Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs*

François Bry
Institute for Informatics, University of Munich
Oettingenstraße 67
80538 Munich, Germany
Francois.Bry@ifi.lmu.de

Tim Furche
Institute for Informatics, University of Munich
Oettingenstraße 67
80538 Munich, Germany
Tim.Furche@ifi.lmu.de

Benedikt Linse
Institute for Informatics, University of Munich
Oettingenstraße 67
80538 Munich, Germany
Benedikt.Linse@ifi.lmu.de

Andreas Schroeder
Institute for Informatics, University of Munich
Oettingenstraße 67
80538 Munich, Germany
Andreas.Schroeder@ifi.lmu.de

ABSTRACT

Query languages for semi-structured data on the Web in the form of XML or RDF have become an essential part of many applications and services. N-ary conjunctive queries, i.e., queries with any number of answer variables, are the formal core of most Web query languages including XSLT, XQuery, SPARQL, and Xcerpt. Despite a considerable body of research on the optimization of such queries against *tree-shaped* XML data, little attention has been paid so far to efficient access to *graph-shaped* XML, RDF, or Topic Maps. We propose a novel evaluation technique for n-ary conjunctive queries that applies to both tree- and graph-shaped data. It has the same complexity as the best known approaches that are restricted to tree-shaped data only. The core of the evaluation technique is a memoization data structure, called "memoization matrix", which holds (intermediary) results. It can be populated and consumed in different ways. For both, population and consumption, we propose two algorithms, each having its own advantages. The complexity of the algorithms is compared analytically and experimentally.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and networks*; H.2.4 [**Information Systems**]: Database Management—*system, query processing*

*This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net/`).

VLDB '06 Seoul, Korea

Keywords

query evaluation and optimization, conjunctive queries, memoization, semi-structured data, XML, RDF, Topic Maps

1. INTRODUCTION

Semi-structured data in the form of XML or RDF now dominates data representation and exchange on the Web. Accessing such Web data, often from multiple sources and in different formats, is more and more an essential part of many applications, e.g., for bibliography management, news aggregation, information classification, and digital asset management. Web query languages such as XSLT [10], XQuery [4], SPARQL [22], or Xcerpt [24] provide convenient and efficient means to access and process such data, whether it is stored in native XML or RDF databases or accessed via Web service query interfaces. Increasingly, Web applications need access not just to XML data, but also to data in other Web formats such as RDF or Topic Maps, even in the same query.

Efficient evaluation of queries over XML data has received considerable attention in recent years [5, 18, 13], including extensive studies of complexity of query evaluation for XPath [14], XQuery [17], and general conjunctive queries over trees [15].

However, these techniques and results have considered XML data as tree-shaped. For many applications, a *graph view* of XML is preferable, e.g., when links after XLink, (X)HTML, or XML's own ID/IDREF mechanism are considered first class elements of the data model. Furthermore, other semi-structured Web data formats such as RDF or Topic Maps are evidently graph-shaped. Therefore, we propose in this article a novel evaluation algorithm that exhibits on tree data the same worst-case complexity as the best known approaches for tree data, but operates with similar complexity also on graph data.

We formalize queries against semi-structured data, whether tree- or graph-shaped, as *n-ary conjunctive queries over unary and binary relations*. Conjunctive queries against tree data form a common formal basis for the query core of a large set of XML query languages such as UnQL [6], XPath [14], and thus XQuery [4]. Conjunctive queries against graph

	tree query	**graph** query
tree data	$O(q \cdot v^2 + o)$	$O(v^q)$
graph data	$O(q \cdot v \cdot e + o)$	$O(v^q)$

Table 1: Overview of Combined Time Complexity (q: number of query variables; e, v number of edges, vertices resp., in the data; o: size of output)

data form a common query core for RDF query languages such as SPARQL [22] and general semi-structured query languages such as Lorel [2] and Xcerpt [24].

Compared to full semi-structured query languages, the main restrictions of n-ary conjunctive queries are twofold: **(1)** They provide no *result construction*: The result of an n-ary conjunctive query is just a set of tuples of bindings for the n answer variables, each tuple representing one match, whereas full semi-structured query languages allow additional construction including grouping and aggregation on these results. **(2)** They are *composition-free* in the sense of [17], i.e., the query can only access the original input data, but no intermediary results can be constructed or queried, preventing in particular the use of views, rules, or functions. The second restriction is less easy to overcome and dropping it makes the query evaluation far more expensive [17].

An extension of the results presented here that drops the first restriction is straightforward and covers *composition-free core XQuery without negation* [17]. Indeed, the algorithms presented in this paper reaffirm the complexity results from [17] on tree data and extend them to graph data.

For the evaluation of n-ary conjunctive queries, we present two algorithms both founded on a compact data structure, called *"memoization matrix"*, for memoizing intermediary results during the evaluation of an n-ary query. The two algorithms differ only in the way the memoization matrix is filled: The first algorithm uses a bottom-up strategy for filling matrix cells starting with variables in leaf nodes of the query. The second algorithm performs a recursive descent over the query tree populating the matrix top-down from root to leaf query nodes. More involved population strategies are conceivable (e.g., a mix of the two presented algorithms or a pathwise population inspired by [20]), but only briefly outlined in this article.

Both algorithms can be applied in the same manner to tree and graph data, only the computation of the structural relations is affected by the type of data. Unsurprisingly, the shape of the query has a more pronounced effect on the complexity and performance of the evaluation algorithms: Where for path and tree queries the complexity of the evaluation algorithms is polynomial, it requires expontential time for evaluating graph queries. This is unsurprising in light of complexity results in [14, 15] that show that evaluation of graph queries even against tree data is NP-complete.

Two variants of matrix consumption and result generation algorithms are discussed: The first variant provides an in-memory representation of all result tuples that is useful if further processing such as aggregation or grouping is going to take place on the result tuples, the second variant computes the result tuples on the fly (using, e.g., a simple nested loop join) in exponential time but requiring only polynomial space. Both variants are justified and useful, depending on the construction performed on the results, if any.

The remainder of this article is organized along its **contributions**:

1: Based on the formalization of the query core of many semi-structured query languages as n-ary conjunctive queries (Section 2), a memoization technique for the compact representation of intermediary and final results of an n-ary conjunctive query is introduced in Section 3. This "memoization matrix" forms the core of the proposed novel evaluation technique, but allows for variation in two respects: first the population of the matrix and second the consumption of the matrix for result generation.
2: We introduce two algorithms for populating this matrix, one bottom-up in Section 4.1, one top-down in Section 4.2, and compare these algorithms w.r.t. complexity and likely usage scenarios. In an outlook, further population strategies are briefly noted. These algorithms can be used for *both* tree and graph data.
3: We introduce three algorithms for matrix consumption, one for tree queries (Section 5.1), one for graph queries (Section 5.2) which also enforces the remaining non-hierarchical relations that are unconsidered during matrix population, and one based on nested loop joins (Section 5.3). Above this, we show how order restrictions can be enforced using the matrix consumption algorithm for graphs (Section 6).
4: Careful complexity analysis of the algorithms in Sections 4 and Section 5 is complemented by an extensive experimental evaluation (Section 7) that confirms that the proposed algorithms are competitive with the best known evaluation techniques for tree data but also extend to graph data. To the best of the authors' knowledge, it is the first evaluation technique for semi-structured data with this property. This result applies over all three classes of queries considered, viz. n-ary path, tree, and graph queries. A summary of the complexity results for tree and graph queries is given in Table 1.
5: In an outlook further optimizations regarding advanced query constructs such as optional query parts or negation and the relation of the presented technique to relational query planing and constraint solving are briefly outlined (Section 8).

