Caching Schema Information and Intermediate Results for Fast Incremental XML Query Processing in RDBSs

Felix Weigel[1] Klaus U. Schulz[1]

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

ABSTRACT

Many index structures and algorithms have been proposed for efficient query processing in XML databases. However, experience with view-based query answering in RDBSs shows that the incremental evaluation based on cached query results can substantially improve the performance compared to the evaluation from scratch. The main problems related to caching are: (1) to determine which cache entries contain the desired data, and (2) to choose those from which the data can be computed with the least execution cost. Most approaches to XML query caching have addressed the first issue *intensionally* by comparing query expressions, which is exponential while tending to ignore valuable cache contents, and have ignored the second. This paper presents a new XML query cache which not only contains the final results of prior queries, but also intermediate "snapshots" of the data obtained throughout the evaluation process, as well as a compact summary of their structure (schema). Where the final result of a restrictive query in the cache may be useless, intermediate (partial) matches are much more likely to overlap with subsequent queries. Schema information allows to compare queries *extensionally* without accessing the actual results in the cache and to retrieve the relevant part of the cache contents, even where a purely intensional approach fails. We also explain query planning and cost estimation for the evaluation from cache and from scratch in an RDBS. Extensive experiments and a cost/benefit analysis illustrate the efficiency, effectiveness and scalability of our approach.

1. INTRODUCTION

As more and more large collections of XML documents become available, XML query processing is receiving much attention and many index structures and algorithms have been proposed for efficient search in XML databases with either a native [1, 2, 3, 4] or a relational data model [5, 6, 7, 8, 9, 10]. However, these approaches only cover the evaluation "from scratch", disregarding previously computed query results which may contain some or even all hits to a new query being processed. Experience with view-based query answering in RDBSs [11] shows that the *incremental* evaluation based on cached query results can substantially improve the performance compared to the evaluation from scratch. The main problems related to caching are similar both for relational and XML data: (1) to determine which cache entries contain (part of) the desired data, and (2) to choose those cache entries from which the final result can be obtained with the smallest computational and I/O effort. Yet for accelerating XML search it is not enough to apply techniques developed for view-based query answering in RDBSs to XML data stored as tuples. An explicit representation of its hierarchical structure is needed to decide if and how some cached query results can contribute to answering a given new query Q^n (except in the trivial case where the same query is asked repeatedly). A truly incremental evaluation exploits *containing* or, more generally, *overlapping* queries, i.e., those which may not be exactly equal to Q^n, but whose results sets in the cache nevertheless include all (containment) or, for overlap, at least some or some parts of the requested data. Results of containing queries must be purged of false positives w.r.t. Q^n. Exploiting overlapping queries is more challenging since they may also yield only a partial evaluation to be completed in following steps, perhaps by accessing the full data set. The result of an overlapping query Q may be incomplete in two ways: not necessarily all parts of Q^n are matched in Q, and also entire hits can be missing (which might be obtained from other cached queries, though). Unlike prior work addressing only *query containment*, where all desired data is subsumed by the result of a single cached query, we consider the more general *overlap problem* because (a) completing partial and retrieving missing matches is usually still faster than answering Q^n from scratch; (b) the cached part of the result is quickly available while the evaluation of the missing part is going on in the background; (c) in top-k search those hits retrieved from cache may even suffice to fulfill the request.

Main idea and contributions

Schema information. A number of techniques for incremental XML querying have been proposed [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23], some with the label *semantic caching*. Most of them address the query containment problem *intensionally*[1], i.e., by comparing only the query expressions, which requires exponential time for tree- and graph-shaped conjunctive regular path queries (including XPath queries) [13]. However, a purely intensional query comparison fails to recognize containing queries in the cache when they have additional constraints (say, an extra node) not mirrored in the new query. Moreover, from intensions alone one cannot decide whether a non-containing cached query

[1]This work was supported by Deutsche Forschungsgemeinschaft (DFG). The authors thank the DFG for the support.

[1]By *intension* we mean an abstract description of data (e.g., query results) in terms of desired properties (like structure and keywords specified by a query), while *extension* denotes some representation of the data with such properties.

overlaps with Q^n, nor extract its relevant results from the cache. (Sect. 3 gives an example.) To handle such queries, we need to know the actual data, i.e., compare the query *extensions* (results), which is of course infeasible if it amounts to an evaluation from scratch. Yet we show that *schema* information summarizing the document structure allows to detect various cases of containment and overlap missed by the purely intensional approaches, and to retrieve all cache contents needed to compute the final query result. From the DataGuide [1], a compact representation of all distinct label paths in the documents, we derive *schema hits*, i.e., concise structural views on query extensions. Each of the schema hits for a query Q represents a subset of hits to Q with a specific structure (its *matches*). In [10], we have shown how to use schema hits for accelerating XML query processing from scratch in an RDBS. In this paper, we use the same schema information for efficiently detecting query containment and overlap through a combined intensional and extensional comparison, exploiting the fact that *even when two queries cannot be compared directly, their schema hits can.* The schema hits of a cached query Q are held in a main-memory index structure for fast cache search without access to the actual query results residing on disk. Comparing the schema hits of Q^n and Q may reveal that while the matches to some schema hits to Q^n must be computed from scratch, others are (perhaps partially) contained in the match set of a schema hit to Q; in other words, Q overlaps with Q^n. Similarly, if the matches to all schema hits of Q^n are fully contained in Q's match sets, then Q contains Q^n. We present an integrated evaluation process for retrieving part of the final result of Q^n from one or more cached queries and the rest from scratch. Fig. 1 sketches how both threads share common steps, e.g., schema matching or query planning.

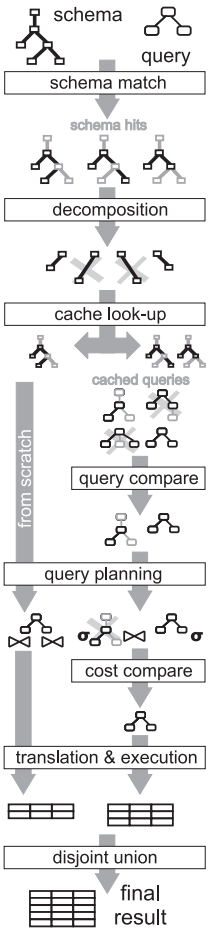


Fig. 1: Combined evaluation with and without cache.

Intermediate results. A second contribution is motivated by the observation that even when the final result to a cached query Q is too restricted for Q^n, a partial evaluation of Q may yield partial matches to Q^n. Therefore we also store and retrieve intermediate results obtained during the evaluation of cached queries. To the best of our knowledge, all existing XML caches cover only the final results of earlier queries. However, queries with branching nodes are typically evaluated in a sequence of steps (e.g., path- or edge-wise) each producing a set of partial matches to the query. While the final result of a restrictive query in the cache may be useless for many new queries, the intermediate matches are much more likely to overlap with subsequent queries (for an example see Sect. 3). Therefore we store multiple "snapshots" of a query result as it evolved during the evaluation of a cached query. We propose special techniques to avoid a cache blow-up in space. The information which snapshots (i.e., intermediate or final results) are available for a given query is derived from the query plan used to obtain them, and annotates the schema hits in the main-memory part of the cache. Thus alternative plans for answering the current query based on the cache are created and compared in terms of join costs, in order to exploit the most useful cache contents.

Methodology and evaluation. While the benefit of incremental query answering is obvious – faster processing due to fewer computational and I/O operations – the cost is often ignored. The third contribution therefore concerns the evaluation methodology: we identify different cost measures and optimization goals for XML query caching, and locate our approach in the resulting trade-off both through a cost/benefit analysis and an experimental evaluation. The notion of *cache support* is introduced which quantifies how useful the available cache contents are for a given query. In two different experiments we study how cost and benefit change as a function of the cache support. A small-scale test simulates typical query editing operations, say, in a user session with relevance feedback. Then, in a large-scale experiment a randomly generated query workload is evaluated against different stages of an evolving synthetic query cache. We find that given a sufficient cache support, the benefit of incremental processing often outweighs by far the cost, with performance gains of more than one order of magnitude. When the cache does not contain useful results, the overhead caused by the unsuccessful cache search increases with the number of schema hits to be checked, not the overall cache size, thanks to our index structure.

