Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems

Felix Weigel* Klaus U. Schulz* Holger Meuss‡

* Centre for Information and Language Processing
University of Munich (LMU), Germany
{weigel,schulz}@cis.uni-muenchen.de

‡ European Southern Observatory
Headquarter Garching, Germany
hmeuss@eso.org

ABSTRACT

In XML retrieval, two distinct approaches have been established and pursued without much cross-fertilization taking place so far. On the one hand, *native XML databases* tailored to the semistructured data model have received considerable attention, and a wealth of index structures, join algorithms, tree encodings and query rewriting techniques for XML have been proposed. On the other hand, the question how to make XML fit the relational data model has been studied in great detail, giving rise to a multitude of *storage schemes* for XML in relational database systems (RDBSs). In this paper we examine how native XML indexing techniques can boost the retrieval of XML stored in an RDBS. We present the *Relational CADG (RCADG)*, an adaptation of several native indexing approaches to the relational model, and show how it supports the evaluation of a clean formal language of conjunctive XML queries. Unlike relational storage schemes for XML, the RCADG largely preserves the underlying tree structure of the data in the RDBS, thus addressing several open problems known from the literature. Experiments show that the RCADG accelerates retrieval by up to two or even three orders of magnitude compared to both native and relational approaches.

Categories and Subject Descriptors

H.3.1 [**Information Storage And Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.3 [**Inform. Storage And Retrieval**]: Inform. Search and Retrieval

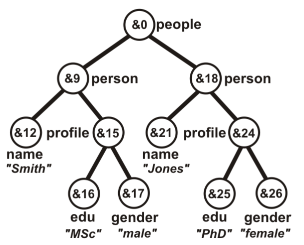
General Terms Algorithms, Performance

1. INTRODUCTION

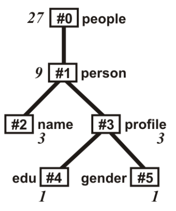
XML is ever more used as data model for storing and querying persistent semistructured data. Query languages like XPath and XQuery are de-facto standards, and today applications relying on XML storage and retrieval scale up to the multi-gigabyte level. Motivated by the widespread use of XML, the development of *native XML databases* has received much attention [23, 21, 20, 24]. Throughout this work, the term "native" subsumes systems whose evaluation kernel operates outside a general-purpose database system, although such a system may be used as back-end, and which manipulates tree data structures rather than, say, flat sets of tuples. Meanwhile many index structures for native XML engines have been proposed [8, 25, 16, 30, 22]. But now that scalability and retrieval efficiency are major concerns, storing and querying XML data in a *relational database system (RDBS)* seems particularly promising because (1) efficient access methods for relational data have been developed for over thirty years and (2) query planning and optimization in the relational algebra is well-understood. Besides, RDBSs are widely deployed and offer key features for a productive use, e.g., concurrency, transactions and safety.

Consequently, a variety of relational storage schemes for XML have emerged, which are either based on a fixed DTD [28, 1, 3] or generic in nature [11, 17, 27, 34, 29, 13]. All of these "shred" the hierarchical XML data into tuples, requiring expensive table joins to restore part of the node hierarchy at query time. Approaches like "inlining" [28, 7], meant to avoid the disadvantages of shredding, can only partially remedy this effect. Moreover, the existing storage schemes rely mostly on relational index structures, oblivious of the native XML indexing techniques developed over the years.

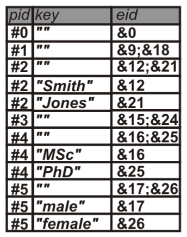
The aim of this paper is to investigate how advanced index structures for native XML like the Content-Aware DataGuide (CADG) [30] and the BIRD tree encoding [32] can be integrated with a relational storage and retrieval scheme, and how this improves the performance of both native and relational XML databases. We present the *Relational CADG (RCADG)*, a DataGuide [8] derivate consisting entirely of relational data structures. The RCADG indexes the tree structure (or *schema*) of the data in a concise manner, and associates it with the actual elements and keyword occurrences via a foreign key. Its schema part is typically several orders of magnitude smaller than the original document tree. Unlike all other relational approaches we know of, the RCADG preserves the compositional nature of label paths as sequences of nodes. This drastically reduces the number of elements to be retrieved from disk, speeding up retrieval considerably. Thus the virtues of the tree-based native XML index carry over to its flat relational counterpart. When matching query constraints directly against the elements, the BIRD encoding replaces expensive joins with simple arithmetic operations, leading to great performance gains. Besides, the RCADG benefits largely from the mature relational indexing and query planning techniques, unlike prior DataGuide-based approaches [34, 27, 17], thus achieving indeed a synergy between native and relational XML database research to combine the best from both worlds.



a. document tree



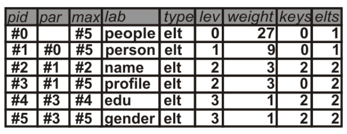
b. index tree



pid	key	eid
#0	""	&0
#1	""	&9;&18
#2	""	&12;&21
#2	"Smith"	&12
#2	"Jones"	&21
#3	""	&15;&24
#4	""	&16;&25
#4	"MSc"	&16
#4	"PhD"	&25
#5	""	&17;&26
#5	"male"	&17
#5	"female"	&26

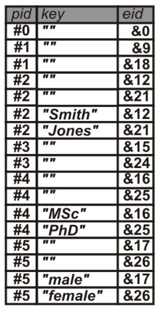
c. element table

Fig. 1: CADG. In *a.* and *c.*, &0 denotes a BIRD element ID. In *b.* and *c.*, #0 denotes a path ID and 27 a BIRD weight (Section 2.3).



pid	par	max	lab	type	lev	weight	keys	elts
#0		#5	people	elt	0	27	0	1
#1	#0	#5	person	elt	1	9	0	1
#2	#1	#2	name	elt	2	3	2	2
#3	#1	#5	profile	elt	2	3	0	2
#4	#3	#4	edu	elt	3	1	2	2
#5	#3	#5	gender	elt	3	1	2	2

a. path table



pid	key	eid
#0	""	&0
#1	""	&9
#1	""	&18
#2	""	&12
#2	""	&21
#2	"Smith"	&12
#2	"Jones"	&21
#3	""	&15
#3	""	&24
#4	""	&16
#4	""	&25
#4	"MSc"	&16
#4	"PhD"	&25
#5	""	&17
#5	""	&26
#5	"male"	&17
#5	"female"	&26

b. element table

Fig. 2: RCADG. *a.* Path table with one tuple per index node (cf. Fig. 1 *b.*). *b.* Element table in 1NF.

The paper is organized as follows. Sect. 2 introduces formal preliminaries before reviewing native XML indexing techniques this work builds on, namely the CADG index and BIRD encoding. Sect. 3 explains how the Relational CADG combines these approaches in a purely relational index structure for XML. Translating conjunctive XML queries into SQL queries against the RCADG is explained in Sect. 4. The results of our experimental evaluation of the RCADG are given in Sect. 5. Finally, Sect. 6 reviews and compares related work before Sect. 7 concludes with future work.