2. PRELIMINARIES

2.1 Graph Data Model

From the perspective of their data model, many Web representation formats such as XML, RDF, and Topic Maps have a lot of commonalities: the data is semi-structured, tree- or graph-shaped, and sometimes ordered, sometimes not (XML elements vs. XML attributes, RDF sequence containers vs. bag containers). In this article, we choose finite **unranked labeled ordered simple directed graphs** as common data model for Web data. Precisely, a query is evaluated against a data graph D over a label alphabet Σ_L and a value alphabet Σ_V. A data graph is represented as a 5-tuple

$$D = (N, E, R, \mathcal{L}, \mathcal{V}),$$

where N is the set of nodes of the graph, $E \subset N \times \mathbb{N} \times N$ the set of edges, $R \subset N$ the set of root vertices, $\mathcal{L} : N \to \Sigma_L$ the labeling function, and $\mathcal{V} : N \to \Sigma_V$ the value assignment function.

D is a *simple directed* graph, i.e., there are neither multi-edges nor loops in the graph (but other, i.e., indirect, cycles are allowed). This restriction is used to simplify the presentation of the algorithms and complexity results below, but is not strictly necessary. Indeed, any directed graph D with

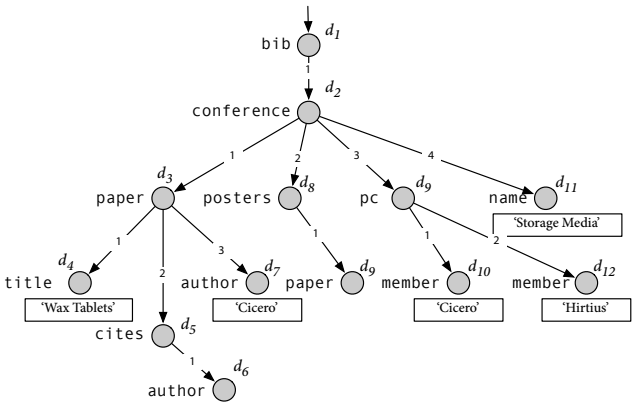

Figure 1: Exemplary Data Graph: Labeled Ordered Simple Directed Graph

multi-edges and loops can be represented as a simple directed graph D' by replacing each edge $(n_1, i, n_2) \in E$ with a new node $n \in N' \backslash N$ and edges (n_1, i, n), $(n, 1, n_2) \in E'$. Obviously, D' has by the number of edges in D more nodes than D: $|N'| = |N| + |E|$ and $|E'| = 2\,|E|$.

D is an *ordered* graph, i.e., the order of the children of a node is significant. Since the order is relative to the parent and a single node can be child of several parents, the order is associated with the edge rather than with the child node. It is assumed, that each child has a unique position in the order of its siblings $(p, i, c), (p, i', c') \in E$: $i = i' \implies c = c'$. On simple graphs this induces a proper order relation among siblings, but only on trees an obvious generalization to an order over all nodes (like document order in XML) exists.

Two *labeling functions* are provided, viz. $\mathcal{L}$ and $\mathcal{V}$. The first associates conventional node labels with each node, the second "content" values. The difference is made to be able to distinguish the cost of comparing two labels vs. two content values. Furthermore, $\mathcal{L}$ is assumed to be total, whereas $\mathcal{V}$ may be undefined for some nodes in the graph. In Figure 1, an exemplary data graph is shown. Labels are denoted to the left of the node, "content" values in boxes under the nodes. A root node is indicated by an incoming arrow.

The definition allows *multiple root nodes*, e.g., if there are several connected components in the graph. Any node may be a root node. In particular root nodes may, in contrast to usual rooted graph models, have parents. Intuitively, root nodes are simply highlighted "entrance" points into the graph that can be chosen arbitrarily when defining the data graph: If the data graph is a single XML document there will be a single such root node, however this formalization also covers collections of XML documents (as in XQuery) and RDF graphs where, e.g., each subject node can be considered a root node. In the following, we assume without loss of generality that a data graph has a single root and is connected. This ensures that $|E| \geq |N| - 1$, and thus $O(|E| + |N|) = O(|E|)$.

Specificities of XML, RDF, and Topic Maps such as attributes, namespaces, containers, collections, etc. are not provided by the basic data model. Handling these specificities poses no challenge to the evaluation algorithm and complexity analysis in this article and is therefore disregarded here.

2.2 Conjunctive Queries

Conjunctive queries are a convenient and relevant formalization of the query core of many XML and RDF query languages such as XSLT, XQuery, SPARQL, and Xcerpt.

Query syntax. A conjunctive query consists of a query *head* and a query *body*. The query body is a conjunction of atoms, and each atom is a relation over query variables. The domain of the query variables are the nodes N of the data graph D the query is evaluated against. The query head is a list of answer variable bindings which form an answer to the query. All answer variables must also occur in the body of the query. All other variables in the query body are existentially quantified [1].

In this article, only *binary and unary relations* are considered in conjunctive queries (though Section 6 briefly discusses an extension for handling order relations on graph data that uses *ternary* relations). Thus, conjunctive queries follow the following grammar:

$$\langle query \rangle ::= \langle label \rangle \text{ `(' } \langle variable \rangle \text{ (`,' } \langle variable \rangle)^* \text{ `)'}$$
$$\text{`}\leftarrow\text{'} \langle atom \rangle \text{ (`,' } \langle atom \rangle)^*$$

$$\langle atom \rangle ::= \langle relation \rangle \text{ `(' } \langle variable \rangle \text{ (`,' } \langle variable \rangle)? \text{ `)'}$$

Query Relations. Three types of relations may occur in conjunctive queries: unary "property" relations that restrict the valuation to nodes with a certain property, binary "structural" relations that require pairs of nodes to stand in the queried data graph in a certain structural relation, and binary "join" relations that compare nodes based on some property.

The proposed algorithms and complexity considerations apply to arbitrary property, structural, and join relations as long as for given nodes, each property relation can be checked in constant time, each structural relation in $O(|E|)$, and each join relation in at most $O(j(|E|))$ for some polynomial j. Additionally, the enumeration of the structurally related nodes for a given node n must be possible in $O(|E|)$. For join relations, a parameterized complexity is used to allow constant-time value joins based, e.g., on a fixed set of key values as well as deep-equal or "string value" joins as in XPath, that access the entire sub-structure rooted at a node and require in worst-case $O(|E|)$ time to be checked on a pair of nodes.

In the remainder of this article, we use the *property* relation ROOT, which is satisfied only by the root nodes of the queried data graph, as well as label relations LABEL_σ for all $\sigma \in \Sigma_L$ (i.e., for all possible labels) that restrict to nodes v where $\mathcal{L}(v) = \sigma$.

As *structural* relations only CHILD and its closures CHILD^+ and CHILD^* are considered. An extension to regular path expressions (or conditional axes [19]) is straightforward, as a regular path expression can be checked with data complexity $O(|E|)$ for two given nodes. In Section 6, an extension with (ternary) sibling-order relations is briefly outlined.

The *join* relations used are IDENT and VALEQUAL. IDENT is the identity relation, VALEQUAL is defined over the value labeling function $\mathcal{V}$. IDENT can be used to rewrite queries with structural relations that form a graph to queries with structural relations that form a tree but contain additional VALEQUAL atoms.

