \emptyset$. In phase 2, $contains(i, q_t)$ tests every index node i matching the parent of a textual query node q_t, provided its government test succeeded. This ensures that i or any of its descendants reference a document node containing q_t's keywords. To determine if annotation fetching for i is justified, the index node is searched in the only member set $J \in I_{q_t}$. The test succeeds iff $i \in J$.

Example Consider the query tree in figure 5 and the DataGuide in figure 1 *(b)*, which is identical to the ICADG tree. Its content/annotation table is given in figure 4. After the index root $\#\mathtt{0}$ matches the query node $\mathtt{\$0}$, $governs(\#\mathtt{0}, \mathtt{\$0})$ succeeds because in both sets associated with $\mathtt{\$0}$, $\{\#\mathtt{5}\}$ and $\{\#\mathtt{2}; \#\mathtt{5}; \#\mathtt{6}\}$, there is a descendant of $\#\mathtt{0}$ (namely $\#\mathtt{5}$). The two paths leaving $\mathtt{\$0}$ are processed one after the other. Reached by a soft edge without label constraint, $\mathtt{\$0}$'s left child $\mathtt{\$1}$ is matched by all index nodes except $\#\mathtt{0}$. First, $governs(\#\mathtt{1}, \mathtt{\$1})$ fails since none of $\#\mathtt{1}$'s descendants is in $\mathtt{\$1}$'s list. This excludes the whole left branch of the index tree from further processing. $\#\mathtt{2}$ never enters path matching, let alone annotation fetching. $\#\mathtt{3}$, as an ancestor of $\#\mathtt{5}$, passes the government test for $\mathtt{\$1}$, but fails in the containment test for $\mathtt{\$2}$ since $\#\mathtt{3} \notin \{\#\mathtt{5}\}$. Its child $\#\mathtt{4}$ satisfies $governs(\#\mathtt{4}, \mathtt{\$1})$ and fails $contains(\#\mathtt{4}, \mathtt{\$2})$ for the same reason as $\#\mathtt{3}$. By contrast, $\#\mathtt{5}$ passes both tests, being a member of $\mathtt{\$1}$'s and $\mathtt{\$2}$'s ID list. Hence its only occurrence of *"XML"*, &8, is fetched from the content/annotation table. $\#\mathtt{6}$ is dismissed by $governs(\#\mathtt{6}, \mathtt{\$1})$. The second query path is processed analogously.

The above query shows how content awareness saves both main-memory and disk operations. Compared to the DataGuide, two subtrees are pruned during phase 1, and only one disk access is performed instead of seven during phase 2. Another one is needed in phase 0 for query preprocessing (for queries with non-existent keywords, it saves the whole evaluation). The results are identical.

4.2 Signature CADG (SCADG)

Unlike the ICADG, the *Signature CADG (SCADG)* uses only approximate keyword information for path pruning. The resulting heuristic government and containment tests may overlook some irrelevant index nodes, recognizing them as false hits only at the content/annotation table look-up in phase 2. (Nevertheless the retrieval is exact, as explained below.) The content information is added not only to the query tree, but also to the index tree, in the form of *signatures* created at indexing time. Since signatures summarize the content information of entire index subtrees, no ancestor/descendant check is needed for the government test.

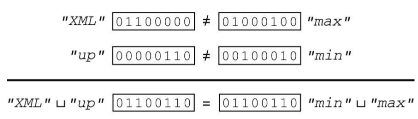


Fig. 6. Ambiguous signatures

Signatures, i.e. fixed-length bit strings, are a common IR technique for concise storage and fast processing of keywords. Every keyword to be indexed or queried is assigned a (preferably unique and sparse) signature. Note that this does not require all keywords to be known in advance, nor to be assigned a signature before indexing. It can be created on the fly, e.g. from a hash code of the character sequence. Sets of keywords in document or query nodes are represented collectively in a single signature, by bitwise disjunction ($\sqcup$) of the individual keyword signatures. As figure 6 shows, overlapping bit patterns may cause ambiguities. Other operations on signatures s_0, s_1 include the bitwise conjunction ($s_0 \sqcap s_1$), inversion ($\neg s_0$), and implication ($s_0 \sqsubset s_1 := (\neg s_0) \sqcup s_1$).