2. PRELIMINARIES

2.1 Data and query model

Let Λ be a finite alphabet of element labels. A *document tree* is a finite ordered rooted tree $D = \langle E, r, \mathit{Child}, \mathit{NextSib}, \lambda\rangle$ where E is the finite and non-empty set of nodes (elements), *Child* is a binary relation on E such that $\langle E, \mathit{Child}, r\rangle$ is an unordered rooted tree with root r, $\mathit{NextSib} \subseteq E \times E$ relates a child to its right sibling, and $\lambda : E \to \Lambda$ assigns a label $\lambda(e) \in \Lambda$ to each node $e \in E$. The *label path* of any element $e \in E$ is the sequence of labels $\lambda(e_0)\cdots\lambda(e_k)$ of all elements $r = e_0, \ldots, e_k = e$ on the root path of element e (i.e., where $\langle e_i, e_{i+1}\rangle \in \mathit{Child}$ for all $0 \le i < k$). Let P be the set of distinct label paths in D. Then the function $\pi : E \to P$ maps any element $e \in E$ to its unique label path in D.

Besides *Child*, *NextSib* and their reverse counterparts (i.e., *Parent* and *PrevSib*) we consider some other binary relations between elements, similar to the set of "generalized XPath axes" defined in [10].[1] For any $e, e' \in E$, $\mathit{NextElt}(e, e')$ holds iff element e' occurs after e in the XML serialization of D, and *PrevElt* is the corresponding inverse relation. For all relations R mentioned so far, we define lower and upper proximity bounds as follows: $R^{\max}_{\min} = \bigcup_{\min \le i \le \max} R^i$ where R^i denotes the i-fold composition $R \circ \cdots \circ R$ of R. In this sense, R is equivalent to R^1_1. The symbol $*$ acts as a "don't care" bound. For instance, *Child* corresponds to the XPath axis child, Child^*_1 to descendant and Child^*_0 to descendant-or-self. The definition of *Following*, *Preceding* and *Self* is obvious. The remaining XPath axes may be modelled based on these relations, collectively referred to as the set $\mathcal{R}_2$ of binary relations, and a set $\mathcal{T} = \{\mathit{element}, \mathit{attribute}\}$ of unary relations indicating the type of any node $e \in E$. Besides, for each $l \in \Lambda$, there is a unary relation lab_l containing exactly all elements $e \in E$ where $\lambda(e) = l$. Furthermore, to indicate the level of any element in the document tree, we define $\mathit{lev}^{\max}_{\min}$ as the relation containing all elements on levels $\min \le i \le \max$, with $\mathit{lev}^0_0 = \{r\}$. Finally, to model the textual contents of XML documents, we define a relation $\mathit{con}_k \subset E$ for each k in the set K of keywords occurring in the data. Given an element $e \in E$, $e \in \mathit{con}_k$ ("e contains k") iff there is a textual occurrence of k in e in the obvious sense. The type, label, level and keyword relations together make up the set $\mathcal{R}_1$ of unary relations on E.

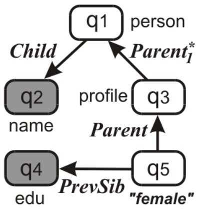


Fig. 3: Query.

We now define a clean formal query model capturing the most relevant features of XPath, similar to *Core XPath* [9]. Like [10], we ignore features like data types, functions, iteration etc. A *(conjunctive) query* Q is a triple $\langle Q_n, Q_r, Q_c\rangle$ where Q_n is a finite non-empty set of query nodes, $Q_r \subset Q_n$ is the non-empty set of result nodes (shaded in Fig. 3) and Q_c is a finite non-empty set of constraints of the form $R_1(q)$ or $R_2(q, q')$ where $q, q' \in Q_n$, $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$. Note that the resulting query graph $\langle Q_n, Q_c\rangle$ need not be acyclic as in Fig. 3. A *match* to Q is a mapping $\mu : Q_n \to E$ such that $\mu(q) \in R_1$ for each unary constraint $R_1(q) \in Q_c$ and $\langle \mu(q), \mu(q')\rangle \in R_2$ for each $R_2(q, q') \in Q_c$. The (partial) *query result* ϱ_μ is the restriction of μ to Q_r.

2.2 Content-Aware DataGuide (CADG)

The *Content-Aware DataGuide (CADG)* [30], a combined structure and text index for XML, consists of two data structures. The *index tree* I (Fig. 1 *b.*) is a summary of the structure of the document tree (Fig. 1 *a.*), also referred to as a *(descriptive) schema*. Each distinct label path $p \in P$ occurs exactly once in I. In the formal definition, we may therefore treat index nodes and label paths as interchangeable. Let P be the set of distinct label paths in the document tree D. The index tree I is the finite rooted unordered node-labelled tree $\langle P, \pi(r), \mathit{Child}, \mathit{Sib}, \lambda'\rangle$ whose nodes are the label paths in D and whose root is the label path of the root r in D. For any two label paths p and p', $\langle p, p'\rangle \in \mathit{Child} \subset P \times P$ iff there is a label $l \in \Lambda$ such that $p' = pl$, and $\langle p, p'\rangle \in \mathit{Sib} \subset P \times P$ iff there is a path p'' such that $\langle p'', p\rangle \in \mathit{Child}$ and $\langle p'', p'\rangle \in \mathit{Child}$. The *Parent* and *Self* relations and all unary constraints defined in Sect. 2.1 apply to index nodes in the obvious way. The labelling function λ' maps a label path $p \in P$ to the last symbol $l \in \Lambda$ in p. For handling keyword constraints in the index tree, see [30]. The second data structure of the CADG is the *element table* (Fig. 1 *c.*) which materializes a relation $\mathit{Occur} \subset P \times K \times E$. For any element $e \in E$ and keyword $k \in K$ in the data, $\langle \pi(e), k, e\rangle \in \mathit{Occur}$

[1]That work also proved that the evaluation of queries involving the full set of relations mentioned below is NP-complete. Polynomial fragments [10] are not considered here.

$$\frac{\mathit{Child}^*_1(e_0, e_1)}{e_1.\mathit{eid} > e_0.\mathit{eid} \wedge \\ e_1.\mathit{eid} < e_0.\mathit{eid} + \pi(e_0).\mathit{weight}}\ (\mathrm{D}^{\mathit{Dec}}_{\mathit{Child}^*_1}) \qquad \frac{\mathit{NextSib}^*_1(e_0, e_1) \quad \exists e_2 : \mathit{Parent}(e_0, e_2)}{e_1.\mathit{eid} > e_0.\mathit{eid} \wedge e_1.\mathit{eid} < e_2.\mathit{eid} + \pi(e_2).\mathit{weight} \wedge \\ e_1.\mathit{eid} \bmod \pi(e_0).\mathit{weight} = 0}\ (\mathrm{D}^{\mathit{Dec}}_{\mathit{NextSib}^*_1})$$

$$\frac{\mathit{Following}(e_0, e_1)}{e_1.\mathit{eid} \geq e_0.\mathit{eid} + \pi(e_0).\mathit{weight}}\ (\mathrm{D}^{\mathit{Dec}}_{\mathit{Following}}) \qquad \frac{\mathit{NextElt}^*_1(e_0, e_1)}{e_1.\mathit{eid} > e_0.\mathit{eid}}\ (\mathrm{D}^{\mathit{Dec}}_{\mathit{NextElt}^*_1}) \qquad \frac{\mathit{Self}(e_0, e_1)}{e_1.\mathit{eid} = e_0.\mathit{eid}}\ (\mathrm{D}^{\mathit{Dec}}_{\mathit{Self}})$$