Figure 2 shows the query graphs for two conjunctive queries. This intuitive representation of graphs is used through-

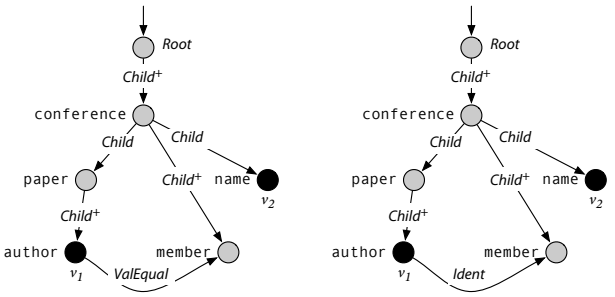

Figure 2: Exemplary Query Graphs (left: Q_1 with value join, right: Q_2 with identity join)

$$
\begin{array}{ll}
[\![\, q(v_1,\ldots,v_n) \leftarrow atom_1,\ldots,atom_m \,]\!]^q_D & \\
\quad = \pi_{v_1,\ldots,v_n}([\![\, atom_1 \,]\!]^q_D \cap \ldots \cap [\![\, atom_m \,]\!]^q_D) & \\
[\![\, unary(x) \,]\!]^q_D & = \{t \in N^q : t[x] \in [\![\, unary \,]\!]_D\} \\
[\![\, binary(x,x') \,]\!]^q_D & = \{t \in N^q : (t[x], t[x']) \in [\![\, binary \,]\!]_D\} \\
[\![\, \textsc{root} \,]\!]_D & = R \\
[\![\, \textsc{label}_\sigma \,]\!]_D & = \{n \in N : \mathcal{L}(n) = \sigma\} \\
[\![\, \textsc{child} \,]\!]_D & = \{(n,n') \in N^2 : \exists i \in \mathbb{N} : (n,i,n') \in E\} \\
[\![\, \textsc{child}^+ \,]\!]_D & = \bigcup_{i>0} ([\![\, \textsc{child} \,]\!]_D)^i \\
[\![\, \textsc{child}^* \,]\!]_D & = \bigcup_{i\geq 0} ([\![\, \textsc{child} \,]\!]_D)^i \\
[\![\, \textsc{ident} \,]\!]_D & = \{(d,d) \in N^2\} \\
[\![\, \textsc{valequal} \,]\!]_D & = \{(d,d') \in N^2 : \mathcal{V}(d) = \mathcal{V}(d')\}
\end{array}
$$

Table 2: Semantics of n-ary Conjunctive Queries over Data Graphs

out this paper: The representation of labels and values, as well as root nodes is as in data graphs, but edges are annotated with structural or join relations. Answer variables are marked by the name of the variable and a black instead of a gray node.

The two queries both select article authors together with article titles and conference names if the author is also a program committee member at the conference. However, the first one uses a value, the second an identity join. Thus, the first matches also if just the values are the same but different nodes are used for the representation of the article author and the program committee (this is a common way to model such data in XML), the second only matches if identical nodes are used (a common way to model the data in RDF where globally unique identifiers for entities such as persons are available).

In the remainder of this paper, we assume w.l.o.g. that every query node (i.e., variable) is reachable from a root query node by a path of structural relations.

Query semantics. Let D be the data graph the query is evaluated against and q the number of variables occurring in Q, then Table 2 gives the precise semantics of n-ary conjunctive queries over graphs as used in this article. The semantics is defined based on *sets of valuations* for query variables. A valuation t for n variables is an n-ary tuple with one column for each of the variables. We use $t[v]$ to denote the binding of variable v in the valuation t, and $[v_1 : n_1, \ldots v_q : n_q]$ for the construction of a q-ary tuple. We also allow an empty tuple []; the combination of the relation $\{[]\}$ with a relation R by cartesian is defined as $R = \{[]\} \times R$.

Query classes. We distinguish graph, tree, and path queries. Tree queries are queries whose graphs are tree-shaped. Analogously, path queries are queries whose graphs are in fact single paths. In addition to these basic classes, we introduce the class of *structural tree queries*. These are queries where the query without join relations is a tree. There may be additional arbitrary join relations, so that the complete graph may be no tree. Both of the example query graphs from Figure 2 are structural tree queries, but no proper tree queries.

Although it is known that evaluating graph queries is NP-complete even against tree data [15], the need for graph queries is evident. It is not only often necessary to retrieve more than one value but to compare the retrieved values with each other. Whenever the number of binary relations equals or exceeds the number of variables, the query is no tree query, since a tree with q nodes has $q-1$ edges and thus a tree query with n variables has $n-1$ structural relations. Indeed, many queries formulated in XQuery, XLST, or XPath fall into the class of graph queries [7].

Graph-shaped conjunctive queries may have multiple root or source variables, i.e., variables that occur only as source in structural relations, but not as sink. Let *SourceVars*(Q) be the set of such variables in the query Q. As for data graphs, we assume in the following w.l.o.g. that there is exactly one such variable in each query, i.e., that all query graphs are rooted. We use *FreeVars*(Q) to reference the answer variables in Q. We write $a \in Q$ to test for the occurrence of an atom a in Q, and $Q \backslash \{a_1, \ldots, a_n\}$ to remove a set of atoms $a_1, \ldots, a_n$ from Q. For brevity, we use if unambiguous in the context also $Q \backslash \{v_1, \ldots, v_n\}$ to indicate the conjunctive query Q' that contains all atoms from Q except those involving variables $v_1, \ldots, v_n$.

Note that (rooted) graph queries can be transformed into structural tree queries by replacing non-tree structural relations with identity joins: First, compute a spanning tree, considering the structural relation edges only. For each non-tree edge representing a structural relation REL between variables x and y, take a fresh variable y'. Replace the edge representing $\textsc{rel}(x,y)$ by the tree edge $\textsc{rel}(x,y')$, and add $\textsc{ident}(y,y')$ to the query Q. The size of the query increases by the number of non-tree edges, which is linear in the size of Q, but quadratic in the number of variables in the query.

For a rooted query Q, we denote a spanning tree of Q, consisting of only structural relations, with $T(Q)$. Based on this spanning tree, *JoinVars*(Q) denotes the set of existentially quantified variables contained in an atom that labels a non-tree edge.

3. MEMOIZATION MATRIX

At the core of the evaluation technique detailed in this paper stands the "memoization matrix". It is a compact data structure holding intermediary results of the evaluation of an n-ary conjunctive query. A memoization matrix associates query nodes (i.e., variables) q with bindings $n \in N$ and *one* sub-matrix, containing for each query child node q' the compatible bindings $n' \in N$ for q' under the binding n for q.

DEFINITION 1. ***Memoization matrix.*** *Given a query Q with variables Vars(Q) and spanning tree $T(Q)$, and a data graph D with nodes N, a memoization matrix for the evaluation of Q against D is a recursive data structure rep-*

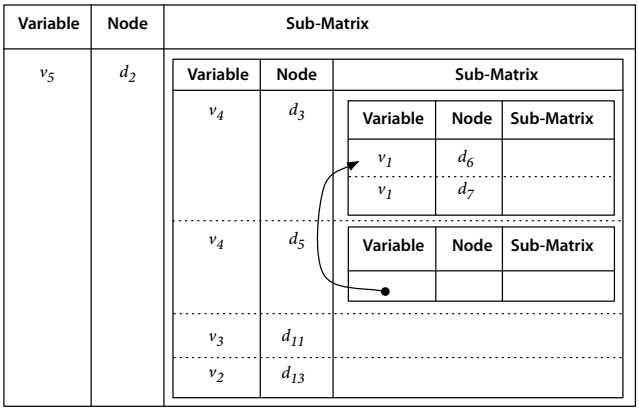

Figure 3: Filled Memoization Matrix (on Data of Figure 1)

resenting all possible *bindings of query variables in Q to nodes from D. The memoization matrix is a relation containing for each $q_s \in SourceVars(T(Q))$ and each possible binding $n \in N$ for q_s that satisfies all property relations on q_s one triple (q_s, n, M') with M' a subset of the memoization sub-matrix for $Q \backslash SourceVars(T(Q))$ such that for each tuple $(q', n', M'') \in M'$ and each atom $\text{REL}(q_s, q') \in T(Q)$, it holds that $(n, n') \in [\![\text{REL}]\!]_D$.*

Intuitively, this definition requires that the bindings for source variables in a sub-matrix M' are structurally compatible with the binding of the source variable in the corresponding tuple of M.