Index tree The Signature CADG's tree (see figure 7) closely resembles the one of the DataGuide (see figure 1), except that each index node i has two signatures. A *containment signature* s_i^c results from disjoining the signatures of all keywords in i's annotations. (If there is none, s_i^c is $\boxed{00000000}$.) A *government signature* s_i^g encodes keywords referenced by i or a descendant. Inner index nodes obtain it by disjoining the government signatures of their children and their own containment signature. For leaf nodes, s_i^g and s_i^c are identical.

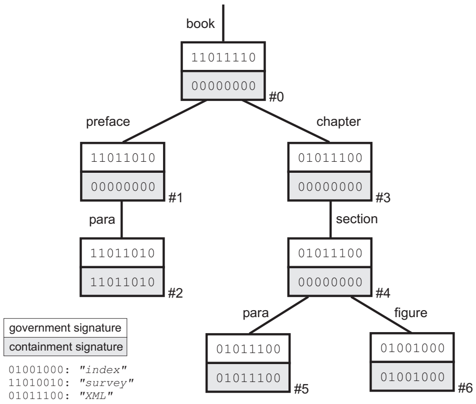


Fig. 7. SCADG index tree

Query preprocessing Unlike index nodes, every query node q has a single signature s_q. For a textual node q_t, s_{q_t} is created from the signatures s_{k_u} of q_t's keywords k_u $(0 \leq u \leq p)$. If k_0 is the only keyword, then $s_{q_t} := s_{k_0}$. For a keyword *conjunction* $\bigwedge_{u=0}^{p} k_u$, $s_{q_t} := \bigsqcup_{u=0}^{p} s_{k_u}$ is the *disjunction* of the keyword signatures (each k_u "leaves its footprint" in s_{q_t}), whereas $s_{q_t} := \bigsqcap_{u=0}^{p} s_{k_u}$ for a disjunction $\bigvee_{u=0}^{p} k_u$ in q_t. A structural query node's signature s_{q_s} superimposes the child signatures s_{q_v} $(0 \leq v \leq m)$, $s_{q_s} := \bigsqcup_{v=0}^{m} s_{q_v}$, to summarize the textual content of the whole subtree below q_s. If childless, q_s is given $\boxed{00000000}$ to make any index node's government test succeed, as one would expect of a query node without textual constraints. Figure 8 shows the tree query from figure 5, preprocessed for the SCADG. Keyword signatures have been taken from figure 7. They are either created on the fly, or fetched from a *signature table* assembled at indexing time. In our experiments this proved faster despite an additional I/O operation at query time and caused only negligible storage overhead.

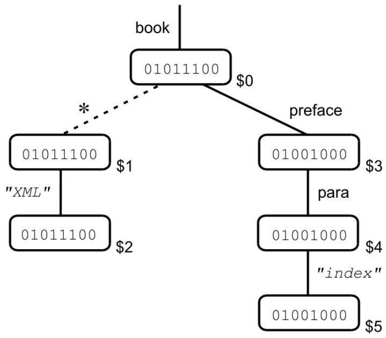


Fig. 8. SCADG query

Government/containment tests The test *governs*(i, q_s) for an index node i and a structural query node q_s is $s_{q_s} \sqsubset s_i^g$, requiring all bits set in s_{q_s} to be also set in s_i^g. This holds when q_s's keywords are governed by i. Yet the converse is not always true: as in figure 6, keywords not covered by s_{q_s} can make s_i^g look as if i were relevant to q_s, and path matching continues below i though its descendants must fail in phase 2. At any rate only irrelevant subtrees are pruned. Analogously, the containment test *contains*(i, q_t) for a textual query node q_t is $s_{q_t} \sqsubset s_i^c$. It succeeds if, but not only if, q_t's keywords occur in i's annotations.

Example Reconsider the query in figure 8 and the index tree from figure 7 with the content/annotation table in figure 4. After $governs(\#0, \$0) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{11011110}}$ succeeds, path matching continues with query node $1, matched by all index nodes except #0. Since $governs(\#1, \$1) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{11011010}}$ fails, the antepenultimate bit being set only in $1's signature, the left index branch is pruned and phase 1 continues with #3. It passes $governs(\#3, \$1) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01011100}}$, but $contains(\#3, \$2) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{00000000}}$ fails. The same is true for #3's only child #4. While #5 passes $governs(\#5, \$1)$ (see $governs(\#3, \$1)$) and $contains(\#5, \$2) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01011100}}$, contributing &8 to the result, $governs(\#6, \$1) = \boxed{\texttt{01011100}} \sqsubset \boxed{\texttt{01001000}}$ fails (bits 4 and 6 are missing in $s^g_{\#6}$), saving a table look-up. The second query path is processed analogously.

