Towards Aggregated Answers for Semistructured Data

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

[1] CIS, University of Munich, Oettingenstr. 67, 80538 Munich, Germany
{meuss,schulz}@cis.uni-muenchen.de
[2] Institute for Computer Science, University of Munich, Oettingenstr. 67, 80538 Munich, Germany

1 Introduction

Semistructured data [5, 34, 23, 31, 1] are used to model data transferred on the Web for applications such as e-commerce [18], biomolecular biology [8], document management [2, 21], linguistics [32], thesauri and ontologies [17]. They are formalized as trees or more generally as (multi-)graphs [23, 1]. Query languages for semistructured data have been proposed [6, 11, 1, 4, 10] that, like SQL, can be seen as involving a number of variables [35], but, in contrast to SQL, give rise to arrange the variables in trees or graphs reflecting the structure of the semistructured data to be retrieved. Leaving aside the "construct" parts of queries, answers can be formalized as mappings represented as tuples, hence called *answer tuples,* that assign database nodes to query variables. These answer tuples underly the semistructured data delivered as answers.

A simple enumeration of answer tuples is problematic for several reasons. First, the number of answer tuples for a query may grow exponentially in the size of both, the query and the database. Second, even if the number of answer tuples is manageable, the frequent sharing of common data between distinct answer tuples is no more apparent in their enumeration.

In this article, it is first argued that enumerating answer tuples is often not appropriate and that *aggregated answers* are preferable. Then, a notion of aggregated answers called *Complete Answer Aggregate (CAA)* generalizing [25, 24] is introduced and algorithms for computing CAAs are given. We only consider CAAs for semistructured data: In this context, CAAs seem particularly attractive since they reflect the graph structure of the database and the query in a very natural way. It is shown that CAAs enjoy nice complexity properties: (1) While the number of answer tuples may be exponential in the size of the query, the size of the CAA is at most linear in the size of the query and quadratic in the size of the database; (2) the complexity of computing the CAA of a query depends on the query's structural complexity (i.e. whether it is a sequence, tree, graph, etc.) but is independent of the structural complexity of the database. For tree queries, efficient polynomial algorithms are given. Besides, CAAs seem to be particularly appropriate for *answer searching* and *answer browsing.*

This article is organized as follows. The need for aggregated answers and the basics of CAAs are illustrated with a motivating example in Section 2. Section

3 introduces a few preliminary notions. Section 4 gives the formal definition of CAAs. In Section 5, a hierarchy of query problems of increasing complexity is defined. In Section 6, we describe algorithms for computing the CAA and analyze their complexity, for each problem of the hierarchy. Query answering using CAAs is discussed in Section 8. Section 9 discusses related work. Section 10 is a conclusion. For space reasons, no proofs are given. They are given in the full version of this paper [26].

2 Motivating Example

Consider a database $\mathcal{D}$ as in Figure 1 on research projects offering information on project managers, members and publications. Assume that the database is organized as a graph with labeled edges and nodes according to a model for semistructured data [9, 19, 1]. Projects x and their managers z such that the string "XML" occurs in a title element t (at any depth) of some articles y of projects x are retrieved by a query depicted in Figure 1. (The restructuring

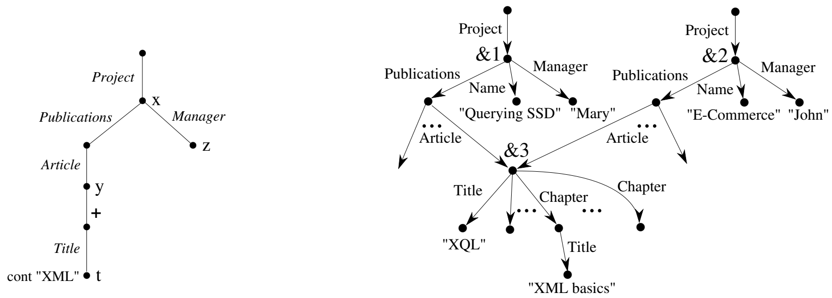


Fig. 1. An example query (left) and database (right)

facilities of full-fledge query languages for semistructured data [22, 12, 36] are not considered in this paper. Thus, only the "select" parts of queries are mentioned, possible "construct" parts are left implicit.)

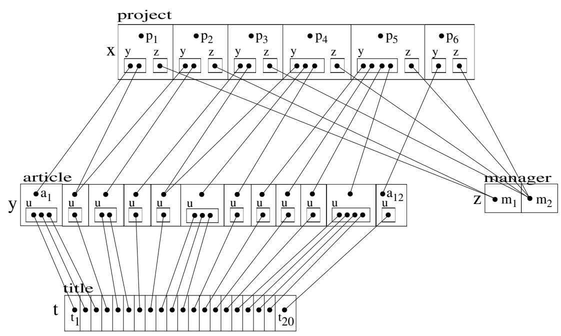
Some of the articles retrieved by the query Q are common to several projects in the database $\mathcal{D}$ (Figure 1). Evaluated against the database $\mathcal{D}$, the query Q returns the answer tuples $\langle x \mapsto \&1, y \mapsto \&3, z \mapsto$ "Mary", $t \mapsto$ "XML basics"$\rangle$ and $\langle x \mapsto \&2, y \mapsto \&3, z \mapsto$ "John", $t \mapsto$ "XML basics"$\rangle$.