Notice that only the spanning tree of Q, denoted by $T(Q)$, is considered in the memoization matrix. The memoization matrix ensures only consistency w.r.t. relations within $T(Q)$. It does not ensure that the valuations are consistent w.r.t. relations outside $T(Q)$. Exploiting the tree shape of $T(Q)$, this makes a local evaluation of relations possible: A full-match can be incrementally computed from local matches that consider parent and child variables in the tree query in isolation.

The memoization matrix has been proposed first in a variant of which has been proposed first in [23]. Here, we refine this proposal with tuple sharing: To avoid multiple computations of matches in the case of queries where the same data node can be a match for a variable under different constellations of the remaining variable, the memoization matrix shares tuples where possible: Each tuple (q, n, M) exists only once and is referenced if the same tuple may occur in different sub-matrices. Notice that sharing of tuples only occurs between sub-matrices at the same level (i.e., sub-matrices of the same common super-matrix). The following sections show how this property can be ensured during the construction of the memoization matrix. Once more that this property relies on the tree structure of the relations checked in the memoization matrix.

It is furthermore assumed that the matrix is clustered by variables in order to allow linear time access to all entries relating to a variable.

Figure 3 shows the memoization matrix for the evaluation of the query from Figure 4 against the sample data graph from Figure 1.

The algorithms for matrix population discussed in the following section that guarantee a population of the matrix for a given n-ary conjunctive query Q against a data graph D takes at most $O(|Vars(Q)| \cdot |N| \cdot |E|)$ time, where $|Vars(Q)|$ denotes the number of variables in Q, $|N|$ the number of nodes, and $|E|$ the number of edges in the data graph D. Note that in the special case of tree-shaped data, $|E| = |N| - 1$, so that the worst case complexity becomes $O(|Vars(Q)| \cdot |N|^2)$. The size of the memoization matrix is in $O(Vars(Q) \cdot |N|^2)$ independently from the used algorithm, just by assuming sharing of submatrices, as demonstrated in the following.

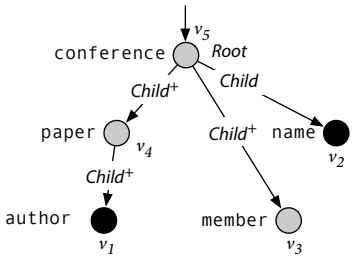

Figure 4: Modified Exemplary Query

LEMMA 1 (SIZE OF MEMOIZATION MATRIX). *The size of the memoization matrix M for a query Q and a data graph D with nodes N is bounded by $(2q-1) \cdot v^2$, where $q = |Vars(Q)|$, and $v = |N|$.*

PROOF. By structural induction over $T(Q)$.
Query leaves: It holds that $q = 1$, and obviously the number of valuations for a single variable is bounded by v. The size of the memoization matrix is $q \cdot v \leq (2q-1) \cdot v^2$.
Inner query nodes: Let the inner query node i have c children. It holds that the sum of nodes of all child queries is equal to $q - 1 = \sum_{j=1}^{c} q_j \quad (*)$. There are again at most v valuations of i. As tuples are shared over parent matrices, there is at most *one* tuple for each such valuation. The size of the sub-matrix contained in the tuple itself is bounded by $c \cdot v$, as each child has at most v assignments. The size of all tuples for the inner node i (i.e. of the complete sub-matrix of i) is hence $c \cdot v^2$. The overall matrix size is

$$\sum_{j=1}^{c}(2q_i - 1) \cdot v^2 + c \cdot v^2 \overset{(*)}{=} (2(q-1) - c + c) \cdot v^2 \leq (2q-1) \cdot v^2,$$

using the induction hypothesis. □

Based on the populated matrix, the algorithms discussed in Section 5 traverse the memoization matrix, enforce the remaining (non-hierarchical) relations, if there are any, and create the output according to the query semantics introduced above.

4. MATRIX POPULATION

The compact memoization matrix introduced in the last section can be produced bottom-up ($\mathsf{Match}_{\uparrow}$, Section 4.1) or top-down ($\mathsf{Match}_{\downarrow}$, Section 4.2), that is, starting with the root variable and the root data node or with the leaf variables and all data nodes. While both algorithms have the same worst case complexity, experimental evaluation in Section 7 shows that an in-memory implementation of the bottom-up algorithm has an experimental runtime close to the worst case complexity, while the top-down approach displays far better runtime behavior in realistic cases.

4.1 Bottom-Up Approach

The bottom-up approach ($\mathsf{Match}_\uparrow$) is a bulk-processing approach often employed in secondary-storage databases. It starts by matching the leaf variables of $T(Q)$ with all nodes of D, and uses these results to successively fill the domains of variables that have a common structural relation with these leaf variables. This process is repeated iteratively until either a variable domain becomes empty, indicating that the query has no matches, or the root variable of Q is reached, indicating that all matches of the query are found.

Algorithm 1 $\mathsf{Match}_\uparrow(Q, D)$

```
 1: q ← |Vars(Q)|; V_Q ← Vars(Q); N ← nodes(D); ρ ← ∅
 2: root ∈ SourceVars(Q)
 3: while ρ(root) = ∅ do
 4:    take x ∈ V_Q : ρ(x) = ∅ ∧
                ∀REL(x, x') ∈ T(Q) : ρ(x') ≠ ∅
 5:    M ← ∅
 6:    for all n ∈ N do
 7:       if ∃REL(x) ∈ Q : n ∉ [[ REL ]]_D then
 8:          continue n
 9:       M_S ← ∅
10:       for all REL(x, x') ∈ T(Q) do
11:          M_R ← ∅
12:          for all (x', n', M') ∈ ρ(x') do
13:             if (n, n') ∈ [[ REL ]]_D then
14:                M_R ← M_R ∪ {(x', n', M')}
15:          if M_R = ∅ then
16:             continue n
17:          M_S ← M_S ∪ M_R
18:       M ← M ∪ {(x, n, M_S)}
19:    ρ(x) ← M
20:    if ρ(x) = ∅ then
21:       return ∅
22: return ρ(root)
```

The algorithm uses a helper function ρ to associate variables with sets of tuples representing bindings for these variables. ρ is initially empty and populated step by step in the outer **while** loop: Starting with the leaf nodes, the algorithm generates the set of tuples $\rho(x)$ (l. 3 and 19) for each variable x, until either no match is found for a variable and thus the query fails (returns an empty set) (l. 20–21) or the root node has been processed (l. 3) and the memoization matrix for the root node is returned (l. 22).

When processing a variable x (l. 3–21), the algorithm creates an empty memoization matrix for that variable and iterates over each node n in the nodes N of D (l. 6–18) verifying first the unary relations (l. 7–8). If any of the relations fails on n, the next node is tested. If it succeeds, a sub-matrix M_S is initialized (l. 9). For every variable x' related to x via a structural relation in the spanning tree $T(Q)$ (l. 10) only the tuples with nodes n' that actually satisfy the relation with n are included in the sub-matrix (l.12–14). A temporary matrix M_R for each relation is used to test whether there is no matching node pair for that relation. If so, the current data node is skipped (l. 11 and 15–16). Only if the matrix M for x is empty at the end of the processing of x, the query fails as there are no compatible bindings for x in the data.

Notice, that the algorithm does not specify the details of row sharing between matrices at the same level. It is assumed that in l. 14 and l. 18 pointers to M', resp. M are used instead of copies.