Fig. 4: BIRD data-level matching rules for deciding forward binary relations ($e_0, e_1, e_2 \in E$; for reverse relations see [31]).

$$\frac{\mathit{Parent}^*_1(e_0, e_1)}{e_1.\mathit{eid} = e_0.\mathit{eid} - (e_0.\mathit{eid} \bmod \pi(e_1).\mathit{weight})}\ (\mathrm{D}^{\mathit{Rec}}_{\mathit{Parent}^*_1}) \qquad \frac{\mathit{Self}(e_0, e_1)}{e_1.\mathit{eid} = e_0.\mathit{eid}}\ (\mathrm{D}^{\mathit{Rec}}_{\mathit{Self}})$$

$$\frac{\mathit{PrevSib}^i_i(e_0, e_1) \quad \exists e_2 : \mathit{Parent}(e_0, e_2) \ \wedge \ e_0.\mathit{eid} - i \cdot \pi(e_0).\mathit{weight} > e_2.\mathit{eid}}{e_1.\mathit{eid} = e_0.\mathit{eid} - i \cdot \pi(e_0).\mathit{weight}}\ (\mathrm{D}^{\mathit{Rec}}_{\mathit{PrevSib}^i_i})$$

Fig. 5: BIRD data-level matching rules for reconstructing binary relations ($e_0, e_1, e_2 \in E$).

iff e contains k as defined in Sect. 2.1. An extra entry for each element and the empty keyword serves for matching query nodes without keyword constraints.

Note that while the index tree resides in memory in the original work, being typically orders of magnitude smaller than the document tree, the element table is stored in an RDBS. During the evaluation of a query Q, the unary and binary query constraints in Q_c are first checked against the schema, which yields a restricted set of label path candidates to be looked up in the element table. When evaluating keyword constraints $\mathit{con}_k(e)$ (where k may be empty) attached to a query node $q \in Q_n$ against the data, only those tuples $\langle \pi(e), k, e\rangle$ are selected from the element table where $\pi(e)$ is a candidate label path for q. In the end, the binary constraints are checked once more against these elements in order to retrieve matchings for the entire query graph (s. b.).

2.3 BIRD tree encoding

Checking query constraints on the schema level discards all mismatching label paths, but even occurrences of candidate paths may not combine into matches to the whole query graph. Verifying the binary constraints again on the data level is accelerated considerably if the unique element IDs encode part of the structure of the document tree. The *BIRD scheme* [32] not only *decides* for two given element IDs whether a specific binary relation holds between those elements (e.g., "decide whether e' is the parent element of e"). Given the corresponding index node, one can also compute any ancestor or preceding sibling of a given element without accessing the element table, a feature which we refer to as the *reconstruction* of element IDs. Experiments [32] show that reconstruction capabilities are key to efficient query evaluation. This motivates the use of BIRD for the RCADG, although other tree encodings might be used, too.

Due to space limitations, we only sketch the BIRD scheme here; for details see [32]. In a nutshell, each label path $p \in P$ is assigned a single integer *weight* reflecting the maximum descendant count of any element with path p. Weights can be conveniently stored as fields of the index nodes (Fig. 1 *b.*). Each element e is given as ID ($e.\mathit{eid}$) a multiple of the weight of $\pi(e)$ ($\pi(e).\mathit{weight}$) such that besides uniqueness and compatibility with document order, special decision and reconstruction properties of BIRD IDs are satisfied [31, 32], some of which are listed in Fig. 4–5. The evaluation algorithm in Sect. 4 guarantees that the label path $\pi(e)$ is known for any $e \in E$ to which a decision or reconstruction rule is applied.

The rules in Fig. 4 translate binary query constraints corresponding to XPath's forward axes into suitable join conditions during RCADG evaluation. Numerators list "input" (i.e., given query constraints) and denominators "output" (i.e., deduced conditions on BIRD IDs). For instance, applied to element e_0 = &24 in Fig. 1 *a.*, whose label path $\pi(e_0)$ = #3 has the weight 3 (Fig. 1 *b.*), rule $\mathrm{D}^{\mathit{Dec}}_{\mathit{Child}^*_1}$ states that all elements e_1 with $24 < e_1.\mathit{eid} < 24 + 3$ are e_0's descendants. With the full set of rules [31], BIRD decides all binary constraints mentioned in Section 2.1.[2] Depending on the evaluation strategy, however, it is often more efficient to reconstruct rather than decide certain relations, because decision requires two IDs to be given whereas reconstruction computes an unknown ID from a single given one. In particular, the parent ID e_2 needed in rule $\mathrm{D}^{\mathit{Dec}}_{\mathit{NextSib}^*_1}$ (Fig. 4) may be obtained via reconstruction. Fig. 5 summarizes reconstruction with the BIRD scheme. For e_0 = &24, e_2 = &18 according to rule $\mathrm{D}^{\mathit{Rec}}_{\mathit{Parent}^*_1}$ ($e_2.\mathit{eid} = 24 - (24 \bmod 9) = 18$). Note that apart from the trivial reconstruction of the *Self* relation (rule $\mathrm{D}^{\mathit{Rec}}_{\mathit{Self}}$), only ancestors and preceding siblings at an arbitrary, but fixed distance i can be reconstructed.

3. Relational CADG (RCADG)

The RCADG migrates the CADG entirely to the relational data model, with several benefits: (1) the overhead at query time is reduced since no native-to-relational interface (such as JDBC) separates the index tree and evaluation algorithm from the database back-end; (2) large intermediate results need not be loaded into main memory; (3) RDBS features such as indexing, query optimization and concurrency are available at the schema level; (4) tree-related statistics stored in the RCADG can enhance the RDBS query planner. The impact of these advantages is quantified in Section 5.2.

As a summary of the document tree, the index tree could be shredded into tuples using any of the relational storage schemes listed in Section 1. In order to avoid fragmentation and to keep the evaluation as simple as possible, we choose a generic (i.e., DTD-oblivious) storage scheme representing each index node (i.e., label path) p as a tuple $\langle \mathit{pid}, \mathit{par}, \mathit{max}, \mathit{lab}, \mathit{type}, \mathit{lev}, \mathit{weight}, \mathit{keys}, \mathit{elts}\rangle$ consisting of p's preorder ID in the index tree (assuming an arbitrary

[2]Note that besides rules for constraints of the form $R^*_1(e_0, e_1)$, no rules for constraints $R^j_i(e_0, e_1)$ are needed. These are rewritten into $R^*_1(e_0, e_1)$ earlier during evaluation after matching the proximity bounds i, j.

sibling order), the ID of p's parent, the greatest ID among p's descendants, p's label (i.e., the last node label in the corresponding label path), node type, level and BIRD weight (see Section 2.3), as well as the number of distinct keywords contained by p and the number of elements with path p. While the first seven fields are mandatory, the last two fields contain optional input to the query planner (see Section 4).