More generally, if the string "XML" occurs in k titles of article &3, then Q admits $2k$ answer tuples all referring to &3. Furthermore, if an article is shared by n projects, then Q has $n \cdot k$ answer tuples. In case of more complex queries and/or of database items with more complex interconnections, as often arise in Web and e-commerce servers, an enumeration of answer tuples à la Prolog or à la SQL results in a combinatorial explosion. If e.g. the functional dependency *Project* $\rightarrow$

Manager does not hold and if each of the two projects has m managers, then Q admits $2 \cdot k \cdot m$ answer tuples. In general, such a product giving the number of answers of a query like Q can have any number of factors, i.e. the number of answers is exponential.

Arguably, for many applications such an enumeration of answer tuples is not appropriate. Instead, a data structure stressing the common subelements shared between (parts of) answer tuples as well as their graph relationships would often be more convenient. Let us call *aggregated answer* such a hypothetical data structure. Aggregated answers can help to recognize "bottlenecks" in the "answer space". Furthermore, aggregated answers make advanced query answering forms possible, cf. Section 7.

In this paper, *Complete Answer Aggregates (CAAs)* are proposed as a formalization of such a notion of aggregated answer and CAAs are shown to be efficiently computable. A CAA reflects the graph structure of the query it is computed from. The CAA computed for Q over an example database (larger than $\mathcal{D}$) has a "slot", represented in the figure below by a rectangle, for each variable x, y, z, and t. A slot for variable v contains possible binding elements for v: E.g. the slot for x contains project identifiers and the slot for z contains manager identifiers. The edges in are *CAA links.* They represent (sequences of) database edges. Note that a presentation of a CAA such as in the following figure is not intended for end users.



Admittedly, it is possible to generate in some cases simple kinds of aggregated answers with usual query languages like [4, 10], but this requires nested queries that might be complex [1] (cf. Sec. 4.1). In contrast, with CAAs, no complex queries are needed and aggregated answers are obtained in all cases.

CAAs are nothing else than semistructured data items of a certain kind. Thus queries can be posed to CAAs. Provided that the implementation of the query language is "CAA aware", such a querying can be performed without requiring from the user or application issuing the query to be aware of the internal structure of the CAAs constructed during query evaluation. Thus, CAAs are a convenient basis for an iterative, or cascade style query-answering: The evaluation of a query yields a CAA which can be stored and in turn queried using the same query language. Such a cascade style query-answering is often sought for in e-commerce applications [18]. Cf. Section 7 for more details.

3 Preliminary Notions

Definition 1. *A* database *is a tuple $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ where*
- *N is a (finite) set of* nodes*,*
- *$E \subseteq N \times N$ is a set of directed* edges*,*
- *L_N is a (finite) set of* node labels*,*
- *L_E is a (finite) set of* edge labels *or* features*,*
- *$\Lambda_N : L_N \to 2^N$ is a (total) function assigning to each node label a set of nodes,*
- *$\Lambda_E : E \to L_E$ is a (total) function assigning labels to edges.*

$\mathcal{D}$ is a sequence *(resp.* tree*,* DAG*,* graph*)* database *if the structure imposed by E upon N defines a (finite) set of sequences (resp. trees, DAGs, graphs).*

For simplicity, we do not consider multigraphs with multiple edges between nodes. Note that sequence, tree, and DAG databases are acyclic and that nodes in sequence (resp. tree) databases have at most one parent and one child (resp. one parent). Node labels are aimed at modeling attributes and attribute values. Multiple node labeling is allowed, so as to model textual content: A label w of node n might express that a string w occurs in the textual content of n. In the sequel, regular expressions α over the alphabet L_E of edge labels will be considered.

Definition 2. *Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and α a regular expression over L_E. A node $d \in N$ is an* α-ancestor *of $e \in N$, if there exists a path from d to e such that the sequence of labels along the path belongs to the regular language $\mathcal{L}(\alpha)$ induced by α.*

Definition 3. *Let X be an enumerable set of variables and $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ a database. An* atomic path constraint *is an expression of the form*
- *$A(x)$ called a labeling constraint,*
- *$x \to_1 y$ called a child constraint,*
- *$x \to_f y$ called a f-child constraint,*
- *$x \to_+ y$ called a descendant constraint,*
- *$x \to_\alpha y$ called an α-descendant constraint,*

where $x, y \in X$, $A \in L_N$, $f \in L_E$, and α is a regular expression over L_E. Atomic path constraints of the latter four types are called edge constraints*.*

In the sequel, we only consider regular expressions α where the empty word ϵ does not belong to $\mathcal{L}(\alpha)$. Edge constraints of the form $x \to_1 y$, $x \to_f y$, and $x \to_+ y$ can be seen as special cases of α-descendant constraints. Without loss of generality, the (non-atomic) path constraints considered in this paper are conjunctions of atomic path constraints containing at most one atomic edge constraint $x \to_? y$ for each (ordered) pair (x, y) of query variables. Depending on their nature, edge constraints might impose sequence, tree, DAG, or graph structures on the variables.