THEOREM 1 (COMPLEXITY OF $\mathsf{MATCH}_\uparrow$). *Let $v = |N|$, $q = |Vars(Q)|$, and $e = |E|$. Then, $\mathsf{Match}_\uparrow$ is in $O(q \cdot v \cdot e)$ combined time complexity and in $O(q \cdot v^2)$ combined space complexity.*

PROOF. There are q variables, so that the outer loop (l. 3) is bounded by q. The loop over all nodes (l. 6) is bounded by v. The verification of the property relations takes constant time, as there is a fixed number of such relations in the language and each test (such as a label test) is assumed to be constant (l. 7–8). Since $T(Q)$ is a spanning *tree* there are at most $q-1$ structural relations in that tree that need to be tested in the iteration over all binary relations (l. 10). As each binary relation is visited only once (when the source variable of that relation is processed), the loop body (l. 11–17) is executed $(q-1)\cdot v$ times. Since at most all nodes in the document can match with a variable, the iteration in l. 12 is bounded by v. As the verification of $(n, n') \in [\![\text{REL}]\!]_D$ is required to be in $O(e)$ for structural relations (cf. Section 2), the overall time complexity is in $O(q \cdot v^2 \cdot e)$, if the structural relations are verified for each pair (n, n'). However, we can assume that the structural relations are precomputed in an index structure such as a bit array which provides constant verification of the structural relation at a space cost of $O(v^2)$ and time cost $O(v \cdot e)$. This is an acceptable trade-off as there are usually only a small number of structural relations and as the memoization matrix already requires $O(q \cdot v^2)$ space as discussed above. Under this assumption, the overall combined time complexity becomes $O(q \cdot v \cdot e)$. The overall space complexity is dominated by the size of the memoization matrix $O(q \cdot v^2)$, as the only helper structures are the precomputed relations at $O(v^2)$ and ρ which is bounded by q. □

Even though the bottom-up approach has a nice upper bound of computational complexity, it needs further refinements to be usable in practice as the experimental evaluation in Section 7 demonstrates. To obtain a practically useful performance, the bottom-up algorithm needs efficient index structures on the property relations occurring in the query. Examples of such index structures are so-called streaming schemes [8]. A further performance increase might result from evaluating groups of structural and property relations at once using holistic tree queries, cf. [5, 18]. The benefits of the latter approach are not clear for n-ary graph queries, where most query variables are either answer variables or involved in non-structural joins, preventing large groups of relations that can be evaluated at once using [18] or similar approaches. Further investigation of the use of such holistic schemes for n-ary conjunctive queries with graph-shape is required, but out of the scope of this paper.

4.2 Top-Down Approach

The runtime of the bottom-up approach can be very close to its worst case complexity if the query leafs are not selective enough or if efficient evaluation of property relations through indices is not available. For an in-memory evaluation of n-ary conjunctive queries without indices, the top-down approach matching the query from the root to the leafs and restricting the number of candidate nodes primarily based on query structure presents a feasible and often superior alternative. Furthermore, the top-down algorithm does not need any adjacency index to guarantee a runtime in

$O(q \cdot v \cdot e)$. However, iteration over the range of a structural relation for one data node must be guaranteed in $O(e)$ time. This assumption holds for any structural relation occurring in XPath, XSLT, XQuery, SPARQL, or Xcerpt.

Like the bottom-up algorithm, the top-down algorithm needs an additional helper structure ρ. However, in this case it associates tuples of query variable *and* data node to entire sub-matrices. Constant access is assumed for this structure by basing it on a two-dimensional array. It is assumed that $\rho = \emptyset$ at the first call of the algorithm. Furthermore, an explicit "no match" indicator $\perp$ is used to mark combinations of nodes and variables that are certain not to match. This must be distinguished from the case where the combination has not yet been computed and the case where there is no sub-matrix for the combination (i.e., the variable is a leaf in the query).

Algorithm 2 $\mathsf{Match}_{\downarrow}(x, n)$

```
 1: if ρ(x,n) = ⊥ then
 2:   return ∅
 3: if ρ(x,n) defined then
 4:   return {(x,n,ρ(x,n))}
 5: if ∃REL(x) ∈ Q : n ∉ [[ REL ]]_D then
 6:   ρ(x,n) ← ⊥
 7:   return ∅
 8: M_S ← ∅
 9: for all REL(x,x') ∈ T(Q) do
10:   M_R ← ∅
11:   for all n' ∈ N : (n,n') ∈ [[ REL ]]_D do
12:     M_R ← M_R ∪ Match(x',n')
13:   if M_R = ∅ then
14:     ρ(x,n) ← ⊥
15:     return ∅
16:   M_S ← M_S ∪ M_R
17: ρ(x,n) ← M_S
18: return {(x,n,M_S)}
```

The top-down algorithm is a typical recursive descent over the query structure. It has two parameters, a query and a data node, and computes the memoization matrix for these two nodes. If called with the root of the query Q and the root of the data graph D, the result is the memoization matrix for the evaluation of Q against D.

The algorithm $\mathsf{Match}_{\downarrow}$ operates on pairs (x, n) of variables and data nodes, and a matching is computed at most once for each combination of query and data node. Lines 1–4 verify whether a matrix for the given pair of variable and value has already been computed. If this is the case, the call immediately returns. The unary relations of the input variable x are verified on the data node n (l. 5). If any of the unary relations fails, the matching fails, stores the information that the pair is incompatible in ρ and returns (l. 6–7). If n satisfies all unary relations, the sub-matrix M_S of the entry is initialized (l. 8) and $\mathsf{Match}_{\downarrow}$ iterates over all structural relations in the spanning tree $T(Q)$ with x as source variable. It creates a temporary matrix M_R for each such relation (l. 10). M_R is filled with the result of recursive calls to $\mathsf{Match}_{\downarrow}$ (l. 12) with any data node n' that satisfies the structural relation REL together with n. If the matching fails, the result of the recursive call is an empty set, thus leaving M_R unchanged. Again, if for any structural relation in the spanning tree no matching data node is found and thus M_R is still empty in line 13, the matching of x with n fails (l. 13–15). At the end of the loop starting in line 9, the temporary matrix M_R is added to the sub-matrix M_S for x and n. Finally, if all structural relations for x are processed, the resulting matrix M_S is stored in $\rho(x, n)$, and a matrix with a single tuple (x, n, M_S) is returned representing the entry for x and n.

THEOREM 2 (COMPLEXITY OF $\mathsf{Match}_{\downarrow}$). *Let $v = |N|$, $q = |Vars(Q)|$, and $e = |E|$. Then, $\mathsf{Match}_{\downarrow}$ is in $O(q \cdot v \cdot e)$ combined time complexity and $O(q \cdot v^2)$ combined space complexity.*

PROOF. The use of matrix memoization (l.1–3, 14, 17) guarantees that $\mathsf{Match}_{\downarrow}$ is executed at most once for each combination of variable and data node (x, d). Testing unary predicates takes again constant time as argued in the proof of Theorem 1. As each of the $q - 1$ relations is visited at most once, the loop over all binary relations (l. 9) is visited at most $(q - 1) \cdot v$ times. The enumeration of all values of any structural relation is required to be in $O(e)$ and thus the set initialization of the inner loop (l. 11) takes time in $O(e)$. Since there are at most v elements in the range of any structural relation and the loop body (l. 12) is constant (the recursive call is amortized by memoization), $\mathsf{Match}_{\downarrow}$ is in $O(q \cdot v \cdot e)$ combined time complexity. The space complexity of the memoizatin matrix is $O(q \cdot v^2)$, and the size of ρ is in $O(q \cdot v)$, so that the space complexity of $\mathsf{Match}_{\downarrow}$ is again dominated by the memoizatin matrix. □

As section 7 shows, the algorithm $\mathsf{Match}_{\downarrow}$ is a competitive algorithm, exhibiting a linear time complexity in many real world scenarios, even without any index structures. Streaming schemes [8] and similar techniques could most likely be used to refine the algorithm further and speedup the average runtime. However, such optimizations are beyond the scope of this paper.

5. MATRIX CONSUMPTION

The consumption of a memoization matrix for the evaluation of a query Q against a data graph D creates the extensional representation of the result. That is to say, the compact in-memory result representation in the memoization matrix is *expanded* to a set of valuations for answer variables, i.e. a set of tuples associating answer variables with matching data nodes. This is comparable to the transformation of a non-first-normal-form relation into a flat relation, except for two details: Nested matrices consist of bindings for several relations and must hence be decomposed into partitions before the flattening takes place and a sub-matrix tuple can be referenced by more than one matrix. With the same reasoning as for sharing of sub-matrices, the results of the transformation from matrices to flat valuations must be memoized to avoid their repeated computation.