The tuples for all index nodes are stored in a *path table* replacing the index tree, which is no longer needed when evaluating queries with the RCADG. Fig. 2 *a.* shows the path table corresponding to the index tree in Fig. 1 *b.* The strings in the *lab* and *type* columns are given only for illustration; in fact this information is encoded numerically. Note that although the structure of the index tree could be restored from the *pid* and *par* fields alone, the *max* field is included to allow for checking $Parent_{\min}^{\max}$ constraints without unnecessary self-joins of the path table. The evaluation procedure described below exploits the fact that for any two index nodes $p, p' \in P$, $\langle p, p' \rangle \in Parent_1^*$ iff $p'.pid < p.pid \leq p'.max$. In fact, this is a variant of the *extended preorder* encoding [20]. Sect. 4.2 explains the benefit of this path representation.

To exploit relational index structures to the largest extent possible, the RCADG's element table is stored in first normal form (1NF), i.e., the lists of element IDs in the CADG's element table (Fig. 1 *c.*) are broken up into separate tuples (Fig. 2 *b.*). In 1NF, a B$^+$-Tree on the *eid* field can accelerate the evaluation of equality and range conditions on element IDs, as required by the BIRD rules in Figures 4 and 5.

4. QUERY EVALUATION WITH THE RCADG

Before explaining the use of the RCADG in detail, let us briefly summarize the specific requirements to be met, and anticipate how this work addresses them:

Exploit standard relational techniques. The RCADG consists of two tables with mostly numeric fields, favourable for efficient indexing. Path matching involves integer keys only, thus receiving excellent support by B$^+$-Trees even for wildcard-prefixed paths (unlike string-matching approaches like [34]). BIRD decision and reconstruction is easily expressed in standard SQL, unlike comparable tree encodings [26, 2]. Query planning is left to the RDBS to a large extent.

Avoid needless joins. Using BIRD to reconstruct binary query constraints saves many expensive joins with the large element table, especially compared to systems relying only on decision to restore the tree structure [20, 35, 11, 13].

Avoid joins involving large tables. The RCADG checks query constraints against the small path table first. The element table, usually orders of magnitude larger, is accessed only for label paths belonging to a match of the *entire query graph* on schema level. Other approaches scan the large element table with much weaker constraints (e.g., a single matching label or label path [20, 11, 34]).

Reduce intermediate results. Stricter conditions on joins with the element table also reduce the number of intermediate result tuples. Moreover, the RCADG provides statistical information for effective query planning (Sect. 4.3). Third, representing paths as index nodes with tree encoding (Sect. 4.2) and merging schema/data-level matches (Sect. 4.4) helps to discard many false hits early during the evaluation.

p1	*w1*	*p2*	*w2*	*p3*	*w3*	*p4*	*w4*	*p5*	*w5*
#1	9	#2	3	#3	3	#4	1	#4	1
#1	**9**	**#2**	**3**	**#3**	**3**	**#4**	**1**	**#5**	**1**

a. schema level

p1	*w1*	···	*p5*	*w5*	*e5*	*e4*	*e3*	*e1*
#1	**9**	···	**#5**	**1**	**&26**	**&25**	**&24**	**&18**

b. data level, step s_1

p1	*w1*	···	*p5*	*w5*	*e5*	*e4*	*e3*	*e1*	*e2*
#1	**9**	···	**#5**	**1**	**&26**	**&25**	**&24**	**&18**	**&21**

c. data level, step s_2

e2	*e4*
&21	**&25**

d. final result

Fig. 6: Query results for Fig. 3: *a.–c.* intermediate; *d.* final.

4.1 Translating conjunctive XML queries to SQL

```
1  proc evaluate (Q: query)
2    call translateSchema (Q)
3    call matchSchema (Q)
4    S := call createPlan (Q)
5    for all steps s ∈ S do
6      call translateData (Q)
7      call matchData (s)
8    end for
9    call createResult (Q)
10 end proc
```

Fig. 7: Query evaluation.

RCADG query evaluation is divided into two phases. Phase 1 processes the query constraints on the *schema level* (lines 2–3 in Fig. 7), in a single SQL statement expressing a self-join of the path table (Sect. 4.2). This produces a first intermediate *result table* with IDs ($\mathtt{p}_k$) and weights ($\mathtt{w}_k$) of all candidate label paths that together form a matching to entire query graph (rows in Fig. 6 *a.*). Path IDs in column $\mathtt{p}_k$ match query node q_k. In the second phase (lines 4–8), the query is evaluated on the *data level*, based on the schema matching result from phase 1. Step by step (Fig. 6 *b.–c.*) occurrences $\mathtt{e}_k$ of candidate paths $\mathtt{p}_k$ in the result table (Fig. 6 *a.*) are retrieved from the element table and checked against the query constraints. Where applicable, reconstruction is used to avoid the expensive element table joins. First a *constraint-solving plan* is created (Sect. 4.3) specifying which binary constraints to reconstruct and in which order to decide the others. This also determines the number of joins with the element table. The first join retrieves occurrences of selected paths and stores them in another intermediate result table (Fig. 6 *b.*), provided they satisfy the query constraints. Constraint decision in phase 2 may cause candidates from phase 1 to be discarded, as seen in Fig. 6 *a.–b.* The table also contains IDs of reconstructed elements. Similarly, each subsequent step operates on the result table created in the previous step (Sect. 4.4). Selecting all matches to result query nodes in the last table (Fig. 6 *c.*) yields the final query result (Fig. 6 *d.*).

4.2 Constraint checking on the schema level

In phase 1 a $|Q_n|$-fold self-join of the path table (Fig. 8 *a.*) creates a first result table (Fig. 6 *a.*) for later joins with the element table (Sect. 4.4). For each query node the matching label paths are retrieved and projected onto their *pid* and *weight* fields. If needed for query planning (Sect. 4.3), the *keys* and *elts* fields may be added to the `SELECT` clause, too (omitted in Fig. 8 *a.*). The *schema matching rules* in Fig. 9 (in the example, S_{lab}, S_{type}, $\mathrm{S}_{Parent_1^1}$ and $\mathrm{S}_{Parent_1^*}$) translate unary (binary) constraints on q to match (join) conditions on q's label paths in the `WHERE` clause. (Note that $Child(q_0, q_1)$ constraints were replaced with the equivalent $Parent(q_1, q_0)$.) See [31] for the full set of schema-level rules.