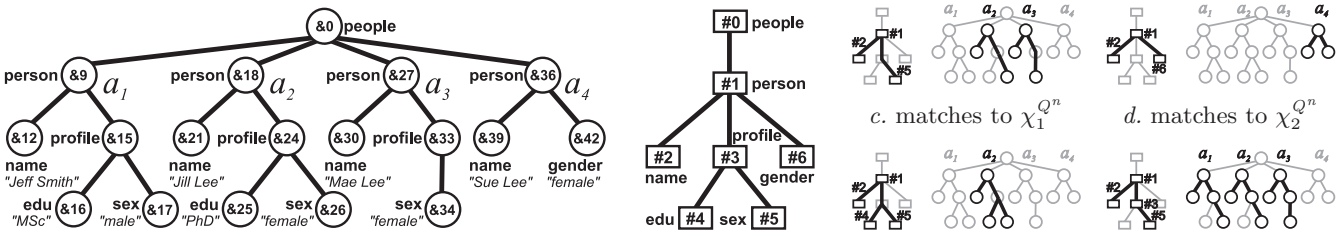
The paper is organized as follows. First some preliminaries are defined, including the data and query model. Section 3 gives an overview of our approach and introduces the example used throughout the paper. Section 4 briefly explains how we process XML queries from scratch, which is needed for the detailed description of our incremental technique in Section 5. Section 6 reports on the experimental results and discusses a couple of optimizations. Finally, Section 7 reviews related work before we conclude in Section 8.

2. PRELIMINARIES

Let Λ be a finite alphabet of node 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)[2], *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 immediate right sibling in the obvious way, 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 path from the root down to element e (i.e., where $\langle e_i, e_{i+1} \rangle \in \mathit{Child}$ for all $0 \leq i < k$). 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. Fig. 2 *a.* shows a heterogeneous document tree (node IDs are explained later).

The *schema tree* (*DataGuide* [1]) for D is the finite rooted

[2]For simplicity, we refer to document nodes as *elements* in the sequel. XML attributes are treated analogously.



a. document tree D *b.* schema tree S

Fig. 2: A sample document (*a.*) and schema (*b.*) and schema hits for the queries from Fig. 3 with their matches (*c.–f.*).

unordered node-labelled tree $\mathrm{S} = \langle P, \pi(r), \mathit{Child}', \lambda' \rangle$ whose nodes are the label paths in D and whose root is the label path of the root r in D. Fig. 2 *b.* shows the schema tree for D in *a.* The labelling function λ' maps a label path $p \in P$ to the last symbol $l \in \Lambda$ in p. For any two label paths p_1 and p_2, $\langle p_1, p_2 \rangle \in \mathit{Child}' \subset P \times P$ iff there exists a label $l \in \Lambda$ such that $p_2 = p_1 l$. Note that for elements $e_1, e_2 \in E$, $\mathit{Child}'(\pi(e_1), \pi(e_2))$ is necessary, but not sufficient for $\mathit{Child}(e_1, e_2)$. If $\mathit{Child}'(p_1, p_2)$, $\mathit{Sibling} \subset P \times P$ relates p_2 to all other children of p_1 (recall that S is unordered).

Besides *Child*, *NextSib* and their reverse counterparts (i.e., *Parent* and *PrevSib*, resp.), we also support binary relations *Following* and *NextElt* (document order) as well as their inverses (*Preceding* and *PrevElt*, resp.). This covers an important XPath fragment similar to *Core XPath* [24]. For all relations except *Following*, *Preceding* and *Sibling*, lower and upper proximity bounds are defined as follows: $R_i^j = \bigcup_{i \le k \le j} R^k$ where R^k denotes the k-fold composition $R \circ \cdots \circ R$ of R. Thus 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 remaining XPath axes are modelled in [10] using unary relations for the type, label and level of elements (with disjunction). To capture the textual contents of XML documents, we define unary relations $\mathit{govern}_k \subset E$ and $\mathit{contain}_k \subset \mathit{govern}_k$ for each keyword k in the set K of keywords occurring in the documents. For any $e \in E$, $e \in \mathit{contain}_k$ iff k occurs directly in e, and $e \in \mathit{govern}_k$ iff there is an e' containing k s.t. $\mathit{Child}_0^*(e, e')$. Keyword con-/disjunctions are treated in the obvious sense.

A *query* Q is a pair $\langle Q_v, Q_c \rangle$ where Q_v is a finite and non-empty set of query nodes and Q_c is a finite and non-empty set of constraints of the form $R_1(q)$ or $R_2(q, q')$ s.t. $q, q' \in Q_v$ and R_1, R_2 are unary and binary relations, respectively. Note that the resulting query graph $\langle Q_v, Q_c \rangle$, illustrated in Fig. 3 *a.–c.*, must be connected but not necessarily acyclic. A *matching* of a query Q against D is a mapping $\mu : Q_v \to E$ such that $\mu(q) \in R_1$ for each unary constraint $R_1(q) \in Q_c$, and analogously $\langle \mu(q), \mu(q') \rangle \in R_2$ for all binary constraints $R_2(q, q') \in Q_c$. The *answer* (*result*) for Q in D, $\mathit{ans}(Q)$, is the set of μ-images of Q_v (*matches*) for all matchings μ. The answer to Q^n in Fig. 3 *c.*, e.g., consists of the subtrees a_2, a_3, a_4 of D in Fig. 2 *a.*

As a structural summary of the documents [1], the schema tree only reflects some of the relations just introduced. The following key definitions separate query constraints that can be matched against the schema tree S from those which must be checked on the document level:

Definition 1. *The set of* S-constraints *to be matched against the schema tree comprises (1)* Parent'^j_i *and* Child'^j_i*, (2) Sibling, (3) label, type and level constraints* ($i \le j \in \mathbb{N}$).

Definition 2. *The set of* D-constraints *to be matched against the document tree comprises (1)* Parent^j_i *and* Child^j_i*, (2)* $\mathit{contain}_k, \mathit{govern}_k$*, (3)* $\mathit{NextSib}^j_i$*,* $\mathit{PrevSib}^j_i$*, (4)* $\mathit{NextElt}^j_i$*,* $\mathit{PrevElt}^j_i$*, (5) Following, Preceding* ($i \le j \in \mathbb{N}$, $k \in K$).

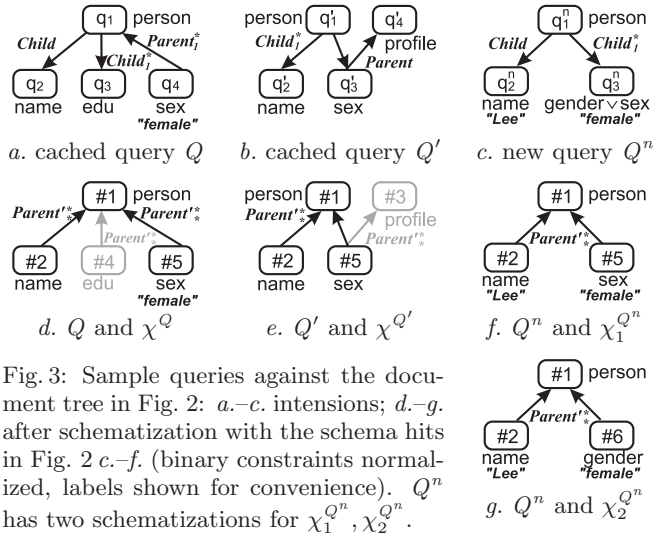
For matching S-constraints against the schema tree, we define $\mu_\mathrm{S} : Q_v \to P$ analogously to μ, and call the μ_S-images of Q its *schema hits*. The two schema hits $\chi_1^{Q^n}, \chi_2^{Q^n}$ for Q^n are shown in Fig. 2 *c.–d.* (left-hand side). Since each match $a \in \mathit{ans}(Q)$ corresponds to exactly one schema hit consisting of the label paths in a, a subset of schema hits for Q partitions $\mathit{ans}(Q)$ in the obvious sense (while other schema hits may have no instances in D). For a schema hit χ, $\mathit{ans}_Q(\chi)$ denotes the subset of $\mathit{ans}(Q)$ corresponding to χ (its *matches* for Q, see the right-hand side in Fig. 2 *c.–f.*). We drop the subscript to *ans* when Q is clear from the context. For instance, $\mathit{ans}(\chi_1^{Q^n}) = \{a_2, a_3\}$ and $\mathit{ans}(\chi_2^{Q^n}) = \{a_4\}$.