In contrast to matrix population, the algorithms for matrix consumption, though still agnostic to the shape of the data, have to treat tree and graph *queries* differently. This is necessary, because graph queries contain binary relations that are not verified by the matrix population algorithms. Obviously, there are no such remaining relations in tree queries. This reduces the matrix consumption algorithm to a simple flattening of the nested memoization matrix to produce the output. Since the output size of tree queries is guaranteed to be larger than every intermediate result, the time and space complexity of the consuming algorithm is

bounded by the result size. For graph-shaped queries, however, this is not the case: an intermediate result of exponential size can be created and only then be reduced through the remaining binary relations that are not part of the query spanning tree relations. Thus, even if the output size is sub-exponential, the matrix consumption for graph queries has exponential combined time complexity. To illustrate this, consider the queries from Figure 2: The memoization matrix only enforces the structural relations, but does not consider VALEQUAL and IDENT. These relations may reduce the result size considerably if they are applied.

In the following three sections, we first take a look at the basic matrix consumption for tree queries (Section 5.1) and compare it in a second step (Section 5.2) with the case for arbitrary graph queries. Finally, we outline briefly the benefits and drawbacks of a nested loop join for incremental output generation (Section 5.3).

5.1 Matrix Consumption for Tree Queries

To obtain the set of answers to a query Q against a data graph D according to the semantics of n-ary conjunctive queries in Section 2, the algorithm is called with the memoization matrix resulting from the application of one of the Match algorithms in Section 4. In the following algorithms, we rely on the fact that the memoization matrix for the root variable consists of one single entry, which is guaranteed by the matrix population algorithms. The matrix population algorithms are called with this single tuple (x, n, M) as parameter.

As in the top-down population algorithm, a helper structure ρ for memoizing the flattened relation for each pair of query and data node is used. It is intially assumed to be $\emptyset$.

Algorithm 3 $\mathsf{Output}_T(x, n, M)$

```
 1: if ρ(x,n) defined then
 2:    return ρ(x,n)
 3: if x ∈ FreeVars(Q) then
 4:    A ← {[x : n]}
 5: else
 6:    A ← {[]}
 7: for all x' ∈ π1(M) do
 8:    A_x' ← ∅
 9:    for all n', M' : (x', n', M') ∈ M do
10:       A_x ← A_x' ∪ Output_T(x', n', M')
11:    A ← A × A_x'
12: ρ(x,n) ← A
13: return A
```

The Output_T algorithm unfolds the compact solution representation to a set of valuations, eliminating duplicates (note the set union l. 10). The algorithm returns a set of valuations, each representing a match of the query in the data. Given a triple (x, n, M), it is first verified whether the answers have already been computed and memoized in ρ in an earlier call for (x, n), in which case they are immediately returned (l. 1–2). Otherwise, the answer set A is initialized accordingly to the type of x (l. 3–6). In the case that x is no free variable, bindings for x should not be part of the answer set. Thus the empty tuple is used as initial valuation. Matrix M is partitioned by variables (l. 7–9). Note that since the considered structural relations stem from a spanning tree, the variable x' determines also the structural relation of the partition. The result of each partition is stored into $A_{x'}$. The innermost loop (l. 6) iterates over each pair of data node n, and sub-matrix M' of the current partition. The result of the recursive calls to $\mathsf{Output}_T(x', n', M')$ are collected in $A_{x'}$. The result A is then multiplied by $A_{x'}$ to create the answer (l. 11). The answer set is a relation over the free variables of Q, as required by the semantics (cf. Section 2.2).

PROPOSITION 1 (COMPLEXITY OF Output_T). *The algorithm Output_T has $O(|Vars(Q)|\cdot|N|^2+|Q(D)|)$ time complexity where $|Q(D)|$ denotes the result size.*

PROOF SKETCH. Each matrix is transformed to an answer set with result size, and the intermediate sets can only get as large as the final results. The worst case scenario is that each recursive call to Output_T creates an answer set of result size. In this case, the elimination of duplicates takes $O(2\cdot|Q(D)|\cdot log(2\cdot|Q(D)|)) = O(|Q(D)|\cdot log\,|Q(D)|)$ using a sort-based elimination, and $O(|Q(D)|)$ using a nondegenerating hash-based duplicate elimination. As the complete matrix structure of size $O(|Vars(Q)|\cdot|N|^2)$ must be traversed to create the output, the worst case complexity becomes $O(|Vars(Q)|\cdot|N|^2+|Q(D)|)$. □

Duplicate bindings are generated by projection; as existentially quantified variables are dropped, equal bindings of the remaining variables are produced. The problem is the same as duplicate generation in relational databases, and it can be reasonably assumed that the same elimination techniques may be used (such as [3]).

Avoiding duplicate generation when evaluating tree queries against tree data has been studied extensively, e.g., in [16, 21]. Though such techniques are beyond the scope of this paper, Section 8 briefly addresses the support of regular path expressions that can drastically reduce the number of existential variables in a query and thus, in many queries, the need for duplicate elimination.

5.2 Matrix Consumption for Graph Queries

In [15] it is shown that the evaluation of graph-shaped conjunctive queries over the relations *Child* and *Child** is already NP-complete on tree data. Nevertheless, the need for graph-shaped queries is evident, especially when comparison relations such as VALEQUAL and EQUAL are supported. This is evidenced by the fact that queries in all of XSLT, XQuery, SPARQL, and Xcerpt make extensive use of graph-shaped queries and formalizations for graph-shaped queries can cover much larger sub-sets of these languages than those restricted to tree-shaped queries.

The matrix method applies to graph-shaped queries in the following way: First, a spanning tree $T(Q)$ over the structural relations of Q is computed offline. Second, $\mathsf{Match}_\downarrow$ or $\mathsf{Match}_\uparrow$ is applied to create the memoization matrix of the query problem. Finally, this memoization matrix is consumed with a new output algorithm, Output_G.

The new algorithm must verify whether the produced valuations satisfy all relations that are not part of the spanning tree $T(Q)$. To put it another way, the non-tree relations impose additional selection conditions on the produced valuations. These additional selection conditions are distributed over cartesian products that the consumption algorithm for tree queries computes. They are combined into joins, with possibly non-atomic conditions, if more than one relation must be verified in one cartesian product. Due to the nested structure of the memoization matrix, the output algorithm performs $q-2$ different kinds of cartesian products. As several sub-matrices of a given level share the same structure,

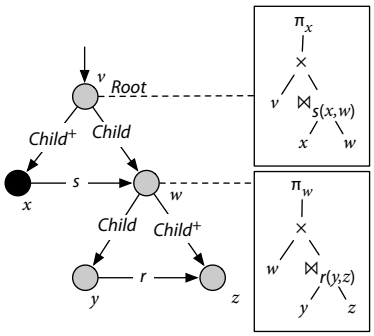

Figure 5: Exemplary Join and Projection Specification

each kind of cartesian product is performed several times for each sub-matrix.

Of course, these additional selections should be applied as soon as possible (i.e., pushed down) to keep intermediate results small. Since existentially quantified variables involved in join conditions must be kept until these joins are performed, it is furthermore necessary to infer the position at which each existentially quantified join variable can be projected away. Hence, a *join and projection specification* ⋈*–Π–spec* is associated with each variable. This specification defines which joins and which projections can be performed when outputting the results for x. It furthermore determines the ordering of joins and projections.

Since join order optimization is out of the scope of this paper, the output algorithm abstracts from these topics by assuming the existence of a specification ⋈*–Π–spec* for each variable, and of a function that applies these join specification to a set of valuation sets. Using a set of valuations instead of a canonical cartesian product allows to use joins instead of selections, increasing the performance of the output algorithm considerably. The join and projection specification is typically created by the query planner and can be executed by a conventional relational query engine.