Definition 4. *A* sequence *(resp.* tree*,* DAG*,* graph*)* query *is a conjunction of atomic path constraints the atomic edge constraints of which impose a sequence (resp. tree, DAG, graph) structure on its variables. If D is a database and Q a query with set of variables X_Q, then $(Q, X_Q, \mathcal{D})$ is an* evaluation problem.

Note that queries of either type can be represented as graphs.

Definition 5. *Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and Q a query. An* answer *to Q in $\mathcal{D}$ is a mapping μ that assigns a node in D to each variable in Q in such a way that*
- *$\mu(x) \in \Lambda_N(A)$ whenever Q contains the labeling constraint $A(x)$,*
- *$(\mu(x), \mu(y)) \in E$ whenever Q contains the child constraint $x \rightarrow_1 y$,*
- *$(\mu(x), \mu(y)) \in E$ and $\Lambda_E(\mu(x), \mu(y)) = f$ whenever Q contains the f-child constraint $x \rightarrow_f y$,*
- *$\mu(x)$ is an α-ancestor of $\mu(y)$ whenever Q contains the α-descendant constraint $x \rightarrow_\alpha y$.*

If $(Q, X_Q, \mathcal{D})$ is an evaluation problem and μ an answer to Q in $\mathcal{D}$, then μ is called a solution *of $(Q, X_Q, \mathcal{D})$.*

Note that according to Definition 5, cyclic, i.e. proper graph queries have no answers in acyclic, i.e. sequence, tree, or DAG databases.

4 Complete Answer Aggregates

Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and Q a query with set of variables X_Q.

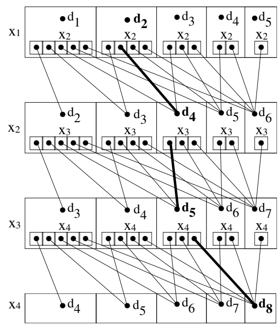
Definition 6. *An* answer aggregate *for the evaluation problem $(Q, X_Q, \mathcal{D})$ is a pair (Dom, Π) such that*
- *$\mathrm{Dom} : X_Q \rightarrow 2^N$ assigns to each variable of Q a set of nodes of $\mathcal{D}$ such that each $d \in \mathrm{Dom}(x)$ has label A if the labeling constraint $A(x)$ of Q,*
- *Π maps each edge constraint $x \rightarrow_1 y$ (resp. $x \rightarrow_f y$, $x \rightarrow_+ y$, $x \rightarrow_\alpha y$) of Q to a set $\Pi(x, y) \subseteq \mathrm{Dom}(x) \times \mathrm{Dom}(y)$ such that for all $(d, e) \in \Pi(x, y)$ e is a child (resp. f-child, descendant, α-descendant) of d. A node $d \in \mathrm{Dom}(x)$ is called a* target candidate *for x. A pair $(d, e) \in \Pi(x, y)$ is called a* link *between d and e.*

Definition 7. *An* instantiation *of an answer aggregate (Dom, Π) is a mapping μ that maps each $x \in X_Q$ to a node $\mu(x) \in \mathrm{Dom}(x)$ such that $(\mu(x), \mu(y)) \in \Pi(x, y)$ whenever Q contains an edge constraint $x \rightarrow_1 y$, $x \rightarrow_f y$, $x \rightarrow_+ y$, or $x \rightarrow_\alpha y$. Each node of the form $\mu(x)$, as well as every link of the form $(\mu(x), \mu(y))$ (for $x, y \in X_Q$) is said to* contribute *to instantiation μ.*

Note that each instantiation of an answer aggregate for $(Q, X_Q, \mathcal{D})$ defines an answer to Q in $\mathcal{D}$.

Definition 8. *(Dom, Π) is a* complete answer aggregate (CAA) *for $(Q, X_Q, \mathcal{D})$ if every answer to Q in $\mathcal{D}$ is an instantiation of (Dom, Π) and if every target candidate and every link of (Dom, Π) contributes to at least one instantiation.*

Example 1. Assume that $Q = x_1 \rightarrow_+ x_2 \wedge x_2 \rightarrow_+ x_3 \wedge \cdots \wedge x_{q-1} \rightarrow_+ x_q$ and assume that D consists of $n > q$ nodes $d_1, \ldots, d_n$ sequentially ordered (i.e. $(d_i, d_{i+1}) \in E$ for all $1 \leq i \leq n-1$). Q has $\binom{n}{q}$ answers in D. For $q = 4$ and $n = 8$ the CAA representing the 70 answers (each mapping the four query variables to database nodes) is depicted below. For one possible instantiation, contributing nodes and links are highlighted.