3. OVERVIEW AND EXAMPLE

3.1 Schema information in the cache

Schema information is useful for incremental query processing because it helps to solve the overlap problem (Sect. 1) for queries which are hard to compare on a purely intensional basis. For instance, consider the three queries in Fig.3 *a.–c.* which represent the intensional viewpoint. Assume that the final results of Q and Q' have already been retrieved and stored in the query cache (ignoring its exact structure for the moment). Obviously the third query Q^n cannot be proved to be contained in any of the cached queries from the intensions alone: the keyword constraints "Lee" and "female" are more restrictive and the label disjunction gender $\vee$ sex less restrictive than Q and Q'. Thus we cannot decide whether the three result sets overlap, nor retrieve exactly the intersection of Q^n with Q or Q', unless we compare the actual results in the documents. Since this would require Q^n to be evaluated from scratch, the cache contents seem useless for answering Q^n. However, below we show how to translate the intensions of all three queries to extensional constraints on the schema level, where they are compared in order to obtain part of the answer to Q^n (namely, a_2, a_3) from the cache at low computational and I/O cost.

In many situations the schema information is indispensable for exploiting cache contents. For a cached query Q^c and a new query Q^n whose intensions tell nothing about inclusion or overlap, schema hits may show whether Q^c nev-



a. cached query Q *b.* cached query Q' *c.* new query Q^n

d. Q and χ^Q *e.* Q' and $\chi^{Q'}$ *f.* Q^n and $\chi_1^{Q^n}$ *g.* Q^n and $\chi_2^{Q^n}$

Fig. 3: Sample queries against the document tree in Fig. 2: *a.–c.* intensions; *d.–g.* after schematization with the schema hits in Fig. 2 *c.–f.* (binary constraints normalized, labels shown for convenience). Q^n has two schematizations for $\chi_1^{Q^n}, \chi_2^{Q^n}$.

ertheless contains Q^n, or else which parts of $ans(Q^n)$ are missing in $ans(Q^c)$. Typical cases include the following:

- Q^n allows $Parent_1^*$ where Q required *Parent* (similar for *Child* and other proximity bounds)
- Q^n has a * where Q required a specific label/type/level
- Q^n accepts a superset of the labels specified in Q
- any combination of the above

In general *descriptive* schemata like DataGuides, as up-to-date summaries of the current document structure, allow to detect more of the reusable cache contents than *prescriptive* schemata like DTDs or XSDs, which are often too general.

Evaluating a query Q^n from scratch is done in two phases: during *schema matching*, we match its S-constraints against the schema tree, which produces the schema hits introduced above. Then during *document matching* we successively retrieve the occurrences of these schema hits while matching Q^n's D-constraints in an interleaved process. Details are given in Sect. 4; suffice it to say here that matching queries against the small schema tree can be done very fast, while document matching may involve expensive joins with the full data set. To rephrase the caching problem, we would like to reuse (maybe partial) occurrences of (at least some) schema hits from cached queries with constraints similar to Q^n, and match only the missing constraints from Q^n against them. S-constraints play an important role in efficiently finding suitable queries in the cache. In the sequel we assume that every query has at least one binary S-constraint (caching queries without *Parent* and *Child* edges is future work).

A simple example. Consider a cached query Q^c that looks like Q^n (Fig. 3 *c.*) without keyword constraints. Clearly Q^c and Q^n have identical S-constraints and hence the same two schema hits $\chi_1^{Q^c} = \chi_1^{Q^n}$ and $\chi_2^{Q^c} = \chi_2^{Q^n}$ (Fig. 2 *c.–d.*), but possibly different result sets. To decide whether Q^c overlaps with Q^n, we compare their D-constraints on each schema hit. In what we call *schematization*, all unary and binary D-constraints in a query are applied to those nodes of a particular schema hit which match the query nodes involved in these constraints. Fig. 3 depicts the schematization of Q^n with $\chi_1^{Q^n}$ (*f.*) and $\chi_2^{Q^n}$ (*g.*). For Q^c and $\chi_1^{Q^c}$, the keyword constraints are missing. *The schematization of D-constraints tells us which part of Q^n and Q^c must be reconciled:* there is overlap on $\chi_1^{Q^n}$ and $\chi_1^{Q^c}$ iff the schematized D-constraints in Q^c are no more restrictive than those in Q^n on the same schema nodes. For the binary constraints, this is trivial since they are equal, but the same would apply, say, to $NextElt_1^+(\#2, \#5)$ as a binary D-constraint in Q^c and $NextSib_1^+(\#2, \#5)$ or $PrevSib_1^+(\#5, \#2)$ in Q^n (now shown in Fig. 3). The test also succeeds for the unary constraints because the empty constraint attached to #2 in Q^c is less restrictive than "Lee" for #2 in Q^n, and likewise for #5. It would fail, e.g., if Q^c specified a single keyword other than "Lee" for #2 or a binary constraint not mirrored in Q^n.

In the example Q^c's D-constraints are *necessary conditions* for matches to Q^n and $\chi_1^{Q^n}$, i.e., $ans(\chi_1^{Q^n}) \subset ans(\chi_1^{Q^c})$. The *sufficient conditions* needed to retrieve exactly $ans(\chi_1^{Q^n})$ follow from the comparison of the schematized D-constraints and together form a *remainder query* [14] to be evaluated only on $ans(\chi_1^{Q^c})$—typically with substantially reduced CPU and I/O cost. Here the remainder query for Q^n against $ans(\chi_1^{Q^c})$ consists simply of the two keyword constraints on #2 and #5, as expected. The other schema hit $\chi_2^{Q^c}$ is treated similarly. To sum up, the schematization reveals which query constraints from Q^n and Q^c correspond and must be tested for restrictiveness. While for homomorphic queries this is trivial, the real benefit shows when Q^n and Q^c are different.