Benefit of index node IDs. Representing label paths as index nodes with extended preorder IDs, we can decide whether two paths matching, say, q_2 and q_3 in Fig. 3 actually have a common prefix matching q_1. By contrast, approaches storing paths as strings [17, 27, 34] may enter phase 2 with many partial matches to the query graph, which drastically blows up intermediate results compared to the RCADG (Sect. 5.4).

```
CREATE TEMP TABLE s0 AS
SELECT pa1.pid AS p1,
 pa1.weight AS w1, ...,
 pa5.pid AS p5,
 pa5.weight AS w5
FROM PathTable pa1, ...,
 PathTable pa5
WHERE pa1.lab = 'person'
 AND pa1.type = 'element'
 AND ...
 AND pa4.lab = 'edu'
 AND pa4.type = 'element'
 AND pa5.type = 'element'
 -- decide Parent(q2,q1)
 AND pa1.pid = pa2.par
 -- decide Parent1*(q3,q1)
 AND pa1.pid < pa3.pid
 AND pa1.max >= pa3.pid
 -- decide Parent(q4,q3)
 AND pa3.pid = pa4.par
 -- decide Parent(q5,q3)
 AND pa3.pid = pa5.par
```

a. schema level

```
CREATE TEMP TABLE s1 AS
SELECT p1, w1, ..., p5, w5,
 elt5.eid AS e5,
 -- reconstruct PrevSib(q5,q4)
 elt5.eid - (1*w5) AS e4,
 -- reconstruct Parent(q5,q3)
 elt5.eid - (elt5.eid%w3) AS e3,
 -- reconstruct Parent1*(q5,q1)
 elt5.eid - (elt5.eid%w1) AS e1
FROM s0, ElementTable elt5
WHERE elt5.pid = p5
 AND elt5.key = 'female'
 -- decide Child(q3,q4)
 AND elt5.eid - (1*w5) >
  elt5.eid - (elt5.eid%w3)
 AND elt5.eid - (1*w5) <
  elt5.eid - (elt5.eid%w3) + w3
```

b. data level, step s_1 (Fig. 10)

```
CREATE TEMP TABLE s2 AS
SELECT p1, w1, ..., p5, w5,
 e5, e4, e3, e1, elt2.eid AS e2
FROM s1, ElementTable elt2
WHERE elt2.pid = p2
 AND elt2.key = ''
 -- decide Child(q1,q2)
 AND elt2.eid > e1
 AND elt2.eid < e1 + w1
```

c. data level, step s_2 (Fig. 10)

```
SELECT DISTINCT e2, e4
FROM s2
ORDER BY e2, e4
```

d. final result

Fig. 8: SQL code for evaluating the query in Fig. 3: *a.–d.* are executed in that order to produce the results in Fig. 6.

$$\frac{lev_i^i(q)}{p.lev = i}\ (\mathrm{S}_{lev_i^i}) \qquad \frac{l(q) \quad l \in \Lambda}{p.lab = l}\ (\mathrm{S}_{lab}) \qquad \frac{t(q) \quad t \in \mathcal{T}}{p.type = t}\ (\mathrm{S}_{type})$$

$$\frac{lev_i^j(q)}{p.lev \geq i \wedge p.lev \leq j}\ (\mathrm{S}_{lev_i^j}) \qquad \frac{Sib(q_0, q_1)}{p_1.par = p_0.par}\ (\mathrm{S}_{Sib})$$

$$\frac{Parent_1^1(q_0, q_1)}{p_1.pid = p_0.par}\ (\mathrm{S}_{Parent_1^1}) \qquad \frac{Parent_1^*(q_0, q_1)}{p_1.pid < p_0.pid \wedge \atop p_1.max \geq p_0.pid}\ (\mathrm{S}_{Parent_1^*})$$

Fig. 9: Selected schema-level matching rules (query nodes $q, q_0, q_1 \in Q_n$ matched by paths $p, p_0, p_1 \in P$, respectively).

4.3 Creating constraint-solving plans

In phase 2 the query constraints are checked against all elements with a candidate path from phase 1. A *constraint-solving plan* specifies which IDs to fetch from the element table and which to reconstruct from known IDs. The planning algorithm [31] exploits the full power of BIRD (Sect. 2.3). Due to lack of space, we only mention the planning goals: (1) to avoid element-table joins by reconstructing many IDs; (2) to reduce the number of rows in intermediate results by processing the most selective constraints first. Selectivity is estimated based on statistical information in the RCADG, e.g., the *elts* and *keys* fields of the path table (Sect. 3).

Note that matches to a query node q can be reconstructed only if there is a query constraint $R(q', q) \in Q_c$, where relation R has a suitable BIRD reconstruction rule, for some $q' \in Q_n$ whose matches are already known from prior matching. Thus the order in which query nodes are matched influences the number of possible reconstructions and, conversely, the number of joins required. For instance, if in Fig. 3 q_5 is matched first, then matches to q_3, q_1 and q_4 can be reconstructed using rules $\mathrm{D}^{Rec}_{Parent_1^*}$ and $\mathrm{D}^{Rec}_{PrevSib_i^i}$ (Fig. 5). Yet conflicts between the two planning goals may arise, e.g. when selectivity estimates recommend matching q_1 first (in line with goal 2), which requires another element-table join (contrary to goal 1). This issue is considered future work.

```
plan $S = \langle s_1, s_2 \rangle$
step $s_1$:
 $Join_1 = \{q_5\}$
 $Rec_1 =$
   $\{PrevSib(q_5, q_4),$
    $Parent(q_5, q_3),$
    $Parent_1^*(q_5, q_1)\}$
 $Dec_1 =$
   $\{Child(q_3, q_4)\}$
step $s_2$:
 $Join_2 = \{q_2\}$
 $Rec_2 = \{\}$
 $Dec_2 =$
   $\{Child(q_1, q_2)\}$
```

Fig. 10: Constraint-solving plan for the query in Figure 3.

Formally, a constraint-solving plan S is a sequence of *steps*, i.e., triples $s_i = \langle Join_i, Rec_i, Dec_i \rangle$ where $Join_i$ is a set of query nodes to be matched in joins with the element table, and Rec_i and Dec_i are sets of binary constraints to be reconstructed and decided, respectively. Fig. 10 shows the first of the plans just sketched: step s_1 matches q_5 in Fig. 3 and reconstructs q_1, q_3, q_4; step s_2 matches q_2 in a second join and decides the remaining binary constraint $Child(q_1, q_2)$. The SQL code in Fig. 8 *b.–c.* is based on this query plan.

4.4 Constraint checking on the data level

$$\frac{con_k(e)}{e.key = k}\ (\mathrm{D}_{con})$$

Fig. 11: Data-level text matching rule ($e \in E, k \in K$).