Lemma 1. *There exists a unique CAA for each evaluation problem.*

Size of CAAs Let $(Q, X_Q, \mathcal{D})$ be an evaluation problem. In the following, q will denote the number of variables plus labeling constraints of Q, n the number of nodes of $\mathcal{D}$, and a the maximal number of ancestors of a database node plus the number of links between these nodes. Note that, if $\mathcal{D}$ is a sequence or a tree database, then a is bounded by $O(h)$ where h is the height of $\mathcal{D}$.

As a measure for the size of a CAA (Dom, Π) for $(Q, X_Q, \mathcal{D})$ it makes sense to retain the total number of target candidates and links in (Dom, Π), i.e. $\Sigma_{x \in X_Q} |Dom(x)| + \Sigma_{x,y \in X_Q} |\Pi(x,y)|$ (where $\Pi(x,y) := \emptyset$ if x and y are not related by an edge constraint in Q). Although the number of answers to a query Q may be exponential, the size of a CAA is at most linear in the size of the query and quadratic in the size of the database:

Theorem 1. *The size of the CAA for an evaluation problem* $(Q, X_Q, \mathcal{D})$ *is* $O(q \cdot n \cdot a)$.

The full version [26] of this paper describes situations, where a better bound $O(q \cdot n)$ can be obtained.

CAAs as Semistructured data: According to Definition 6, the links of the CAA for an evaluation problem (Q, X_Q, D) are not labeled. If $x \rightarrow_f y$ is an f-child constraint in Q, then the label f can obviously be attached to each link in $\Pi(x,y)$. If $x \rightarrow_1 y$ is a child constraint in Q, then each link of $\Pi(x,y)$ clearly represents a labeled edge of $\mathcal{D}$, i.e. can be labeled like this database edge. Thus, if all edge constraints of the query are child- or f-child constraints, the CAA trivially extends to a semistructured data item. If the query Q contains descendant constraints of the form $x \rightarrow_+ y$ or $x \rightarrow_\alpha y$ and if $\mathcal{D}$ is a sequence or tree database, then for each $(d,e) \in \Pi(x,y)$ there exists in D a unique path π from d to e. The corresponding sequence of labels can be attached to the link (d,e) of the CAA yielding a semistructured data item in this case, too. If $\mathcal{D}$ is a proper graph database, a link $(d,e) \in \Pi(x,y)$ in general stands for a regular expression. Although this is less immediate than in the previous cases of sequence and tree databases, this expression can serve as a label of a CAA link. For space reasons, details are left out.

CAAs for extended query formalisms: Many query languages for XML data and semistructured data [11, 6, 23] allow for arbitrary value comparisons (joins), i.e. database leaves are assumed to carry values, and queries may contain value comparisons. A CAA does not suffice to completely represent exactly the answers to a query with value comparisons. Nonetheless, the CAA for the join-free part of the query can be built up representing a coarsening of the set of answers. Note that such a generalization of a query might speed up its evaluation and be appropriate for some applications such as e-commerce [18]. Extensions to CAA as defined here can be thought of for a faithful representation of the answers of a query with value comparisons.

5 A Query Hierarchy

We would to clarify how the structural properties of both query and database affect the complexity of computing the CAA. To this end we introduce the following notions:

Definition 9. *Let $\mathcal{EP} = (Q, X_Q, \mathcal{D})$ be an evaluation problem. $\mathcal{EP}$ is of type* S-S (Sequence-Sequence) *if Q is a sequence query and D a sequence database. Evaluation problem of types S-T, S-D, S-G, T-S, T-T, T-D, T-G, D-S, D-T, D-D, D-G, and G-G are similarly defined: The first letter (S: sequence, T: tree, D: DAG, and G: graph) denotes the type of the query, the second, that of the database.*

The thirteen classes of evaluation problems of Definition 9 form a hierarchy of increasing structural complexity. This hierarchy does not include the types G-S, G-T, G-D, because they would correspond to evaluation problems with cyclic (i.e. type G) queries: According to Definition 5, evaluation problems with cyclic queries have no solutions in acyclic (i.e. type S, T, and D) databases. Note that all the evaluation problems of the types specified in Definition 9 are non-trivial in the sense that they might have solutions.

Definition 10. *A query is called* simple *if all its edge constraints are child, f-child, or descendant constraints. An evaluation problem $(Q, X_Q, \mathcal{D})$ is* simple *if Q is simple.*

According to Definitions 4 and 3, a query is simple if it does not involve regular expressions. For simple queries, the algorithms described in the next section have optimal complexity.

6 Computation of CAAs

In this Section an algorithm for computing the CAA of a simple sequence query is first given. Then, its adaptation to simple tree queries is outlined. For space reasons, further adaptations, e.g. to tree queries involving regular path expressions, are not explained in this paper. A polynomial algorithm for this case is given in the full version [26].