The general case. In general Q^n has a different structure from the cached queries, and therefore its schema hits are not exactly identical to hits in the cache. To recognize overlap, we also schematize and compare their S-constraints. *The schematization of S-constraints helps to locate cached queries that are potentially useful for evaluating Q^n.* First the S-constraints of any query being added to or looked up in the cache are *normalized*, as follows: (1) *Child′* edges are inverted to match equivalent *Parent′* edges; (2) label, type, level and proximities are omitted, being unambiguous for schema nodes. In case of $\chi_1^{Q^n}$ this yields $Parent'^*_*(\#2, \#1)$, $Parent'^*_*(\#5, \#1)$ (Fig. 3 *f.*) and for $\chi_2^{Q^n}$ $Parent'^*_*(\#2, \#1)$, $Parent'^*_*(\#6, \#1)$ (Fig. 3 *g.*). Looking up schema hits in the cache with these constraints, we retrieve χ^Q and $\chi^{Q'}$ in both cases (each sharing two constraints with $\chi_1^{Q^n}$ and one with $\chi_2^{Q^n}$, see Fig. 3 *d.–e.*). For each pair χ^{Q^c}, χ^{Q^n} of schema hits retrieved for a cached query Q^c and a new query Q^n in this way, $ans(\chi^{Q^n}) \subset ans(\chi^{Q^c})$ if (1) χ^{Q^c} is a subgraph of χ^{Q^n} and (2) the associated D-constraints in Q^c are less restrictive than the corresponding D-constraints in Q^n, which is checked as described above. Interestingly, additional constraints in Q^c that do not introduce a proper restriction may be ignored. For instance, the schematization of Q' yields $Parent'^*_*(\#5, \#3)$ which is missing in $\chi_1^{Q^n}$. However, it also applies implicitly to $\chi_1^{Q^n}$ (since the ancestors of #5 are unambiguous) and therefore does not hurt the containment $ans(\chi_1^{Q^n}) \subset ans(\chi^{Q'})$. By contrast, χ^Q is too restrictive for $\chi_1^{Q^n}$ due to the additional constraint $Parent'^*_*(\#4, \#1)$ (compare Fig. 2 *c.*, *e.* to verify that a_3 is part of $ans(\chi_1^{Q^n})$, but not $ans(\chi^Q)$, because of this edge). Likewise, since $\chi_2^{Q^n}$

pid	par	max	lab	type	lev
#0		#6	people	elt	0
#1	#0	#6	person	elt	1
#2	#1	#2	name	elt	2
#3	#1	#5	profile	elt	2
#4	#3	#4	edu	elt	3
#5	#3	#5	sex	elt	3
#6	#1	#6	gender	elt	2

a. path table

pid	key	eid
#0	""	&0
#2	""	&39
#2	"Smith"	&12
#2	"Lee"	&21
#2	"Lee"	&30
#2	"Lee"	&39

pid	key	eid
#5	""	&26
#5	""	&34
#5	"male"	&17
#5	"female"	&26
#5	"female"	&34
#6	""	&42
#6	"female"	&42

b. element table

Fig. 4: The RCADG for the document in Fig. 2 *a.* and *b.*

id	p1	p2	p3	p4
χ^Q	#1	#2	#4	#5

a. schema matching

id	...	p4	e4	e1
χ^Q	...	#5	26	18
χ'^Q	...	#5	34	27

b. doc. matching, step s_1^Q

id	...	p4	e4	e1	e2
χ^Q	...	#5	26	18	21
χ'^Q	...	#5	34	27	30

c. doc. matching, step s_2^Q

id	...	p4	e4	e1	e2	e3
χ^Q	...	#5	26	18	21	25

d. doc. matching, step s_3^Q

Fig. 5: Result tables created while evaluating Q in Fig. 3 *a.*

lacks the $Parent'^*_*(\#5, \#1)$ constraint in $\chi^Q, \chi^{Q'}$ its matches cannot be computed from cached results. In this way we examine all schematized constraints in a cached query that are not mirrored in Q^n to decide whether the query can still contribute matches to Q^n. By contrast, extra constraints in Q^n are simply added to the remainder query (see above).

Through schematization we learn that part of the answer to Q^n (namely, $ans(\chi_1^{Q^n})$) can be obtained from $ans(\chi^{Q'})$ (by matching "Lee" and "female") while the rest (namely, $ans(\chi_2^{Q^n})$) must be retrieved from scratch. Again, this distinction would be impossible on the intensional level and even if a DTD were given. The correctness of our approach follows from how we compare (1) schematized S-constraints during the cache look-up and (2) all additional constraints of cached queries afterwards. (Proofs are omitted for lack of space.) The algorithm is complete in the sense that it retrieves all cached schema hits whose schematized S- and D-constraints are more general than those of any schema hit to Q^n. Of course we cannot detect a coincidental containment or overlap of otherwise unrelated queries.

3.2 Intermediate query results in the cache

The above example assumes that only the final results of Q and Q' are stored in the cache. However, D-constraints are matched step-wise, not all at once. Caching the intermediate results can further increase the effectiveness of the cache when partial matches to a cached query happen to coincide with results to Q^n. As an example, assume the document matching for query Q in Fig. 3 *a.* was done in three phases: in the first two steps, s_1^Q and s_2^Q, $Parent^*_*(\#5, \#1)$, $contain_{\text{"female"}}(\#5)$ and $Parent^*_*(\#2, \#1)$ were matched, producing as intermediate result the two matches a_2, a_3. Only in the third step s_3^Q the edu node was matched, causing a_3 to be discarded from the final result of Q. Thus before s_3^Q, all matches to $\chi_1^{Q^n}$ (namely, a_2 and a_3) can be obtained from the intermediate result of Q in the cache. This makes Q a competitor of Q' in the contribution of cached query results for evaluating Q^n. Moreover, Q's matches already satisfy $contain_{\text{"female"}}(\#5)$ which also appears in Q^n, but not Q'. The query planner described in Sect. 5 therefore prefers Q to Q', thus saving an access to the document level.

Our evaluation algorithm processes queries with branching nodes in a sequence of steps in much the same way as sketched above and stores the intermediate results in an RDBS backend. In each step s_i one or more query constraints are matched, which makes some of the partial hits more complete while discarding others from the next intermediate result. Thus s_i produces a "snapshot" of the query result as it evolves during the evaluation process. To keep track of the "snapshots" available in the query cache, we tag each schematized D-constraint with the step in which it was matched during the evaluation of the cached query. This allows to determine the latest evaluation step in which a cached schema hit is useful for a schema hit of the new query. Let $[\![\chi]\!]_{s_i}$ denote the part of a schema hit χ which has been matched up to step s_i. In our example, $ans(\chi_1^{Q^n}) = ans([\![\chi^Q]\!]_{s_2^Q})$, hence the intermediate results to Q obtained in the step s_2^Q are used for answering Q^n.

4. QUERY EVALUATION FROM SCRATCH

This section briefly explains our approach to XML query evaluation from scratch in an RDBS, which is the basis for the incremental technique presented in Sect. 5. For details, see [10]. The core data structure is the *RCADG* index (for *Relational Content-Aware DataGuide*), which consists of two tables as shown in Fig. 4. The *path table* (*a.*) is a relational version of the schema tree S in Fig. 2 *b.*: each schema node is represented as a tuple $\langle pid, par, max, lab, type, lev\rangle$ where pid, par, max are the preorder ranks in S of the node itself, its parent and its last descendant (assuming an arbitrary sibling order), and $lab, type, lev$ are the node's label, type and level in S. The *element table* (*b.*) contains a tuple $\langle \pi(e), k, e\rangle$ for all elements $e \in E$ and keywords $k \in K$ s.t. e contains k as defined in Sect. 2. An additional entry for each element, regardless of its textual contents, and the empty keyword serves for matching query nodes without keyword constraints. Note that the path table is typically several orders of magnitude smaller than the element table.

Any given query is first matched on the schema level in an n-way selfjoin of the path table where n is the number of query nodes. While label, type and level constraints are simply translated to conditions on the last three fields in Fig. 4 *a.*, matching the binary S-constraints involves combined conditions on the pid, par and max fields exploiting certain properties of the preorder ranks in S. Proximity bounds translate to level conditions. The schema matching produces a table of schema hits such as the one shown in Fig. 5 *a.* for Q from Fig. 3 *a.* Based on these schema hits, the query is evaluated on the document level as follows. Step by step (Fig. 5 *b.*–*d.*) occurrences e_i of label paths p_i in Fig. 5 *a.* matching a query node q_i are retrieved through joins with the element table and checked against the D-constraints. Each step produces an intermediate result table with partial matches to be completed in subsequent steps. For instance, the two matches to q_4 in Fig. 5 *b.* are obtained by selecting those tuples from the element table where pid=#5 and key="female". Depending on the query constraints, however, there may be a better way to match query nodes which avoids the possibly expensive element-table join: we assign element IDs in such a way that the IDs of some neighbours of a given element e (e.g., its ancestors and left siblings) can be *reconstructed* from the ID of e without accessing the element table. Thus in Fig. 5 *b.*,

the matches to q_1 are computed from those to q_4 in memory without another element-table join. The underlying tree encoding is *BIRD* [25]. Other schemes supporting reconstruction could be used, too.