The number of disk accesses compared to the DataGuide is reduced from seven to two (including query preprocessing), like with the ICADG. Moreover, signatures are more efficient data structures than node ID sets (in terms of both storage and processing time) and make relevance checks and preprocessing easier to implement. Note, however, that if another keyword with a suitable signature occurred in #6's annotation, e.g. the keyword *"query"* with the signature $\boxed{\texttt{00011100}}$, then #6 would be mistaken as relevant for $5's query keyword *"XML"*. The reason is that both $s^g_{\#6}$ and $s^c_{\#6}$, superimposing the *"index"* and *"query"* signatures, would equal the keyword signature for *"XML"*, $\boxed{\texttt{01001000}} \sqcup \boxed{\texttt{00011100}} = \boxed{\texttt{01011100}}$. Hence $governs(\#6, \$1)$ and $contains(\#6, \$2)$ would succeed. Only in phase 2 would #6 turn out to be a false hit. This illustrates how the SCADG trades off pruning precision against navigation efficiency.

5 Experimental evaluation

5.1 Experimental set-up

This section describes a part of our experiments with both CADGs and the DataGuide as control. A more detailed report is given in [21]. For the SCADG, we stored 64-bit keyword signatures in a *signature table* at indexing time, created from hash codes with 3 bits set in each signature ([8] surveys signature creation). Extensive tests have been performed on three XML tree corpora with different characteristics (see table 1): *Cities* is small, homogeneous, and non-recursive, whereas *XMark*, a medium-sized synthetically generated corpus [23], is slightly more heterogeneous and contains recursive paths. The highly recursive and heterogeneous *NP* collection comprises half a gigabyte of syntactically analyzed German noun phrases [17]. Both hand-crafted and synthetic query sets were evaluated against the three corpora, resulting in four test suites. Unlike *CitiesM* with 90 manual queries on the *Cities* collection, *CitiesA* (639 queries), *XMarkA* (192 queries), and *NpA* (571 queries) consist of automatically generated queries on the *Cities*,

name	XML size	*nodes*	keywords	*label paths*	depth
Cities	1.3 MB	*16,000*	19,000	*253*	7
XMark	30 MB	*417,000*	84,000	*515*	13
NP	510 MB	*4,585,000*	130,000	*2,349*	40

Table 1. Document collections

XMark, and *NP* collections, respectively. Only path queries were considered in order to minimize dependencies on the underlying evaluation algorithm.

For a systematic analysis of the results, all queries were classified according to six *query characteristics*, summarized in table 2 (see [21] for an extended scheme). Each characteristic of a given query is encoded by one bit 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. Three groups of query characteristic turned out to be most interesting for our purposes. First, bit 5 distinguishes satisfiable (`0-----`) from unsatisfiable (`1-----`) queries (read "`-`" as "don't care"). Bits 4 to 2 concern the navigational effort during evaluation: queries with `-000--` signatures, being structurally unselective, cause many index paths to be visited. Finally, 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 [21]). For instance, the query classes `0---10` contain satisfiable queries whose keywords occur often in the documents, though only under a small number of different label paths. All 64 path query classes are populated in the test suites, and only few with less than three queries.

5	`1-----`	*query result*	mismatch
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 2. Path query classification scheme

The index structures were integrated into the XML retrieval system X^2 [15]. Since large parts of the query evaluation algorithms and even index algorithms are shared by all 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 1800+, 1 GB RAM, running SuSE Linux 8.2 with kernel 2.4.20). The PostgreSQL relational database system, version 7.3.2, was used as relational backend for storing the index structures (with database cache disabled). To compensate for file system cache effects, each query was processed once without taking the results into account. The following three iterations of the same query were then averaged.