Simple sequence queries

The algorithm depicted below takes a simple sequence query Q and a database $\mathcal{D}$ as arguments and computes the CAA (Dom_Q, Π_Q) for the evaluation problem induced by Q and $\mathcal{D}$. Starting from an empty array (line 3) and an empty set of edges (line 4) the algorithm adds target candidates to the domains (slots) of the query variables (line 15) and adds appropriate links to the set of edges (lines 31, 38). A pair (x, d) is said to be "added" to express that target candidate d is added to the domain (slot) of x. Possibly, pairs (x, d) are added that are "illegal" in the sense that $d \notin Dom_Q(x)$.

```
 1 procedure Aggregate comp_agg(Q,db)
 2 begin
 3   Dom:=empty DomainSet;
 4   Π:=empty EdgeSet;
 5   q_l:=leaf of Q;
 6   for all nodes d in db do
 7     map(Dom,Π,q_l,d);
 8   Agg:=Aggregate(Dom,Π);
 9   clean(Agg);
10   return Agg;
11 end;
12
13 procedure boolean map(Dom,Π,x,dx)
14 begin
15   add dx to Dom(x);
16   if dx satisfies the labeling
17     constraints of x then
18   begin
19     if x=root then return true
20     else
21     begin
22       y:=parent(x);
23       Anc:=appr_ancestors(dx,x,y);
24       map_found:=false;
25       for all dy_i ∈ Anc do
26         if dy_i ∉ Dom(y) then
27           if map(Dom,Π,y,dy_i) then
28           begin
29             map_found:=true;
30             add (y,x,dy_i,dx) to Π;
31           end
32         else
33         begin
34           if not is_red(Dom,y,dy_i) then
35           begin
36             map_found:=true;
37             add (y,x,dy_i,dx) to Π;
38           end;
39         end;
40       if map_found=false then
41         color_red(Dom,x,dx);
42       return map_found;
43     end;
44   else /*labeling constraints not satisfied*/
45   begin
46     color_red(Dom,x,dx);
47     return false;
48   end;
49 end;
```

Illegal pairs are detectecd and marked "red" (lines 40, 46). Only "legal" links, i.e., links in Π_Q, are introduced. As a last step, for each red pair (x, d) node d is deleted from the slot of x. After this slot "cleaning" (line 9), the CAA (Dom_Q, Π_Q) is obtained.

Let $x_1, \ldots, x_p$ be the ordered sequence of query variables such that $x_i \rightarrow_? x_{i+1}$ $(1 \leq i < p)$ occurs in the simple sequence query. The algorithm starts with the (unique) query leaf x_p (line 5). The outermost loop (line 6) calls for each database node d_p the recursive function `map`. This function returns a boolean value indicating whether the pair (x_p, d_p) is legal. First, the pair is added (line 15). Assume the pair (x_i, d_i) has been added. If $i = 1$, then the pair is legal (line 19). Otherwise (x_i, d_i) is legal if and only if there exists a legal pair (x_{i-1}, d_{i-1}) such that (d_{i-1}, d_i) satisfies the edge constraint $x_{i-1} \rightarrow_? x_i$ in Q. Hence, for each "appropriate" ancestor d_{i-1}, which depends on the kind of the edge constraint, it is checked whether (x_{i-1}, d_{i-1}) is a legal pair (lines 27, 34). Two cases are distinguished:

1. If (x_{i-1}, d_{i-1}) has not been previously added, then the legality of (x_{i-1}, d_{i-1}) is checked by a recursive call of the function `map`.

2. Otherwise (x_{i-1}, d_{i-1}) is illegal if it is marked red. In both cases, if (x_{i-1}, d_{i-1}) is legal then a link between (x_{i-1}, d_{i-1}) and (x_i, d_i) is added (lines 30, 37). After inspection of all appropriate ancestors of d_i, the pair (x_i, d_i) is marked red if no legal pair (x_{i-1}, d_{i-1}) is found (line 40). Similarly (x_{i-1}, d_{i-1}) is marked red if d_i does not satisfy all labeling constraints of x_i (line 46).

Complexity: Clearly, for each pair (x, d), `map` is called at most once. Hence, the total number of calls to this function is bounded by the maximal number of pairs $q \cdot n$. Under the assumption that each database node has links pointing to parent nodes, computing the set of appropriate ancestors of a target candidate x_i takes time $O(a)$. Whether a node x_i satisfies a given labeling constraint $A(x_i)$ can be checked in constant time. Since each labeling constraint refers to a unique query variable, the total time needed for all tests related to labeling constraints is $O(q \cdot n)$. Cleaning takes time $O(q \cdot n)$. Therefore, the overall complexity is $O(q \cdot n \cdot a)$.

Simple tree queries

In the case of simple tree queries we introduce the notion of an *adapter point* as the bottom-most common query node of two query paths. The query paths are processed consecutively. For each query path, the above algorithm is modified as follows: When reaching an adapter point x_{i-1} that has already been visited during processing of another path, no new target candidates for slot x_{i-1} is introduced. Furthermore, only the already collected non-red target candidates (x_{i-1}, d_{i-1}) are used for links between target candidates in x_{i-1} and x_i. If, after a query path has been fully processed, a target candidate (x_{i-1}, d_{i-1}) for an adapter point x_{i-1} has no links to a target candidate in x_i, then it is marked red.