The order in which D-constraints are matched is determined by a *query plan* devised after schema matching. The plan P^Q for evaluating Q is given in Fig. 6. The full paper [26] explains query planning in detail. The goal is to avoid as many joins as possible through reconstruction. Only when a binary constraint cannot be reconstructed or we already know both elements to be checked, the edge is *decided* with a suitable condition on the BIRD IDs in a join with the element table. This is the case for the two edges matched in steps s_2^Q and s_3^Q. Note how the second partial match to Q in Fig. 5 *c.* is discarded from the final result in *d.* for not satisfying $Child_*^*(q_1, q_3)$ in s_3^Q. In the end, $join(P^Q) = 3$ joins are needed to evaluate Q from scratch.

plan $P^Q = \langle s_1^Q, s_2^Q, s_3^Q \rangle$
step s_1^Q:
$Join_1 = \{q_4\}, Dec_1 = \{\}$
$Rec_1 = \{Parent_*^*(q_4, q_1)\}$
step s_2^Q:
$Join_2 = \{q_2\}, Rec_2 = \{\}$
$Dec_2 = \{Child_*^*(q_1, q_2)\}$
step s_3^Q:
$Join_3 = \{q_3\}, Rec_3 = \{\}$
$Dec_3 = \{Child_*^*(q_1, q_3)\}$

Fig. 6: Plan for Q.

5. INCREMENTAL QUERY EVALUATION

This section presents a summary of the incremental query evaluation with the RCADG. The full paper [26] explains all data structures and algorithms in detail. The cache stores the queries, query plans and query results (both intermediate and final) obtained in the evaluation procedure just described. Given a new query Q^n, we compute a set of *cache hits* specifying (1) the relevant schema hits of all cached queries overlapping with Q^n on the schema level (Sect. 2) and (2) the evaluation steps providing the right "snapshots" of their result sets (Sect. 3.2). These results may need to be restricted, completed, or both, depending on the additional constraints in Q^n and the cached queries. To this end, one or more query plans are created for computing the results of Q^n based on the data specified by the cache hits and perhaps the full data set in the element table. Also, the same subset of Q^n's result may be obtained from distinct cache hits. A cost measure indicates which of several alternative plans to execute in order to exploit the best-fitting cache hits. Owing to schema information, no duplicates need to be eliminated when merging results from distinct query plans.

Structure of the cache. The cache consists of (1) the results of prior queries in the RDBS backend (Fig. 5) and (2) a main-memory index structure C containing all schema hits of cached queries. In C we look up those schema hits for any cached query Q which share at least one edge with a schema hit for Q^n s.t. the corresponding binary S-constraints in Q and Q^n are equivalent, as described in Sect. 2. To this end, each schema hit for a query Q to be cached is decomposed into its *schema edges* (the schematized binary S-constraints in Q) which are then stored in C. The same decomposition, applied to the schema hits of Q^n, produces the schema edges to be searched in C. For instance, looking up the schema edge $Parent'^*_*(\#5, \#1)$ from $\chi_1^{Q^n}$ (Fig. 3 *f.*) retrieves (among others) the tuple $\langle \chi^Q, s_1^Q, Parent_1^*(q_4, q_1) \rangle$ indicating that the same schema edge as part of χ^Q matched the $Parent_1^*(q_4, q_1)$ constraint in Q before, during step s_1^Q of the evaluation. Note how equivalent binary constraints are identified regardless of their direction, while unary constraints such as query keywords are ignored at this stage. In this way the schema edges in all schema hits for Q^n are looked up in C. A *query node binding* for each edge links the corresponding node pairs in Q^n and the cached queries (for $Parent'^*_*(\#5, \#1)$, the binding is $q_1^n \sim q_1$ and $q_3^n \sim q_4$).

Containment test for schema hits. The cache look-up produces candidate pairs $\langle \chi^Q, \chi^{Q^n} \rangle$ each consisting of a cached and an uncached schema hit, along with query node bindings and planning information. Each of these pairs must undergo the *containment test* to decide whether the set of matches to χ^Q indeed contains those to χ^{Q^n}, at least at some point during the evaluation of Q. The test determines the last step s_f in the query plan P^Q s.t. $ans(\chi^{Q^n}) \subset ans([\![\chi^Q]\!]_{s_f})$ as defined in Sect. 3.2. To this end, the D-constraints in P^Q are examined in the order of the evaluation steps. For each step s_i we need to know (1) whether the unary or binary query constraints processed in s_i are too restrictive for Q^n (this means $s_f = s_{i-1}$ unless $i = 0$), and if not, (2) which constraints in s_i are mirrored in Q^n. In the end, if s_f exists for χ^Q, it is saved in a new cache hit κ specifying also the extra constraints in Q^n to be checked against $ans([\![\chi^Q]\!]_{s_f})$.

Consider again the query plan for Q in Fig. 6. In step s_1^Q the query edge $Parent_*^*(q_4, q_1)$ was processed which has the counterpart $Parent_*^*(q_3^n, q_1^n)$ in Q^n (we can tell that from the query node binding above). Since the unary constraints on the corresponding nodes in both queries are also compatible (see the next paragraph), $ans(\chi_1^{Q^n}) \subset ans([\![\chi^Q]\!]_{s_1^Q})$ holds. Similarly the containment test succeeds for s_2^Q. By contrast, in s_3^Q the constraint $Child_1^*(q_1, q_3)$ was matched which is not mirrored in Q^n. Regarding question (1), we conclude that $ans([\![\chi^Q]\!]_{s_3^Q})$ might not contain all matches to $\chi_1^{Q^n}$ and therefore $s_f = s_2^Q$ for χ^Q. As to question (2), a comparison of the keyword constraints in both queries (see below) reveals that the only additional constraint to be checked against $ans([\![\chi^Q]\!]_{s_3^Q})$ is $contain_{\text{"Lee"}}(\#2)$, which translates into a join of the tables in Fig. 4 *b.* and 5 *c.*

Comparing keyword constraints. The query node bindings produced by the schematization specify pairs of query nodes whose keyword constraints must be reconciled for the containment test to succeed. For instance, the test would fail if q_4 in Fig. 3 *a.* required other keywords than "female", since this would be too restrictive for q_3^n in Fig. 3 *c.* Depending on whether the two query nodes specify (if any) a conjunction or a disjunction of keywords to be contained or governed by matching elements, distinct relations on the keyword sets (equality, sub-/superset, non-empty intersection) are allowed. For the full compatibility matrix, see [26].

Creating query plans. The core of the planning algorithm sketched in Sect. 4 is common to both the evaluation with and without cache. The only difference is that when exploiting cache contents, some D-constraints in Q^n need not be matched any more. In particular, the element table is joined only for query nodes and keyword constraints in Q^n that are missing in the cached query. Additional binary constraints in Q^n are decided if they involve matches in the cached result, otherwise reconstructed if possible. The time and space complexity is linear in the number of edges in Q^n.

In our example, the plan $P_\kappa^{Q^n}$ for the cache hit κ (spec-

ifying how to compute matches to $\chi_1^{Q^n}$ from those to χ^Q) involves a single join with the element table, needed to retrieve those matches to q_2 in $[\![ans(\chi^Q)]\!]_{s_2^Q}$ which contain an occurrence of the keyword "Lee". An alternative plan $P_{\kappa'}^{Q^n}$ based on the cache hit κ' for $\chi_1^{Q^n}$ and $\chi^{Q'}$ requires two joins since the "female" constraint must be checked as well. In general, to avoid the repeated matching of the same schema hit for Q^n, we must decide which cache hit is used to retrieve the matches to which schema hit. This is done based on a cost measure for the query plans created for the cache hits, which at the moment simply counts the number of element-table joins needed to execute a plan. Thus $P_{\kappa}^{Q^n}$ has a lower cost than $P_{\kappa'}^{Q^n}$, and κ' is discarded. More sophisticated methods could also take into account selectivity estimates for keyword and label constraints. If multiple cache hits for distinct schema hits for Q^n are found, the corresponding query plans may be executed in parallel or sequentially, in which case the plans with the lowest cost are executed first. Finally, we create a single query plan covering all remaining schema hits for Q^n that must be matched from scratch ($\chi_2^{Q^n}$ in the example), and add the matches to the final result.