5.2 Results

Figure 9 shows the performance results for three selected sets of query classes, as indicated by the plot titles: while the left column covers all evaluated queries (`------`), the one in the middle narrows down to those queries with mostly unlabelled soft edges (`-000--`). The right column is restricted to all matching queries (`0-----`). The three bar charts in the upper row depict, on a logarithmic scale, the *average speedup* of ICADG (▧) and SCADG (■) over the DataGuide, i.e. the ratio of the DataGuide's to the respective CADG's evaluation time. For each of the four test suites, the speedup was averaged in three steps: first over the three iterations of each query, then over all queries in each query class, and finally over all classes selected for the given plot. This step-wise averaging ensures that query classes of different cardinality are equally weighted. The three plots in the second row depict the ICADG's *speedup distribution*, again on a logarithmic scale. For each test suite, the corresponding graph indicates for how many queries (in percent of the total number of queries in the selected classes; ordinate) the ICADG achieved a specific speedup over the DataGuide (abscissa). As indicated by the position of the symbols (+,□,◇,×), queries have been grouped into five speedup intervals ($(-\infty, 1)$; $[1, 2)$; $[2, 10)$; $[10, 100)$; $[100, \infty)$). For convenience, the distributions are given as interpolating functions rather than as histograms.

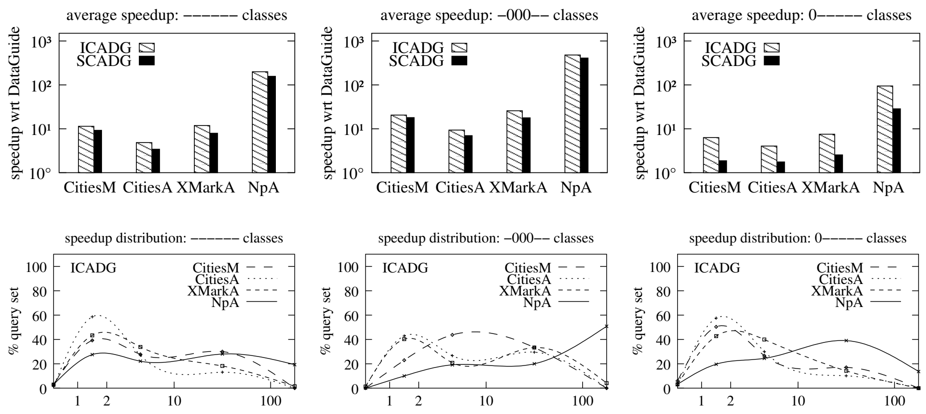


Fig. 9. Retrieval time CADG vs. DataGuide

As shown in the upper left chart in figure 9, the ICADG always performs a little better than the SCADG, beating the DataGuide by a factor 5 in the worst case (*CitiesA*). In the *CitiesM* and *XMarkA* test suites, the ICADG reaches an average speedup of one order of magnitude. On the most challenging collection, *NP*, it is 200 times faster than the DataGuide. As expected, the speedup increases for poorly structured queries (upper middle), where the potential for path pruning is higher. The speedup gain (ICADG 80-140%, SCADG 90-160%) grows with the size of the corpus. In *NpA*, the ICADG evaluates structurally unspecific queries 479 times faster than the DataGuide on average. The distribution plot for `-000--` queries (lower middle) illustrates how under these conditions the bulk of queries is shifted from $[1, 2)$ to $[2, 10)$ for *CitiesM* (— —), and to $[10, 100)$ for *CitiesA* (······) and *XMarkA* (- - - -). For *NpA* (——), the portion of $[1, 2)$ queries drops to 10% (formerly 28%), while the $[100, \infty)$ portion jumps from 19% to 51%, i.e. the ICADG processes one out of two queries by two orders of magnitude faster than the DataGuide. Higher keyword selectivity (`----11` and `-00011`, omitted) again increases the speedup by 10-20% on average, and up to 30% for the ICADG. Yet content awareness pays off even for unselective keywords.