In the rest of phase 2, each step s_i of the constraint-solving plan S (Fig. 10) is translated into a separate SQL statement, as described in the following for $i = 1$ (Fig. 8 *b.*). A new result table (`s1`) is created by joining the previous one (`s0`) with the element table once for each query node $q \in Join_i$ (q_5 in Fig. 8 *b.*). The join condition `elt5.pid=p5` in the `WHERE` clause retrieves only elements whose label paths match q_5. There are also conditions for matching q_5's keyword constraint, created by rule D_{con} (Fig. 11), and for deciding the constraint $Child(q_3, q_4) \in Dec_i$ using the BIRD rule $\mathrm{D}^{Dec}_{Child_1^*}$ (Fig. 4). Here matches to q_3, q_4 are reconstructed using BIRD rules in Fig. 5. Reconstruction of the constraints in Rec_i also appears in the `SELECT` clause, along with the IDs of matches to nodes in $Join_i$ (`e5`) and all fields from former result tables (`p1`–`w5`). Step s_2 is treated alike (Fig. 8 *c.*). Note that an empty keyword constraint is added by default.

Combined schema/data-level matches. The result tables in Fig. 6 *b.–c.* successively combine paths $\mathtt{p}_k$ in schema-level matches with their occurrences $\mathtt{e}_k$ forming data-level matches. Thus when a single path fails in data-level matching, all schema-level matches it belongs to are discarded: in Fig. 6 *b.* the schema-level match ⟨#1, #2, #3, #4, #4⟩ (shaded in Fig. 6 *a.*) is dropped while matching q_5 since no element with path #4 contains *"female"*. Paths appearing only in discarded matches may never be looked up in the element table, which reduces the number of elements to be processed.

5. EXPERIMENTAL EVALUATION

name	XML size	$\|E\|$	$\|P\|$	lev
IMDb	8,633 MB	83,404,825	276	5
XMark	1,145 MB	20,532,979	549	13
INEX	536 MB	12,049,113	10,203	17
DBLP	157 MB	5,390,160	129	7

Tab. 1: Document collections.

We implemented the RCADG as explained in Sect. 3 and built on top of it a retrieval engine for conjunctive XML queries. This test system was compared to a native XML engine [24] based on the CADG, as well as two relational storage schemes with and without a path index [34, 11]. We ran a number of manually created queries against the document collections in Tab. 1. *IMDb* [14] comprises more than 8 GB of web documents about movies and actors. The *XMark* benchmark [33] consists of 1 GB synthetically generated, recursive XML. The highly heterogeneous *INEX* benchmark [15] contains scientific articles and *DBLP* [5] bibliographic data from computer science. The key results of our experiments are:

Corpus	QID	results	XPath query	RC
IMDb	I3	6507	//*[title="love"]/production_year	1.27
	I4	118,150	//movie[.//genre="documentary"]//actor	8.77
XMark	X4	2	/site/open_auctions/open_auction[bidder[personref/ @person="person20"]/following-sibling::bidder[personref/ @person="person17290"]]/reserve	0.44
	X15	1890	/site/closed_auctions/closed_auction/annotation/description/ parlist/listitem/parlist/listitem/text/emph/keyword	0.52
	X14	9461	/site//item[contains(description,"gold")]/name	3.34
	X13	22,000	/site/regions/australia/item[name and description]	0.88
	X2	597,777	/site/open_auctions/open_auction/bidder/increase	17.54

↑ Tab. 2: Performance [s] of the RCADG (RC). Only matches to XPath result nodes were computed (unlike Table 3).

→ Tab. 3: Performance [s] of the RCADG (RC), native X^2 and XPath Accelerator (XA). All query nodes were matched.

Corpus	QID	results	XPath query	RC	X^2	XA
IMDb	I1a	12	//person[name="mastroianni" and born/@place]/biography/movie	0.10	12.72	0.04
	I1b	3	//person[name="felix" and ... (cf. I1a)	0.11	12.60	0.50
	I1c	24	//person[name="cooper" and ... (cf. I1a)	0.15	12.84	215.83
	I1d	72	//person[name="steve" and ... (cf. I1a)	0.52	12.85	> 600
	I2	6507	//title[.="love"]	0.37	0.30	> 600
	I3	6507	//*[title="love"]/production_year	1.52	26.07	> 600
	I4	118,150	//movie[.//genre="documentary"]//actor	34.68	↯	> 600
XMark	X1	1	/site/people/person[@id="person0"]/name	0.09	6.85	0.02
	X21	13	//site//europe//item[.//description// keyword[.="abandon" and .//bold] and .//name and (.//category or .//*[@category]) and .//mail[.//date and .//from and .//to]]	0.32	21.80	> 600
	X13	22,000	/site/regions/australia/ item[name and description]	2.35	2.79	61.46
	X2	597,777	/site/open_auctions/open_auction/ bidder/increase	122.43	↯	292.14

1. Querying XML in an RDBS benefits greatly from native XML indexing techniques. The RCADG is 2–3 orders of magnitude faster than the native and relational baseline systems. Complex queries with large results causing them to break down are answered in seconds by the RCADG.
2. The RCADG easily scales up to the multi-gigabyte level in terms of both speed and storage. Its path table is typically several orders of magnitude smaller than the original data.
3. Constraint-solving plans largely influence the evaluation speed. Inapt planning may spoil the performance gain.

Tab. 2 lists the RCADG performance results (averaged after removing the best and worst of five runs). Queries are given as their closest XPath equivalents. *XMark* queries Xk with $k \leq 20$ capture the XPath portion from the XQuery benchmark [33]. As Tab. 2 shows, the RCADG scales well with both the size of the data and the number of query results.

5.1 Experimental setup

We implemented the RCADG and evaluation procedure as described above. The path and element tables are indexed using B^+-Trees as follows. The path table has a cluster index on the $\langle pid, max, lab \rangle$ fields and a secondary index on the $\langle lab \rangle$ field. The element table has a cluster index on the $\langle pid, key, eid \rangle$ fields and a secondary index on the $\langle key, eid \rangle$ fields. Our native XML system is X^2 [24] (Sect. 5.2), combined with the CADG and BIRD. During query evaluation, X^2 fetches sets of elements from the relational back-end and combines them into query matches in memory. As relational baseline we implemented the *XPath Accelerator* storage scheme [11] (Sect. 5.3), which translates conjunctive XML queries into $|Q_n|$-fold joins of the element table where Q_n is the number of query nodes. Each element is identified by its pre- and postorder ranks in the document tree, which also serve to decide binary query constraints. Reconstruction is not supported, and no schema-level index is available. The element table is indexed using B^+-Trees as described in [11]. We also apply the *shrink-wrapping* optimization proposed in [11]. All three retrieval systems are implemented in Java (J2SDK 1.4.2) and access the relational database backend via JDBC. We also compare the performance of SQL code generated for the RCADG and *XRel* [34], a relational storage scheme with a string-based path index (Sect. 5.4).

Our RDBS is PostgreSQL v. 7.3.2, running on the same machine as the respective retrieval engine, with database cache disabled. Apart from the database server and the clients, the computer was idle. All tests were carried out sequentially on an i686 computer with an AMD Athlon XP 2600+ CPU running at 2 GHz with 256 kB cache. The machine has 1 GB RAM and runs Slackware Linux 1.9.

5.2 RCADG vs. native X^2

A first set of experiments (Table 3) measures the performance gain of the RCADG (RC) over native XML retrieval with X^2 [24]. To avoid a handicap for the X^2 system which always matches the entire query graph, all query nodes are treated as result nodes here. For the RCADG, the runtime performance therefore differs from the results in Table 2.