6. EXPERIMENTAL EVALUATION

To evaluate the incremental query processing described in the previous section, we have conducted two different experiments. One studies how the performance for selected queries varies when cached queries with different degrees of similarity are available, while the other relates the cost and benefit of caching on a larger scale, using a randomly generated cache content and query workload. Both experiments rely on a number of performance measures explained below.

The test system is a Java implementation (JDK 1.5.0) of the data structures and algorithms presented in [26], including the optimizations described there. The mappings in C and L are realized by hash tables providing access in amortized constant time. At system start-up the main-memory part of the query cache is loaded and connections to the RDBS are established once for a test session. This takes 1-2 seconds. The test system accesses through JDBC a PostgreSQL v. 7.3.2 backend running on the same machine, with database cache disabled. Apart from the database server and the test system, the computer is idle during the experiments. All queries are processed 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 9.1 with kernel 2.4.26. Characteristics of all test queries are shown in Fig. 8; for details about the document collections see [10]. The runtime figures shown below represent the time needed for computing all matches to all nodes in a query, obtained by averaging three out of five consecutive runs after discarding the best and worst result to minimize artefacts. Results obtained incrementally have been compared to the results computed from scratch to make sure the evaluation is correct. The cache contents always include intermediate and final results. All result tables on disk are indexed with a B$^+$-Tree on the *id* column.

Cost and benefit. We quantify the benefit of incremental query evaluation by measuring the *processing time* and the *number of joins* needed to compute the result (although counting the number of tuples being joined would be more accurate). Since schema matching is the same for the evaluation from cache and from scratch, we ignore the n-way self-join of the path table (Sect. 4) and simply count the number of joins with the larger element table. On the other hand, retrieving and matching overlapping queries and their schema hits in the main-memory part of the cache takes some extra processing time not needed when evaluating from scratch. We refer to this overhead as *(cache) search time*. Besides, the persistent cache data structures consume extra storage both in main memory and on disk, which we denote as *cache size (in memory/on disk)*.

We now define the notion of *cache support* to measure how "useful" the cache contents C are for evaluating a given query Q^n. Let X be the set of schema hits for Q^n and P an evaluation plan for processing Q^n from scratch with minimal join cost $join(P)$ (Sect. 4). Besides, let $\bigcup_i X_{\kappa_i}$ be a partition of X s.t. each X_{κ_i} contains the schema hits represented by the cache hit κ_i computed for Q^n and C (Sect. 5). Finally, let P_{κ_i} denote the query plan devised for κ_i. Then the cache support for Q^n and C is defined as $[1-\frac{\Sigma_i |X_{\kappa_i}| \cdot join(P_{\kappa_i})}{|X| \cdot join(P)}] \cdot 100\%$. Intuitively, this quantifies how many joins have been avoided compared to the evaluation from scratch. Note that a cache support of 100% does *not* necessarily mean that Q^n is found in the cache, only that no joins are needed to evaluate Q^n from cache. In the experiments described next, the cache support indicates to what extent the evaluation can possibly benefit from the cache. Our main questions regard the following three optimization goals:

effectiveness Are useful cached queries exploited if available?
efficiency Does the benefit of caching outweigh the overhead?
scalability How does the overhead vary with growing cache size?

Small-scale experiment. To illustrate the effectiveness of our approach, we consecutively evaluate a single query Q_i^n on five caches each containing another singleton query Q_{ij}, $1 \le j \le 5$ (all queries for *IMDb*, see Fig. 8). As can be seen from the abscissa in Fig. 9, the cache support grows from 0% ($j = 1$) to 100% ($j = 5$). The experiment is conducted four times, and in each run $1 \le i \le 4$ the cached queries Q_{ij} are derived from Q_i^n by a specific class of editing operations as they occur in user sessions with relevance feedback: `N` denotes modifications of the query nodes and `L` of the labels; `-`/`+` means making the query more or less restrictive, respectively. Adding a node, e.g., is in class `N-`, whereas adding an alternative label to a node which already has a label constraint would be in `K+`. Fig. 9 shows that for all classes, the processing time decreases significantly with growing cache support, down to 20% of the time needed without cache.

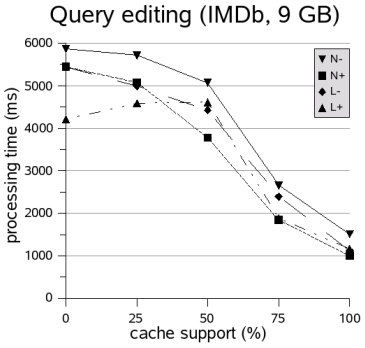


Fig. 9: Small-scale results.

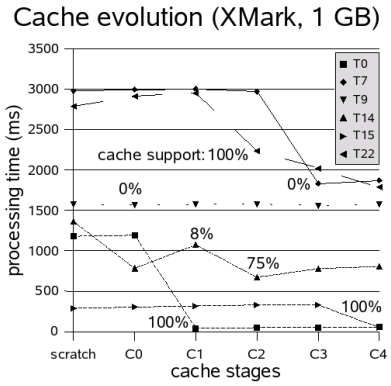
Large-scale experiment. The second experiment targets all optimization

cache contents	cache stages			
	C1	C2	C3	C4
queries	38	73	134	199
cache edges	525	1097	3169	5932
mem. size (MB)	1	1	3	5
disk size (MB)	77	378	646	872

Fig. 10: Cache contents in the large-scale experiment (*XMark*).

query	scratch	time 0	time 1	time 2	time 3	time 4	sup 0	sup 1	sup 2	sup 3	sup 4	ovh 0	ovh 1	ovh 2	ovh 3	ovh 4
T0	1181	1193	39	50	54	58	0	100	100	100	100	0	37	42	45	47
T4	7629	7672	79	94	113	113	0	100	100	100	100	0	33	45	32	52
T14	1363	783	1073	671	780	807	0	8	75	75	75	0	4	12	13	32
T15	289	305	318	330	332	54	0	0	0	0	100	0	3	6	7	45
T20	29765	30372	30143	6139	4097	4721	0	0	98	98	98	0	0	4	4	46
T22	2788	2914	2951	2237	2020	1788	0	0	100	100	100	0	0	1	1	1
T21	16005	15403	15492	14049	14875	14953	0	17	17	17	17	0	0	0	1	1
T7	2980	2996	3005	2970	1833	1872	0	0	0	0	0	0	1	1	2	3
T18	1154	1177	1102	1122	979	954	0	0	0	0	50	0	1	2	3	3
T19	3243	3267	3380	3320	3263	3013	0	0	0	0	0	0	1	1	2	3
T23	16842	16919	16965	16894	6379	6489	0	0	0	0	0	0	0	0	1	1
T2	11015	11159	11220	11335	12767	13572	0	0	0	0	0	0	1	2	4	6
T3	40	50	85	101	117	157	0	0	0	0	17	0	35	41	50	52
T5	39	54	61	78	69	84	0	0	0	0	0	0	29	42	39	39
T9	1576	1567	1574	1580	1560	1576	0	0	0	0	0	0	0	0	0	0
T13	6896	6974	7079	7191	8977	9050	0	0	0	2	2	0	1	3	4	7
T10	26	27	27	29	27	26	0	100	100	100	100	0	0	0	8	9
T16	41	58	48	51	50	52	0	100	100	100	100	0	2	3	3	4
T17	48	47	54	66	67	80	0	100	100	100	100	0	15	26	41	31

a. complete results (ms) (sup: % cache support; ovh: % search time)



b. selected results

Fig. 7: Results of the large-scale experiment on 1 GB *XMark* for all stages C0–C4 in the cache evolution.