With simple tree queries, cleaning is more complicated. Call *downwards isolated* target candidates (x_{i-1}, d_{i-1}) such that for some child x_i of x_{i-1} there are no links from (x_{i-1}, d_{i-1}) to a target candidate within the slot x_i. After entering all target candidates, downwards isolated target candidates are detected. Since they are illegal, they are marked red. Removal of red nodes may result in new downwards isolated target candidates. The removal of red target candidates is based upon a variant of (the second part of) the well-known AC-4 arc-consistency algorithm [29].

Complexity: Since the recursive calls do not fill slots of adapter points twice, in this case as well no target candidates are processed twice in the first part of the algorithm. This gives a time complexity of $O(q \cdot n \cdot a)$ for this part. For cleaning an adaption of arguments from [29] yields time complexity $O(q \cdot n \cdot a)$. Space complexity is still $O(q \cdot n \cdot a)$.

Adding regular path expressions

A simple modification suffices to adapt the algorithms for sequence and tree queries described above can also be applied to queries involving regular path expressions. At line 23, if the query contains an α-descendant constraint $y \rightarrow_\alpha x$, then the set of appropriate ancestors of the current node dx is now the set of all α-ancestors of the database node dx.

The computation of the sets of α-ancestors needs some extra time. Using standard techniques from automata and graph theory, it is shown in the full version [26] that the resulting time complexity is $O(q^2 \cdot n \cdot a)$.

Simple DAG and graph queries

Four evaluation problems of the hierarchy turn out to be NP-complete with respect to combined complexity.

Define the *weight* of a node (resp. variable) of a database (resp. query) as 1 plus the number of labels attached to the node (resp. variable). Define the *size* of a database (resp. query) as the sum of the weights of its nodes (resp. variables) and its the number of edges. Define the *size* of an evaluation problem $(Q, X_Q, \mathcal{D})$ as the sum of the size of Q and the size of $\mathcal{D}$. It is shown in the full version [26] that so-called 1-in-3 problems over positive literals [13] can be encoded as D-T evaluation problems, using a polynomial translation. The following theorem is a simple consequence.

Theorem 2. *Whether a simple D-T (resp. D-D, D-G, G-G) evaluation problem* $(Q, X_Q, \mathcal{D})$ *has a solution is NP-complete with respect to the size of* $(Q, X_Q, \mathcal{D})$.

		Database			
		S	T	D	G
Queries	S	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$
without	T	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$
reg. path	D	$O(q \cdot e \cdot n^3 \cdot a)$	(*) NP-compl.	(*) NP-compl.	(*) NP-compl.
expressions	G	-	-	-	(*) NP-compl.
Queries with	S	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$
reg. path expr.	T	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$

Table 1. Complexity for computing the CAA resp. (*) for deciding solvability

Table 1 summarizes the complexity results for the computation of CAAs (resp. for deciding solvability, marked with *) given in this section. The case D-S in the upper table is established in the full version [26]. The parameter e denotes the number of edges in the query.

Due to the results on the size of CAAs (cf. Section 4), the bounds given at lines S and T of the upper table are optimal.

In practice the worst case time complexity for computing a CAA can be exponential with D-T, D-D, D-G and G-G evaluation problems. This could be faced by a polynomial-time computation of an "upper approximation" to the

CAA, i.e. an answer aggregate yielding not only all answers, but also possibly containing target candidates or links not contributing to any answer. Such an upper approximation to a CAA can be obtained by first selecting a spanning tree T of the considered query, then compute the CAA for the subquery induced by this spanning tree. Links representing possible interpretations of the query edges that have been omitted from Q might then be added. Arc consistency techniques [29] can be used to erase nodes and edges that do not contribute to any instantiation.

7 Advanced Query Answering Using CAAs

Once the CAA for an evaluation problem has been computed, it can be exploited for advanced query answering techniques we call *answer searching* and for *answer browsing.* These notions are explained referring to the example of Section 2: A query Q to a research project database $\mathcal{D}$ retrieves projects x and the managers z of these projects such that the string "XML" occurs in a title element u (at any depth) of some article y of a project x.

Answer Searching: In essence, the CAA of a query is a data structure making explicit the interdependencies between the answers to a query. Comparing queries and investigating the interrelationships between the various answers to a query is needed in many applications. CAAs can be used for scanning, comparing, filtering, ordering in the style of search engines as well as for analyzing in any other manner the answers to a query. Particularly promising are search primitives for detecting commonalities and differences between answers, computation of aggregate values like averages, maxima and minima. Note that the nodes of database items stored in a CAA potentially give access to the subelements rooted at these nodes, thus giving rise to a semantically rich "answer searching".

The set of answers for the above-mentioned query Q can be searched for:
• managers leading the highest number of project,
• for managers leading at least 2 projects,
• for projects with at least 10 XML articles.
To answer such queries, aggregate values have to be computed from the CAA of query Q. Note that such semantically related aggregate values are often computed in the same query. Many database applications like molecular biology sequence analysis and e-commerce require to perform such advanced comparisons from large answer sets computed from some "base query" [18].