Figure 5 shows an example of a join and projection specification. Recall, that *JoinVars*(Q) is the set of join variables, i.e., the set of variables that are existentially quantified (no answer variables) and occur in at least one binary relation that is not part of $T(Q)$.

Algorithm 4 Output$_G(x, n, M)$

```
1: if ρ(x,n) defined then
2:    return ρ(x,n)
3: if x ∈ FreeVars(Q) ∪ JoinVars(Q) then
4:    A_S ← {{[x : n]}}
5: else
6:    A_S ← {{[]}}
7: for all x' ∈ π_1(M) do
8:    A_x' ← ∅
9:    for all n', M' : (x', n', M') ∈ M do
10:      A_x' ← A_x' ∪ Output_T(x', n', M')
11:   A_S ← A_S ∪ {A_x'}
12: A ←apply ⋈–Π–spec(x) to A_S
13: ρ(x,n) ← A
14: return A
```

The new algorithm however exhibits exponential worst case runtime in that it may perform at worst $q-3$ cartesian products without any selection based on non-tree edges (q being again $q = |Vars(Q)|$). In this case, the size and time complexity are both in $O(|N|^q)$, as the output algorithm keeps the set of valuations in memory.

Furthermore, the cost of value-based joins that are assessed with a cost function $j(|N|)$ must be considered. The worst case estimation is as follows: as every variable can be involved in a join, there are at most $q-1$ value-based joins (as equality is transitive, a query with more than $q-1$ joins can be transformed into an equivalent query with $q-1$ joins). Furthermore, every tuple of an exponential sized intermediate result is joined with each value-based join. As the application of a join reduces the result size by a factor at least linear in $|N|$, the overall runtime can be aproximated as $O(\sum_{i=2}^{q} j(|N|) \cdot |N|^i) = O(j(|N|) \cdot |N|^q)$.

Proposition 2 (Complexity of Output$_G$). *The algorithm* Output$_G$ *has* $O(j(|N|) \cdot |N|^q)$ *time complexity and* $O(|N|^q)$ *space complexity.*

Creating a structural tree query from a graph query is unfavorable for this worst case complexity, since it is exponential in the number of variables and the corresponding structural tree query with value joins for a graph query has up to twice the number of variables as the graph query. For realistic cases however, this is a technique to transform tree-relation join conditions that are not verifiable in constant time into identity joins. Alternatively, it is possible in the match algorithms to create (in the top-down approach reasonably small) on-the-fly indexes for the non-tree structural relations, assuring a fast verification of these relations in Output. The quadratic increase of the exponential factor can hence be avoided.

5.3 Incremental Matrix Consumption for Trees and Graphs

The previous two algorithms are tailored to provide an in-memory representation of all answers of a query and are thus both in time and space complexity bound by the output size. An in-memory representation of the answers is useful to perform further processing based on the answers, e.g., for structural grouping, aggregation, or ordering. However, in many cases an incremental output of the answers is preferable, in particular if further processing can also be realized in an incremental manner. Incremental answer generation can be realized using the algorithm Output$_{NLJ}$, a slightly modified incremental *nested loop join* over the memoization matrix. The algorithm uses the structure of the matrix instead of join attributes, but is otherwise – leaving aside partitioning issues – a standard nested loop join and therefore omitted here for space reasons.

Proposition 3 (Complexity of Output$_{NLJ}$). *The algorithm* Output$_{NLJ}$ *has time complexity* $O(|N|^q)$ *and space complexity* $O(q \cdot |N|^2)$ *on tree queries, on graph queries time complexity* $O(j(|N|) \cdot |N|^q)$ *and space complexity* $O(q \cdot |N|^2)$.

The advantage of Output$_{NLP}$ is the low space complexity that is essentially bounded by the size of the memoization matrix. However, this advantage is paid for by an exponential time complexity in almost all cases. Furthermore, this exponential time complexity is reached in many practical cases, making this algorithm suitable only for cases where space consumption is clearly more important than run time of the evaluation algorithm.

$[\![\text{NEXT}]\!]_D = \{(c, c') : \exists (p, i, c), (p, i', c') \in E : i + 1 = i'\}$
$[\![\text{NEXT}^+]\!]_D = \{(c, c') : \exists (p, i, c), (p, i', c') \in E : i < i'\}$
$[\![\text{NEXT}^*]\!]_D = \{(c, c') : \exists (p, i, c), (p, i', c') \in E : i \leq i'\}$

Table 3: Semantics of Order Relations

6. ORDER RELATIONS ON GRAPH DATA

In the previous sections, graph-shaped data is considered equivalent to tree-shaped data w.r.t. query evaluation. Although the worst case complexity of the matrix population algorithms is the same for both cases, the order of two sibling nodes becomes context dependent in graphs, cf. Section 2.1.

For the support of ordered queries in graph data, a ternary relation NEXT and its closures NEXT^+, NEXT^* are introduced (cf. Table 3). In fact, once the matrix consuming algorithms support join conditions, the handling of the ternary order relations is simple: it can be handled as additional join condition in the join and projection specification of each node (cf. Figure 6).

Besides this very rudimentary exploit of order relations as join conditions in the output algorithm, it is possible to take advantage of them in the matrix population algorithms, if the edge position from the data model is accessible. Assuming that $Next^*(x, y, y')$ must hold and that I is the set of positions of all y valuations and I' of y' respectively, it must obviously hold that $\forall i \in I : i \leq max(I')$ and $\forall i' \in I' : min(I) \leq i'$. If the domain of y is populated before y' (as the match algorithm does in its recursive descent), these conditions can be used to reduce the number of candidate valuations for y', and prune valuations for y: When populating the domain of y', the minimum position of valuations of y is known, so that the condition can be imposed on all valuations of y'. After populating y', all valuations of y that have a position greater than the maximum position of y' valuations can be dropped.

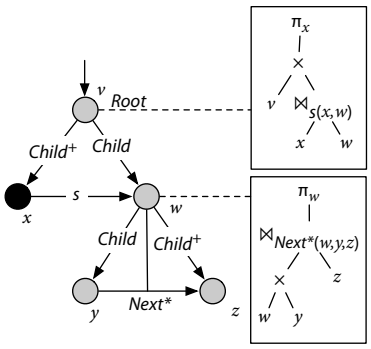

Figure 6: Join and Projection Specification with Order Relation

7. EXPERIMENTAL EVALUATION

The experimental evaluation is based on both synthetic and real data. The set of structural relations is extended by the additional relation ATTRIBUTE in order to support attribute queries. The tests have been executed on an AMD Athlon 2400XP machine with 1GB main memory. The algorithms are implemented in Java executed on JVM version 1.5.0_06-b05. All tests show the processing time without data parsing. Each measurement is averaged over 500 runs. The algorithms have been implemented straightforwardly close to the form presented without additional optimizations.

Synthetic data is used to demonstrate and confirm the complexity of the presented algorithms. The real data scenarios stem from the University of Washington XMLData repository[1], and demonstrate the competitiveness of the algorithms in their basic form.

The first experiment confirms on synthetic data that the memoization of intermediary results is essential, not only for the complexity but also for the experimental query evaluation time. The $\mathsf{Match}_\downarrow$ algorithm without memoization of variable domains (i.e., the helper structure ρ and all access to it is removed from the algorithm) exhibits an exponential growth of time consumption in the size of the query (cf. Figure 7), because several common sub-matrices are built repeatedly. Though the exponential growth of the output size can be a factor, the query used in these experiments is unary and produces a linear output. In contrast, Figure 8 depicts the effect of increasing arity in a worst-case scenario, where the query is entirely unrestrictive and a binding for one answer variable is related to all bindings of all other variables.