query	small scale (IMDb)				large scale (XMark)																	
properties	N-	N+	L-	L+	T0	T4	T14	T20	T22	T21	T7	T18	T19	T23	T2	T3	T5	T9	T13	T10	T16	T17
nodes	6	5	5	5	4	4	11	13	5	5	8	11	2	10	4	5	6	2	5	2	2	2
labels	6	5	5	7	4	1	4	4	5	1	8	10	1	8	5	4	5	2	4	2	2	2
keywords	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1
schema hits	1	1	1	2	1	6	12	48	1	6	1	1	514	4	84	6	1	1	57	1	8	1
result size	44838	44838	44838	35908	18	96	32	1859	60588	147123	16859	11000	157239	9576	6343	6	1	489	4941	1	1	1

Fig. 8: Properties of all test queries run against the 9 GB *Internet Movie Database* (*IMDb*) and 1 GB *XMark* collection.

goals in a large-scale setting, simulating a cache growing from 0 to 199 distinct queries in five stages (Fig. 10, initial stage C0 omitted), as it could emerge during continuous retrieval. In the absence of a real-world query workload which could only be extracted from the log of a system in productive use, we model the workload as a sequence of random queries, including some popular or "hotspot" queries which are more likely to be asked repeatedly (possibly with modifications). From a seed of 150 distinct randomly generated tree queries against the 1 GB *XMark* collection (Fig. 10), we randomly remove 15 hotspot queries. Then we create five exact copies and five variants of each hotspot query according to the editing operations discussed in the previous experiment, e.g., by adding/removing a node, label or keyword constraint. The workload is the union of the resulting 150 hotspot queries and the remaining 135 seed queries. Now 23 distinct test queries are randomly removed from the workload. Note that the probability that a hotspot query is selected (which benefits from its duplicates or variants in the cache) is the same as the probability that a hotspot query occurs in the original workload, which is reasonable. The remaining workload (199 queries after removal of duplicates) is randomly sorted and partitioned into four subsets of 38, 35, 61 and 65 distinct queries, resp. These are evaluated from scratch and the results of each query set are added to the initially empty cache, which yields the stages in Fig. 10. The main-memory footprint of the cache is modest even when nearly 1 GB of results are cached on disk (including B^+-Trees). Since the cache size grows linearly with the number of cache edges, the system should easily scale up by three orders of magnitude.

The results of evaluating all test queries against the five cache stages and from scratch are listed in Fig.7 *a.* The time, sup and ovh columns list the processing time, cache support and search time (relative to the processing time) for each query and cache stage, respectively. The first seven queries all benefit from cache contents at different stages. For instance, there are overlapping queries for T0 and T4 in C1 which avoid additional joins (cache support 100%) and decrease the processing time by a factor 30 and 97, respectively. In subsequent stages, the search time increases a little, but does not depend on the overall cache size. The other five queries benefit only at later stages, up to which point there is a negligible overhead. T21 retrieves 130,000 distinct hits, 10% of which are retrieved from the cache after only 0.6 seconds. This illustrates how incremental evaluation may increase the reactivity of the system even when only part of the results can be obtained from the cache. Note that for T22 which already has 100% cache support in C2, the performance further improves in C3 and C4 where newly cached queries permit more efficient query plans. This effect, which we also observe for the second group of queries in Fig. 7 *a.*, is not captured by our notion of cache support. Fig. 7 *b.* plots selected results for all stages. Note the negligible overhead of incremental evaluation against an empty cache, compared to the evaluation from scratch.

The third group in Fig. 7 *a.* lists queries which do not benefit from cache contents, mostly by lack of overlapping queries in the cache. Again we observe a small search overhead (inevitable for deciding whether or not to use the cache) which grows much slower than the cache. The results of T3 and T13 are computed from 2-3 overlapping queries with small cache support, hence the evaluation from scratch is faster. Query planning with selectivity estimates, as mentioned above, is likely to eliminate such cases. The same applies to the queries in the fourth group, where the cache look-up does not pay off compared to the extremely fast evaluation from scratch. T2 benefits largely from the fact that query node bindings shared by multiple schema hits are processed only once [26]: In line with the structure of the

query (Fig. 8), there are thousands of containing schema hits in the cache which only differ w.r.t. the leaf node. Matching the shared schema edges repeatedly would cost a needless extra 14 seconds, which is completely avoided. Nevertheless, this indicates that for queries with an extremely unspecific structure the search time might be considerable. Such cases are detected immediately after schema matching, when the number of schema hits is known. If a certain threshold is exceeded, one may still decide to look up only some schema hits in the cache, or evaluate the whole query from scratch. Besides, queries with very large result sets should probably not be cached (also in view of the storage consumption).

7. RELATED WORK

Many papers have studied the theoretical complexity of view-based query processing on semistructured data for different query languages, mostly based on regular path expressions [27, 28]. For results in the presence of DTDs, see [29, 30, 31]. [13] shows that both *view-based query containment* and *view-based query answering* are PSPACE-complete for tree-shaped conjunctive queries and EXPSPACE-complete for arbitrary conjunctive queries (including forward and reverse relations in both cases). In question answering the results of a given query are computed from both the view definitions and extensions [32]. Thus the problem is closer to our approach than query containment where queries are compared based on their intensional description only.

Despite the high theoretical complexity of query processing in the presence of views, a number of different approaches have been proposed which strive to push the practical efficiency to its limits, building on various query evaluation techniques such as, e.g., native XQuery engines [20, 18], two-way finite state automata [13], tree automata [33] or LDAP servers [15]. These approaches differ in several respects from the present work. First, unlike our approach to enhancing user interaction, prior work in the field is mostly targeted towards efficient query processing in a distributed environment, where the query cache acts as part of the middleware in a multi-tier architecture [20, 21, 16]. Besides, query containment and overlap is mostly checked intensionally (except in [15]) and in particular, without exploiting schema information. Although DTDs or DataGuides are sometimes used for formulating queries [12, 1], no approach we know of uses schema information for indexing cache contents or checking query containment and overlap. Third, no strategies or index structures have been proposed for choosing the closest queries in the cache, a problem which arises for all but the few approaches where all cached query results are merged [12]. Finally, we are not aware of any systems caching both final and intermediate results. Consequently, the impact of query planning on the cache contents is largely ignored.

Since a thorough comparison of the aforementioned and more recent work [22, 23, 19] is infeasible in the scope of this paper, we only sketch a couple of representative approaches instead. *ACE-XQ* [20, 34, 17, 16] (formerly *XCache* [33]) answers XQuery expressions using materialized views. A *containment mapping* is established between the variables in a new XQuery and a cached one. To this end, queries to be cached are normalized and then described in terms of the variables occurring in the `RETURN` clause or elsewhere in the query, the path expression connecting them and conditions such as keyword constraints. To benefit from cached results, variables in the new query may only involve stricter conditions on structure or content than their counterparts in the cached query (although [20, 16] report on results for overlapping queries, which are not explained). [20] elaborates on XQuery containment in the presence of *hierarchical multi-valued dependencies* among variables, which can define different groupings of the same data. Replacement strategies are studied in [34]. Yet the problem of how to choose the best cached query for containment mapping remains open.