Queries I1a–I1d retrieve the movies and place of birth of different people mentioned in the movie database *IMDb*. Note how the performance of both RCADG and X^2 remains stable as the selectivity of the query keyword decreases: while "mastroianni" is contained only in 406 elements, the frequency of "felix" is almost ten times higher; "cooper" occurs in 10,398 elements and "steve" in 38,983. The RCADG's performance gain is two orders of magnitude for the most selective keyword and still more than a factor 20 for the most frequent keyword. As queries I2–I3 show, this is due mainly to "output" nodes like @place and movie, i.e., leaves without text constraints: while X^2 is highly competitive for I2, the decision caused by production_year in I3 slows down the system by two orders of magnitude. Query I4 illustrates another potential weak-spot in native retrieval systems: processing very large intermediate result sets in memory easily exceeds the hardware capacities. During the evaluation of I4 X^2 quickly runs out of memory; allocating more memory avoids a crash but results in swapping.

Query X21 against *XMark* examines how the systems cope with queries whose complexity is in the graph structure, not the result size. While the RCADG spends 85% of the time in generating extremely efficient SQL code for reconstructing four $Parent_1^*$ constraints, X^2 is again trapped in too many decision operations. The result for X1 (X2) confirms the earlier observations for I3 (I4). Note that when returning only matches to XPath result nodes, the RCADG answers the same queries again up to 7 times faster (see Tab. 2), retrieving more than half a million hits in less than 20 s.

5.3 RCADG vs. relational XPath Accelerator

We now compare the RCADG to the relational XPath Accelerator (XA) [11]. As mentioned before, XA decides all binary constraints via self-joins on the element table, lacking reconstruction and a path table. Consequently, in our test with different keyword selectivities (I1a–I1d in Tab. 3), XA's evaluation time rapidly grows with the size of intermediate results, reaching 820 s for I1d. Unselective queries like I2–I4 also take longer than ten minutes to evaluate. Only for highly selective queries (I1a, X1) XA is slightly faster than the RCADG, possibly because the latter issues multiple SQL queries. XPath Accelerator suffers much more from com-

Corpus	QID	query	results RC	results XR	RC	XR
INEX	N1a	//p	609	609	< 0.01	0.09
	N1b	//p[sub]/b	27	3,485,916	0.01	515.22

↑ Tab. 4: Performance [s] of the RCADG (RC) and XRel (XR) for schema-level matching only (phase 1).

→ Tab. 5: Performance [s] of the RCADG (RC) and XRel (XR). Only XPath result nodes were matched.

Corpus	QID	XPath query	results RC	results XR	RC	XR
XMark	X1	/site/people/person[@id="person0"]/name	1	1	0.09	3.96
	X22	//parlist[.//text[.="zenelophon"]]/listitem/text	133	↯ 183	0.14	27.95
	X14	/site//item[contains(description,"gold")]/name	9461	9461	3.34	> 600
	X13	/site/regions/australia/item[name and description]	22,000	22,000	0.88	> 600
	X23	//regions[contains(.,"zyda@ask")]//keyword	416,175	416,175	32.21	310.03
	X2	/site/open_auctions/open_auction/bidder/increase	597,777	597,777	17.54	6.12
DBLP	D1a	//article[author="codd"]/title	34	34	0.12	9.18
	D1b	/dblp/article[author="codd"]/title	34	34	0.12	9.14

plex query graphs like X21 than the schema-aware RCADG and X^2, its join conditions on the element table capturing only singleton labels rather than label paths. Query X2 is reported as critical in [12], too. Note that when retrieving only increase elements, the RCADG outperforms XA by more than one order of magnitude (18 vs. 220 s). As I4 shows, the latter does not scale well to unselective queries with // steps, probably due to the resulting range conditions in self-joins. Here the RCADG is two orders of magnitude faster. Summing up, the experiments prove that the native XML indexing techniques underlying the RCADG entail a decisive performance gain in the relational domain.

5.4 RCADG vs. path-based relational XRel

The XRel scheme [34] indexes entire label paths as strings, unlike the RCADG's compositional paths (Sect. 4.2). This has a number of disadvantages. First, string matching is slower than ID comparison, especially for query paths starting with //. Tab. 4 compares how fast RCADG and XRel match a query graph on schema level (phase 1) and how many matches they retain for phase 2. For N1a both retrieve 609 matches, but the RCADG is slightly faster. Second, XRel produces many partial matches to be discarded in phase 2: for N1b its intermediate result is five orders of magnitude larger than that of the RCADG. This also slows down the subsequent data-level matching, as shown in Tab. 5. On *DBLP* the RCADG is almost two orders of magnitude faster (D1a), even for absolute paths (D1b). On *XMark*, the difference is between one and three orders of magnitude. XRel outperforms the RCADG only for a single unselective query path without // steps (X2). Third, XRel yields *wrong final results* for branching // queries on recursive collections like *XMark*. For X22, e.g., XRel retrieves 50 false hits (see [31] for details). Further limitations are discussed in Sect. 6.

5.5 Storage requirements

Maintaining label path like the RCADG or CADG is cheap in terms of storage. In our tests, the path table occupies only between 48 kB (*DBLP*) and 120 kB (*XMark*) on disk, including relational indices (Sect. 5.1). The CADG index tree in memory occupies 2 MB in both cases. Only for the heterogeneous *INEX* corpus the path table is nearly 2 MB and the index tree 25 MB. The RCADG's element table grows to 17 GB for *XMark* and 34 GB for *IMDb* (again including all relational indices). In particular, the materialized join of elements and keywords causes considerable redundancy, which on the other hand speeds up query evaluation. Note that in our experiments, both XPath Accelerator tables together are only a little smaller than the RCADG (element table: *XMark* 3 GB, *IMDb* 12 GB; content table: *XMark* 11 GB, *IMDb* 19 GB). The CADG's element table in non-first normal form is considerably smaller (*XMark*: 4.5 GB, *IMDb*: 5.2 GB). The RCADG uses 1NF to speed up the evaluation of BIRD rules (see Section 3). Summing up, the tests show that the RCADG scales up well to the multi-GB level.

6. DISCUSSION AND RELATED WORK

Some of the original native XML indexing structures [8, 22, 25] are compared to the CADG in [30]. Tree encodings are surveyed in [18]. A thorough analysis of different encodings compared to BIRD is found in [32]. A recent survey [19] provides a comprehensive overview and classification of a large number of papers dealing with XML and RDBSs. Relational storage schemes for XML are divided into *schema-based* [6, 28, 1, 3] and *schema-oblivious* [11, 17, 27, 34, 29, 13] approaches. The latter store any kind of XML in the same set of tables regardless of the structure of the data (although it may be captured in a path index, as by the RCADG). Among schema-oblivious storage schemes, the *edge-oriented* [17] ones materialize the $Parent_1^1$ relation in the document tree [34, 27, 17], like the well-known Edge Approach [7]. This entails an equijoin for each $Parent_1^1$ step in a query path, and recursive SQL queries for matching the $Parent_1^*$ relation and restoring entire document subtrees [19]. By contrast, *node-oriented* [17] schemes [12, 11, 13] avoid recursive queries for these tasks, but replace equijoins with range joins. Neither kind of scheme supports reconstruction of element IDs. Both edge- and node-oriented approaches have been combined with a *path table* [17, 27, 34] corresponding to the DataGuide [8] or 1-Index [25] of the document tree. [29] states that "relational optimizers need to understand the hierarchical structure of XML". The RCADG further enhances the use of path materialization by storing statistical information which enables query planning and optimization based on properties of the document tree (not merely a set of nodes in the element table).