In many cases, the querying of the CAA of a base query can be carried out automatically. In some cases however, such an automatic "search" is not sufficient or not possible, an interactive "browsing of the answer space" is desirable.

Answer Browsing: A visualization of a CAA for a query can be convenient a basis for browsing the answers to that query. One can easily identify nodes of the CAA of query Q, like project 5, that deserve special attention by looking at the number of departing article links of CAA 1 in Figure 2. More elaborate visualization facilities would make it possible to directly browse between e.g. projects, articles, titles by following the links of the CAA.

If the CAA has a large number of nodes, then the user might get "lost in answer space". In such cases, it might be beneficial to restrict the visualization to a *view of the CAA*. In case of query Q, one might wish to restrict, say, the CAA to information associated with projects with at least three XML related articles. Also, *slot hiding* can provide with a better overview. Note that slot hiding corresponds to the projection operator of relational databases. If slot hiding is applied, then CAA links are inherited. Applying slot hiding to the running example yields CAA 2 depicted in Figure 2.

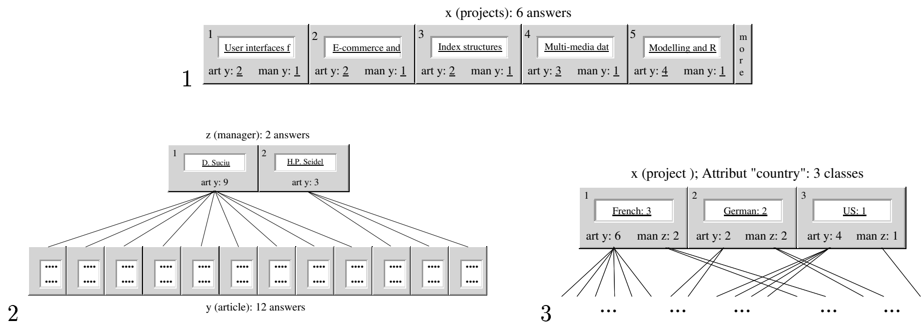


Fig. 2. Three presentation forms of CAAs

A further visualization technique based upon CAA is *clustered aggregation.* Assuming that nodes have further attributes, CAA nodes that have identical attribute values can be merged. In case of query Q, if projects have a "country" attribute, if projects p_1, p_3, p_4 are French, projects p_2 and p_6 are German, and if p_5 is a US-project, then applying clustered aggregation might yield CAA 3 of Figure 2. This presentation might be used to show which countries have projects of interest and how many.

General picture: cascade style query-answering: The possibilities for automated search as well as interactive browsing of answer sets the CAAs offer suggests that, for many applications, query answering can be processed in two or more successive phases, the first of which resulting in the construction and storing of the CAA, the following phases consisting in an inspection of this CAA or in the construction of further, more specialized CAAs. CAA inspection can consist both in an automated search or in an interactive browsing. In addition, to help analyzing and/or browsing answers, such a query answering based upon CAA can help to react rapidly to user query requests.

Cascade style query-answering query answering is possible with any query language, indeed. The contribution of CAAs lies in an intermediate data structure supporting this form of query answering which can be efficiently computed.

Note that for many novel applications such as e-commerce such a cascade style query-answering is needed [18].

8 Related Work

Tree Databases: For tree queries and tree databases, a simplified form of CAA based on the Tree Matching formalism [20] has already been introduced in [25, 24]. The present paper significantly extends over this early work.

Query formalisms for semi-structures data: Several query models for XML and semistructured data have been designed and/or implemented and used [23, 3, 33, 15, 30, 31, 4, 10], cf. [4, 10] for surveys. These query models are more ambitious than the model presented in this paper, however, in contrast to this paper they are not devoted to *aggregating answers*. Thus, the contribution of this paper is widely orthogonal and complementary.

Conjunctive queries: The queries considered in this paper are a special case of the conjunctive queries over relational databases as investigated in [7, 16]. However, it must be stressed that the distinction between tree queries and DAG or graph queries does *not* correspond to the conventional distinction between acyclic and cyclic conjunctive queries in database theory.

The distinction of database theory between acyclic and cyclic conjunctive queries refers to the hypergraph of the query, which is an undirected graph. In contrast, the query atoms considered in this paper are unary (labeling constraints) or binary (edge constraints). A binary atom $r(x, y)$ imposes a fixed orientation $x \rightarrow y$ on $\{x, y\}$ which reflects the direction of edges in the database. The conjunctions considered in this paper are such that the set of their binary atoms induces a sequence, tree, DAG, or graph structure on the query. Hence, the NP-hardness result for DAG-queries given in Section 6 is not in conflict with general results on polynomial tractability of acyclic conjunctive queries.

Dynamic programming and constraint reasoning: The algorithms described in Section 6 are closely related to methods of dynamic programming and to the arc-consistency techniques for constraint networks, cf. the full version [26] for details.