[12] proposes an *incomplete tree* as a main-memory cache structure. It is a prefix of the document tree that gradually becomes larger as more query results are added to the cache. Queries are prefixes of the schema tree (a simplified DTD is used instead of the DataGuide as in our approach), together with content conditions and label negation, but no * and // constructs. To indicate which data are missing from the cache, the complements of the cached queries are intensionally represented in the incomplete tree by extra paths with content conditions and DTD-style multiplicities. This incompleteness information is inferred by negating incoming queries, which may result in an exponential growth of the cache. [12] discusses several workarounds which either make the cache manipulation more complex or further restrict the query language. On the other hand, the incompleteness information allows to create non-redundant remainder queries in polynomial time. However, [12] states that this does not guarantee practical efficiency, and no experimental evaluation is provided. By contrast, our approach is to deduce missing data on-the-fly to avoid the exponential blow-up.

Based on the work in [12], [14] introduces a *modified incomplete tree (MIT)* which intensionally describes the data currently cached rather than the data missing from the cache. Combined with a partitioning of the possible element content into predefined domain ranges, this avoids the exponential growth of the incomplete tree. Built on top of the MIT, the *XCacher* system [14] answers simplified XQuery requests which are first translated into expressions of the same prefix-selection query language as in [12]. Both use a cache which contains all elements on root paths to query matches, but is still supposed to reside in main memory. To cope with obvious space limits, [14] sketches a simple replacement strategy (*least-recently used*). By contrast, our approach exploits the BIRD labelling scheme to reconstruct root paths, which therefore need not be cached. It also differs from [12, 14] in its combined extensional and intensional handling of query containment and overlap (see above).

HLCaches [15] is an XPath cache on top of the LDAP data model. Results are cached with the original XPath string (combined intension/extension). Each result element is stored together with its context node. Comparing sets of context nodes of a cached and a new query with the same intension determines equivalence and overlap. Complex queries may be decomposed into subqueries to be looked up in the cache, but no decomposition strategy for new or cached queries is given. Only subqueries benefit from cached results which have been asked before in exactly the same form, i.e., partial results cannot be reused or combined.

8. CONCLUSION AND FUTURE WORK

In this paper we present a method for exploiting cached results during the evaluation of XML queries in an RDBS. We show how extending the cache with schema information, intermediate results and query plans helps to detect reusable queries in the cache and to extract the useful parts of their

results. Algorithms are presented for the combined query evaluation with and without cache, and for selecting those cache contents from which the desired data can be obtained with the least computational and I/O effort. A careful evaluation of costs and benefits demonstrates that our approach is effective, efficient and scalable.

Several problems remain to be solved in future work. First, to retain the most useful data in a cache of limited size, a replacement strategy is needed. Besides standard strategies for the maintenance of priority queues, like *LRU/LFU*, possible hints for assigning appropriate priorities could come from explicit user feedback or silent monitoring of user interaction. Alternatively, we might choose to retain results which were expensive to compute. Second, when the document collection changes some or all cache contents may become stale. Since the label paths to updated elements are known from the element table, we might use the existing main-memory index to retrieve stale cache contents efficiently, exploiting schema information in the same way as for detecting query overlap. To some extent, the robustness of our cache also depends on the underlying tree encoding.

9. REFERENCES

[1] Goldman, R., et al.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: Proc. 23rd VLDB. (1997) 436–445

[2] Naughton, J.F., et al.: The Niagara Internet Query System. IEEE Data Engin. Bulletin **24** (2001) 27–33

[3] Jagadish, H.V., et al.: TIMBER: A Native XML Database. VLDB Journal **11** (2002) 274–291

[4] Fiebig, T., et al.: Anatomy of a Native XML Base Management System. VLDB J. **11** (2002) 292–314

[5] Shanmugasundaram, J., et al.: Relational Databases for Querying XML Documents: Limitations and Opportunities. In: Proc. 25th VLDB. (1999) 302–314

[6] Deutsch, A., Fernández, M., Suciu, D.: Storing Semistructured Data with STORED. In: Proc. 18th SIGMOD Conference. (1999) 431–442

[7] Bohannon, P., Freire, J., Roy, P., Siméon, J.: From XML Schema to Relations: a Cost-based Approach to XML Storage. In: Proc. 18th ICDE. (2002) 64–75

[8] Jiang, H., et al.: Path Materialization Revisited: An Efficient Storage Model for XML Data. In: Proc. 13th Austr. Database Conf. (2002)

[9] Grust, T.: Accelerating XPath Location Steps. In: Proc. 21st SIGMOD Conf. (2002) 109–120

[10] Weigel, F., et al.: Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems. In: Proc. 7th WIDM. (2005)

[11] Halevy, A.Y.: Answering queries using views: A survey. VLDB Journal **10** (2001) 270–294

[12] Abiteboul, S., et al.: Representing and Querying XML with Incomplete Inform. In: Proc. 20th PODS. (2001)

[13] Calvanese, D., et al.: View-based Query Answering and Query Containment over Semistructured Data. In: Proc. 8th DBPL. (2002) 40–61

[14] Hristidis, V., Petropoulos, M.: Semantic Caching of XML Databases. In: Proc. 5th WebDB. (2002) 25–30

[15] Marrón, P.J., Lausen, G.: Efficient Cache Answerability for XPath Queries. In: Proc. Int. Workshop on Data Integration over the Web. (2002)

[16] Chen, L., Rundensteiner, E.: ACE-XQ: A CachE-aware XQuery Answering System. In: Proc. 5th WebDB. (2002)

[17] Chen, L.: A Semantic Caching System for XML Queries. PhD thesis, Worc. Polytechnic Inst. (2003)

[18] Shah, A., Chirkova, R.: Improving Query Performance using Materialized XML Views: A Learning-based Approach. In: Proc. 1st XSDM. (2003) 297–310

[19] Mandhani, B., Suciu, D.: Query Caching and View Selection for XML Databases. In: Proc. 31st VLDB. (2005) 469–480

[20] Chen, L., Rundensteiner, E.A.: XQuery Containment in Presence of Variable Binding Dependencies. In: Proc. 14th WWW-Conf. (2005) 288–297

[21] Kang, H., Han, S., Kim, Y.: Schemes of Storing XML Query Cache. In: Proc. 16th Australasian Database Conf. (2005) 55–64

[22] Matsumura, H., Tajima, K.: Incremental Evaluation of a Monotone XPath Fragment. In: Proc. 14th CIKM. (2005) 245–246 (Poster).

[23] Xu, W., Özsoyoglu, Z.M.: Rewriting XPath Queries Using Materialized Views. In: Proc. 31st VLDB. (2005) 121–132

[24] Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing XPath Queries. In: Proc. 28th VLDB. (2002) 95–106

[25] Weigel, F., et al.: The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithm. Operations. In: Proc. 3rd XSym. (2005) 49–67

[26] Weigel, F., Schulz, K.U.: Caching Schema Information and Intermediate Results for Fast Incremental XML Query Processing in RDBSs. Technical report, Univ. of Munich (LMU) (2006) See http://www.cis.uni-muenchen.de/∼weigel/Literatur/weigel06cachingtech.pdf.

[27] Miklau, G., Suciu, D.: Containment and Equivalence for an XPath Fragment. In: Proc. 21st PODS. (2002) 65–76

[28] Calvanese, D., et al.: Reasoning on Regular Path Queries. SIGMOD Record **32** (2003) 83–92

[29] Deutsch, A., Tannen, V.: Containment and Integrity Constraints for XPath. In: Proc. 8th KRDB. (2001)

[30] Wood, P.T.: Containment for XPath Fragments under DTD Constraints. In: Proc. 9th ICDT. (2003) 300–314

[31] Neven, F., Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs, and Variables. In: Proc. 9th ICDT. (2003) 315–329

[32] Calvanese, D., et al.: Query Processing using Views for Regular Path Queries with Inverse. In: Proc. 19th PODS. (2000) 58–66

[33] Chen, L., Rundensteiner, E.A., Wang, S.: XCache: a Semantic Caching System for XML Queries. In: Proc. 21st SIGMOD Conf. (2002) 618–618 Demo.

[34] Chen, L., Wang, S., Rundensteiner, E.A.: Evaluation of Replacement Strategies for XML Query Cache. Data and Knowledge Engineering J. **49** (2004) 145–175