The two plots in the right column of figure 9 focus on the subset of satisfiable queries (`o-----`). It makes up 50% of the test suites. While the ICADG's speedup still reaches 4-7 in the smaller test suites (vs. 5-12 for all queries) and two orders of magnitude in *NpA*, the SCADG performs only twice as good as the DataGuide on the *Cities* and *XMark* collections. On *NP* it beats the DataGuide by one order of magnitude (avg. speedup 28). Obviously the SCADG excels at filtering out unsatisfiable queries, especially those with non-existing keywords which it rejects in retrieval phase 0. In practice this is a valuable feature, as users are unwilling to accept long response times when there is no result in the end. Moreover, a suitable evaluation algorithm might list query hits incrementally, allowing users to browse the result while evaluation continues. Yet for unsatisfiable queries there is nothing to display, such that every second of the response time is lost.

The experiments also prove that both CADGs are most effective for large document collections such as *NP*, in terms of both retrieval time and storage. As shown in figure 10, the ICADG grows to 87% (2.4 MB) and the SCADG to 168% (4.6 MB) of the size of *Cities* in the database (DataGuide 1.6 MB). However, this storage overhead is reduced considerably for *XMark* and completely amortized for *NP* (ICADG 3% (21 MB), SCADG 6% (36 MB), DataGuide 3% (15 MB)). The SCADG's overhead over the ICADG, with 64-bit signatures in the signature table and index tree, ranged from 2 MB (*Cities*) to 16 MB (*NP*). 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 [9]. This further reduces the storage overhead. The resulting index, although inexact, is well suited for XML result ranking [10,18,12,22].

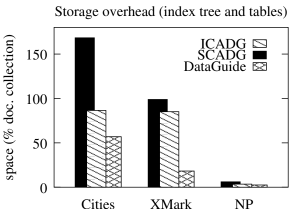


Fig. 10. Index size

6 Related work

In this section we focus on related XML index structures with support for textual content. Work on index structures for XML in general is surveyed in [20].

Closest in spirit to the CADG approach are the *IndexFabric* [6] and the *Joined Lists* approach proposed in [12]. Based on the DataGuide, both provide a link between index nodes and keyword occurrences, a crucial concept for efficient query evaluation. In the case of the IndexFabric, this link is explicit: the index nodes are enriched with Tries representing textual content. This is equivalent to the CADGs' materialized join. Additionally, the IndexFabric is equipped with a sophisticated layered storage architecture, but provides no content-aware navigation like CADGs. Joined Lists are inverted files that are joined during query evaluation. Index nodes and keyword occurrences are linked indirectly with references from each keyword occurrence to the corresponding index node. Unlike CADGs, Joined Lists involve neither materialized joins nor content-aware navigation, relying on suitable join algorithms for efficient query evaluation.

The *BUS index* [19] integrates structured document retrieval and relevance ranking based on term frequency in document nodes. Keywords are mapped to the containing document and index nodes. This results in a content/structure join at query time, albeit at index node rather than document node level. CADGs abandon this join and use content awareness to minimize path matching and I/O.

The *Signature File Hierarchy* [5] provides content-aware path matching like the SCADG. However, without signatures for index nodes, or text blocks as proposed in [8], a new government signature must be fetched from disk for each document node during path matching. This entails a significant I/O overhead. Pruning is less effective by lack of containment signatures in the index tree.

More IR-oriented approaches like [7] tend to use the document structure for determining which parts of the data to retrieve and for ranking, but not for path matching. Label paths are therefore not represented as a navigable index tree.

The *Context Index* [16], extending inverted-file based indices, uses *structure signatures* to discard mismatches early during retrieval. They hold approximate information about the structural context (label path) of keyword occurrences.

Materialized Schema Paths [3] represent the structural context of keywords, entire text portions, and node labels statistically. In a content-centric approach (see section 3.1), they can be used to index keyword-specific corpus fragments.

7 Conclusion

Results The Content-Aware DataGuide (CADG) is an efficient index for textual XML data. Combining structure and text matching during all retrieval phases by means of standard IR and DB techniques, it abandons joins at query time and avoids needless I/O operations. Among the two concrete realizations of the CADG, the faster and smaller ICADG is more promising than the heuristic SCADG, in accordance with results from earlier work on flat text data [24]. Based on a novel query classification scheme, experiments prove that the ICADG outperforms the DataGuide on large corpora by two orders of magnitude on average. It is most effective for queries with little structure and selective keywords, which have been shown to be most important in real-world applications. The greatest speedup (factor 636) and lowest storage overhead (3% of the original data) is achieved for a large, heterogeneous document collection of 510 MB.