Index structures: Index structures as discussed in [23, 2, 14, 28, 27] can be used to improve the practical efficiency of the computation of CAAs. Details can be found in the full version [26].

9 Conclusion

The paper motivated and introduced "complete answer aggregates (CAAs)" as a model for aggregating the answers to a sequence, tree, DAG, or graph query in a semistructured database. Algorithms for the computation of CAAs for queries of various structural kinds have been presented. A hierarchy of evaluation problems the CAAs of which can be computed in polynomial time (with respect to combined complexity) has been given. Cases have been characterized where computation of CAAs is NP-complete.

The query model presented in this paper has been implemented for the special case of tree queries and of tree databases. The implementation is being tested with large collections of complex structured documents. The prototype currently available does not handle regular path expressions, but can cope with left-to-right order constraints between the children of a query node, as needed in document management. The necessary adaptation of the notion of a CAA as well as the mathematical and algorithmic background is given in [25, 24]. For this expanded signature, the time complexity of the algorithm for computing the CAA is $O(q \cdot n \cdot a \cdot log(n))$. The additional logarithmic factor comes from the fact that order information is not being taken into account and handled during query evaluation. An "answer browser" based on CAAs is currently being developed.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.
2. R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *SIGMOD Record*, 25(1):67–79, 1996.
3. The BBQ Page. `http://www.npaci.edu/DICE/MIX/BBQ/`, May 2000.
4. A. Bonifati and S. Ceri. A comparative analysis of five XML query languages. *SIGMOD Record*, March 2000.
5. P. Buneman. Semistructured data. In *Proc. ACM PODS'97*, 1997.
6. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks*, 31(11–16):1171–1187, May 1999.
7. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. 9th Annual ACM Symp. on Theory of Computing*, 1977.
8. XML/CML - Chemical Markup Language. `http://www.nottingham.ac.uk/pazpmr/README`, 1999.
9. M. Consens and A. Mendelzon. Graphlog: a visual formalism of real life recursion. In *Proc. ACM PODS'90*, 1990.
10. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *IEEE Data Bulletin*, 22(3):10–18, 1999.
11. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the WWW Consortium: `http://www.w3.org/TR/NOTE-xml-ql/`, August 1998.
12. M. Fernandez, J. Siméon, and P. Wadler. XML query languages: Experiences and exemplars. Draft, `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`, 1999.
13. M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.
14. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB'97*, 1997.
15. R. Goldman and J. Widom. Interactive query and search in semistructured databases. In *WebDB'98, Proc. Int. Workshop on the Web and Databases*, 1998.
16. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. In *Proc. 39th Annual Symp. on Foundations of Computer Science*, 1998.

17. N. Guarino, editor. *Int. Conf. on Formal Ontology in Information Systems*. IOS Press, 1998.
18. A. Gupta. Some database issues in e-commerce. Invited talk at the Int. Conf. on Extending Database Theory, http://www.edbt2000.uni-konstanz.de/invited/talks.html, 2000.
19. M. Gyssens, J. Paredaens, J. V. den Bussche, and D. V. Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, Aug. 1994.
20. P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Dept. of Computer Science, University of Helsinki, 1992.
21. A. Loeffen. Text databases: A survey of text models and systems. *SIGMOD Record*, 23(1):97–106, Mar. 1994.
22. D. Maier. Database desiderata for an xml query language. In *QL'98 - The Query Languages Workshop*, 1998.
23. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.
24. H. Meuss. *Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents*. PhD thesis, Dept. of Computer Science, University of Munich, 2000.
25. H. Meuss and K. U. Schulz. Complete answer aggregates for structured document retrieval. Technical Report 98-112, CIS, University of Munich, 1998.
26. H. Meuss, K. U. Schulz, and F. Bry. Towards aggregated answers for semistructured data. Technical report, Institute for Computer Science, University of Munich, 2000. http://www.cis.uni-muenchen.de/~meuss/agg_answers_full.ps.gz.
27. H. Meuss and C. Strohmaier. Improving index structures for structured document retrieval. In *IRSG'99, 21st Annual Colloquium on IR Research*, 1999.
28. T. Milo and D. Suciu. Index structures for path expressions. In *ICDT'99, Proc. 6th Int. Conf. on DB Theory*, 1999.
29. R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
30. F. Neven and T. Schwentick. Query automata. In *PODS'99*, 1999.
31. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. ACM PODS'00*, 2000.
32. J. Oesterle and P. Maier-Meyer. The gnop (german noun phrase) treebank. In *First International Conference on Language Resources and Evaluation*, pages 699 – 703, 1998.
33. The Strudel Page. http://www.research.att.com/sw/tools/strudel/, May 2000.
34. D. Suciu. An overview of semistructured data. *SIGACT News*, 29(4), 1998.
35. J. D. Ullman. *Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, 1989.
36. World Wide Web Consortium: XML Query Requirements. W3C Working Draft, http://www.w3.org/TR/2000/WD-xmlquery-req, August 2000. Eds. D. Chamberlin and P. Frankhauser and M. Marchiori and J. Robie.