The key contribution of our approach compared to previous work with path tables [17, 27, 34] is the *compositional path representation*: the RCADG is one of very few relational storage schemes [4] to represent a label path not as a string, but as a sequence of index nodes (i.e., tuples in the path table) with IDs encoding part of the tree structure. This approach has several advantages:

- Path matching receives excellent support by B^+-Trees on numerical path IDs, which is more efficient than string matching especially for // paths. Matching query paths via self-joins of the small path table is cheap and happens entirely in the realm of the relational query optimizer.
- The *entire* query structure is matched against the path table before accessing the element table. This saves much needless join effort caused by unrecognized partial matches. Interleaved matching on the schema and data levels further reduces the size of intermediate results (Sect. 4.4).
- Unlike [34], the RCADG handles existential XPath predicates against recursive corpora correctly without a massive join overhead, which is considered an open problem in [19].
- The RCADG evaluates queries involving any XPath axis and //* steps, which are not supported by string-based approaches like [34, 17]. Furthermore, /* steps do not entail extra self-joins of the element table, as in [34].

7. CONCLUSION

This paper introduced the RCADG as an embedding of native XML indexing techniques into a relational database system (RDBS). As such the RCADG benefits both from the tree-awareness of its data structures (CADG tree, BIRD encoding) and from established and highly developed RDBS techniques. Compared to the native XML system X^2 and the relational systems XPath Accelerator and XRel, the RCADG achieved performance gains of up to three orders of magnitude. By integrating native XML indexing techniques even deeper into the core of an RDBS, as suggested in [19, 29], we expect further improvements. In particular, we outlined how the RCADG can provide XML-specific statistics to the relational query planner and optimizer to make them aware of the hierarchical structure of the data.

8. REFERENCES

[1] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: a Cost-based Approach to XML Storage. In *Proc. 18th ICDE Conf.*, pages 64–75, 2002.

[2] J.-M. Bremer and M. Gertz. An Efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, CS Dept., Univ. of Calif. at Davis, 2003.

[3] Y. Chen, S. Davidson, C. Hara, and Y. Zheng. RRXS: Redundancy Reducing XML Storage in Relations. In *Proc. 29th VLDB Conf.*, 2003.

[4] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS : An Efficient XPath Processing System. In *Proc. 23rd SIGMOD Conf.*, pages 47–58, 2004.

[5] Dig. Bibliogr. & Library Project. `dblp.uni-trier.de`.

[6] A. Deutsch, M. Fernández, and D. Suciu. Storing Semistructured Data with STORED. In *Proc. 18th SIGMOD Conf.*, pages 431–442, 1999.

[7] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[8] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23rd VLDB Conf.*, pages 436–445, 1997.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. 28th VLDB Conf.*, pages 95–106, 2002.

[10] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. In *Proc. 23rd ACM Symposium on Principles of Database Systems*, pages 547–556, 2001.

[11] T. Grust. Accelerating XPath location steps. In *Proc. 21st SIGMOD Conf.*, pages 109–120, 2002.

[12] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. 30th VLDB Conf.*, 2004.

[13] P. J. Harding, Q. Li, and B. Moon. XISS/R: XML Indexing and Storage System Using RDBMS. In *Proc. 29th VLDB Conf.*, 2003. Demo.

[14] IMDb. The Internet Movie Database. `www.imdb.org`.

[15] Initiative for the Evaluation of XML Retrieval 2004.

[16] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Join. In *Proc. 29th ICDE Conf.*, pages 253–263, 2003.

[17] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Path Materialization Revisited: An Efficient Storage Model for XML Data. In *Australasian Database Conf.*, 2002.

[18] H. Kaplan, T. Milo, and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In *Proc. 13th Symp. on Discrete Algorithms*, pages 271–281, 2002.

[19] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-to-SQL Query Translation Literature: The State Art and Open Problems. In *Proc. 1st Int. XML Database Symposium (XSym)*, pages 1–18, 2003.

[20] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. 27th VLDB Conf.*, pages 361–370, 2001.

[21] D. Barbosa et al. ToX – the Toronto XML Engine. In *Workshop Inf. Integr. on the Web*, pages 66–73, 2001.

[22] B. Cooper et al. A Fast Index for Semistructured Data. In *Proc. 27th VLDB Conf.*, pages 341–350, 2001.

[23] R. Goldman et al. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proc. 2nd Int. Workshop on the Web and Databases*, pages 25–30, 1999.

[24] H. Meuss, K. U. Schulz, F. Weigel, S. Leonardi, and F. Bry. Visual Exploration and Retrieval of XML Document Collections with the Generic System X^2. *Journal of Digital Libraries*, 5(1):1–70, 2005.

[25] T. Milo and D. Suciu. Index Struct. for Path Expressions. In *Proc. 7th ICDT Conf.*, pages 277–295, 1999.

[26] P. O'Neil, E. O'Neil, and S. Pal et al. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. 23rd SIGMOD Conf.*, pages 903–908, 2004.

[27] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. *Proc. 3th Int. Workshop on the Web and Databases*, pages 47–52, 2000.

[28] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. 25th VLDB Conf.*, 1999.

[29] I. Tatarinov, S. Viglas, K. S. Beyer, and J. S. sundaram et al. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. 21st SIGMOD Conf.*, pages 204–215, 2002.

[30] F. Weigel, H. Meuss, F. Bry, and K. U. Schulz. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Proc. 26th Europ. Conf. on Information Retrieval*, pages 378–393, 2004.

[31] F. Weigel, K. U. Schulz, and H. Meuss. Exploiting Native XML Indexing Techniques for XML Retrieval in RDBSs. Technical report, 2005. `www.cis.uni-muenchen.de/~weigel/Literatur/weigel05rcadgtech.pdf`.

[32] F. Weigel, K. U. Schulz, and H. Meuss. The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations. In *Proc. 3rd Int. XML Database Symposium (XSym)*, 2005.

[33] XMark. XML Benchmark. `www.xml-benchmark.org`.

[34] M. Yoshikawa and T. Amagasa et al. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *Transactions on Internet Technology*, 1(1):110–141, 2001.

[35] C. Zhang, J. F. Naughton, and D. J. DeWitt et al. On Supporting Containment Queries in RDBMS. In *Proc. 20th SIGMOD Conf.*, pages 425–436, 2001.