Future Work IR and DB integration for structured document retrieval has advanced recently, but is still far from complete. We plan to adapt the CADG to XML ranking models from IR [10,18,12,22] and to enhance the X^2 retrieval system [15] with index browsing [11] and relevance feedback. A further optimized ICADG may support keyword negation in queries. One might also investigate techniques for adaptively increasing the level of content awareness based on query statistics. An ongoing project tries to amalgamate DataGuide and CADG techniques with the rather limited indexing support for XML in commercial relational database systems. We also plan to refine the query classification scheme [21] and to assess content-aware navigation and the materialized join separately.

References

1. S. Amer-Yahia and P. Case. XQuery and XPath Full-Text Use Cases. W3C Working Draft, 2003. See `http://www.w3.org/TR/xmlquery-full-text-use-cases`.
2. R. Baeza-Yates and G. Navarro. Integrating Contents and Structure in Text Retrieval. *SIGMOD Record*, 25(1):67–79, 1996.
3. M. Barg and R. K. Wong. A Fast and Versatile Path Index for Querying Semi-Structured Data. *Proc. 8th Int. Conf. on DBS for Advanced Applications*, 2003.
4. S. Buxton and M. Rys. XQuery and XPath Full-Text Requirements. W3C Working Draft, 2003. See `http://www.w3.org/TR/xquery-full-text-requirements`.
5. Y. Chen and K. Aberer. Combining Pat-Trees and Signature Files for Query Eval. in Document DBs. *Proc. 10th Int. Conf. on DB & Expert Systems Applic.*, 1999.
6. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. *Proc. 27th Int. Conf. on Very Large DB*, 2001.
7. H. Cui, J.-R. Wen, and T.-S. Chua. Hier. Indexing and Flexible Element Retrieval for Struct. Document. *Proc. 25th Europ. Conf. on IR Research*, pages 73–87, 2003.
8. C. Faloutsos. Signature Files: Design and Performance Comparison of Some Signature Extraction Methods. *Proc. ACM-SIGIR Int. Conf. on Research and Development in IR*, pages 63–82, 1985.
9. W. B. Frakes, editor. *IR. Data Structures and Algorithms.* Prentice Hall, 1992.
10. N. Fuhr and K. Großjohann. XIRQL: A Query Language for IR in XML Documents. *Research and Development in IR*, pages 172–180, 2001.
11. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proc. 23rd Int. Conf. on Very Large DB*, 1997.
12. R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. *Proc. 20th Int. Conf. on Data Engineering*, 2004. To appear.
13. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proc. 27th Int. Conf. on Very Large DB*, pages 361–370, 2001.
14. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A DB Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, 1997.
15. H. Meuss, K. Schulz, and F. Bry. Visual Querying and Explor. of Large Answers in XML DBs with X^2. *Proc. 19th Int. Conf. on DB Engin.*, pages 777–779, 2003.
16. H. Meuss and C. Strohmaier. Improving Index Structures for Structured Document Retrieval. *Proc. 21st Ann. Colloquium on IR Research*, 1999.
17. J. Oesterle and P. Maier-Meyer. The GNoP (German Noun Phrase) Treebank. *Proc. 1st Int. Conf. on Language Resources and Evaluation*, 1998.
18. T. Schlieder and H. Meuss. Querying and Ranking XML Documents. *JASIS Spec. Top. XML/IR 53(6):489-503*, 2002.
19. D. Shin, H. Jang, and H. Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. *Proc. 3rd ACM Int. Conf. on Digital Libraries*, 1998.
20. F. Weigel. A Survey of Indexing Techniques for Semistructured Documents. Technical report, Dept. of Computer Science, University of Munich, Germany, 2002.
21. F. Weigel. Content-Aware DataGuides for Indexing Semi-Structured Data. Master's thesis, Dept. of Computer Science, University of Munich, Germany, 2003.
22. J. E. Wolff, H. Florke, and A. B. Cremers. Searching and Browsing Collections of Structural Information. *Advances in Digital Libraries*, pages 141–150, 2000.
23. XML Benchmark Project. A benchmark suite for evaluating XML repositories. See `http://monetdb.cwi.nl/xml`.
24. J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted Files Versus Signature Files for Text Indexing. *ACM Transactions on DB Systems*, 23(4):453–490, 1998.