Figures 9 and 10 show a *comparison between the two approaches* for matrix population discussed in this article. A path query consisting of four variables and CHILD^* (descendant) relations only, but without label restrictions, is used. This query exhibits worst case complexity for the top-down algorithm $\mathsf{Match}_\downarrow$, as the match context is never restricted by a previous context. As expected, the plot shows a quadratic runtime growth in the data size for the top-down algorithm. The bottom-up approach exhibits, as expected, a cubic runtime when lacking index structures for the CHILD^* relation, whereas the runtime of the bottom-up algorithm with CHILD^* index and top-down is almost entirely identical (to the extent that the two plots are nearly indistinguishable in Figure 10).

The above experiments clearly reinforce the theoretical complexity results derived in this work. Moreover, at least the top-down algorithm performs quite well even in its basic form discussed here in real query scenarios. Figure 11 shows how the runtime of the top-down algorithm scales with the data size for path-, tree-, and graph-shaped queries. These queries are executed against the MONDIAL database of geographical information, cf. `http://www.dbis.informatik.uni-goettingen.de/Mondial/`. The plot shows additionally that already for path queries the bottom-up algorithm exhibits polynomial runtime; the naive bottom-up approach has an average runtime that is very close to its worst-case. On the other hand, the $\mathsf{Match}_\downarrow$ exhibits a linear runtime in all queries, even in the graph query case. This shows manifestly the power of the context-aware processing of a top-down approach.

The final test on increasing fragments of a large XML document (the Nasa dataset from the above mentioned repository) shows that the runtime of $\mathsf{Match}_\downarrow$ scales nicely with the data size and is very competitive even in the basic form implemented for these experiments.

[1] `http://www.cs.washington.edu/research/xmldatasets/`

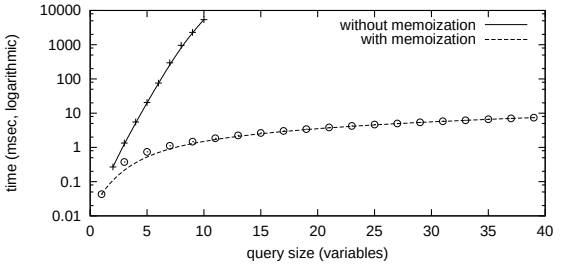

Figure 7: Effect of Memoization over Query Size

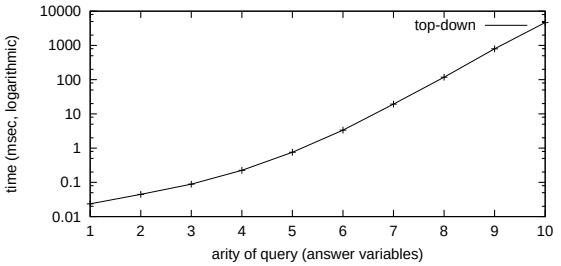

Figure 8: Worst-Case Effect of Query Arity

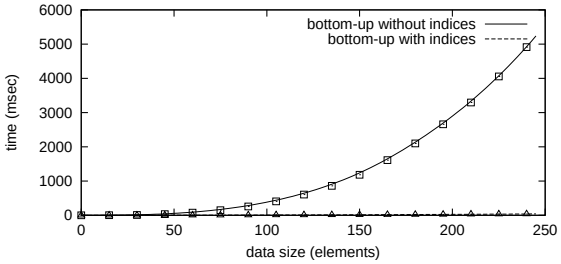

Figure 9: Comparison of Bottom-Up with and without Relation Indices

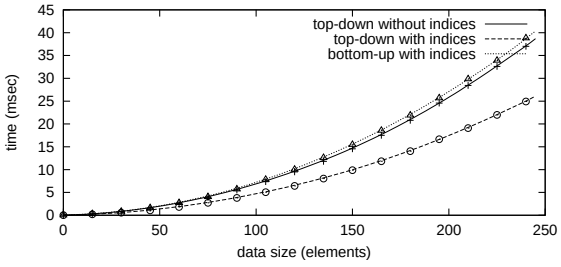

Figure 10: Comparison of Top-Down and Bottom-Up Approach

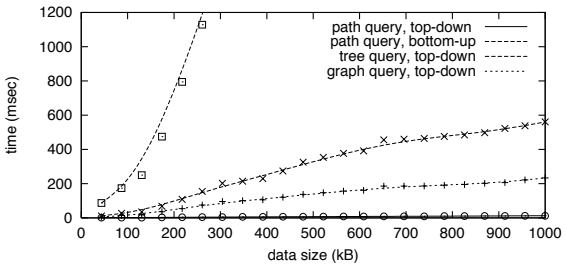

Figure 11: Path, Tree, and Graph Queries over Real-life Data

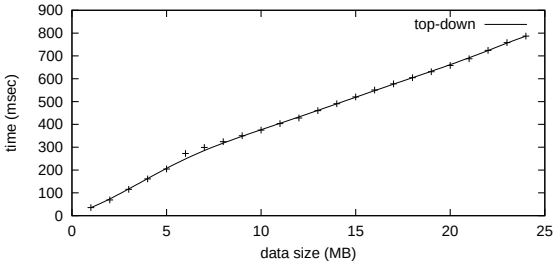

Figure 12: Top-Down Algorithm over Large, Real-life Data

8. EXTENSIONS AND OUTLOOK

Though the experimental evaluation shows that even the basic form of the proposed top-down algorithm performs nicely, there are quite a number of extensions and further optimizations likely to give interesting results: First of all, there are extensions of the top-down matching algorithm to a *complete unification algorithm*, needed for pattern matching as in Xcerpt [24]. This algorithm must handle negated and optional query parts as in general tree patterns [9].

Arc consistency, as used in constraint solving algorithms, can be used to reduce the size of the matrix structure. First experiments have shown, however, that verification of arc consistency does not always improve evaluation time: in cases where the runtime of $\mathsf{Match}_{\downarrow}$ is linear in the data size, applying arc consistency renders the runtime quadratic. This effect arises when the number of actually performed joins is reduced drastically through the query context.

Above this, *partial unnesting of matrices* can be used to remove existentially quantified variables eagerly at matrix population time: a link to and from an existentially quantified variable valuation is replaced by a direct link. In this way, the space complexity of path queries can be reduced from $O(|Vars(Q)| \cdot |N|^2)$ to $O(n \cdot |N|^2)$, n being the arity of the query. One step in this direction is the support of more expressive structural relations, e.g., conditional axes [19] that allow collapsing entire paths both for population and consumption of the matrix. Indeed, [20] uses a similar approach for the computation of complete answer aggregates over tree data.

Relational *query planning* and *execution* techniques could be incorporated to improve duplicate elimination [3] and to optimize the partitioning of relations between spanning tree and join specifications as well as the choice of an efficient join specification.

Finally, we plan to investigate a *combination* of bottom-up and top-down matching techniques, in order to combine the benefits of both a sophisticated bottom-up approach, i.e., early pruning in the case of selective query leaves, and the contextual narrowing of a top-down approach.

9. RELATED WORK

As previously mentioned, the complexity of conjunctive queries and monadic queries *over trees* is studied thoroughly in [15, 12]. A restriction of the bottom-up algorithm discussed in this article to conjunctive *tree queries* is roughly similar to the complete answer aggregates algorithm of [20] and has the same complexity.

Matching conjunctive queries against trees and graphs can be seen as a constraint solving problem. It is well established

that tree-shaped constraint problems (i.e., tree queries) can be solved in $O(q \cdot v^2)$ [11] where q is the number of variables and v the variable domain size. This result assumes, however, $O(1)$ verification time for all relations. Furthermore, the implication from arc to global constraint consistency used in this result, does not hold for graph-shaped constraint problems.

In [15] it is shown that there are special cases where arc consistency is at least sufficient to retrieve one single consistent solution: if all binary constraints have the $\underline{X}$-property (read: X-underbar) over an order $<$, arc-consistency is sufficient to guarantee that the minimal solution (in terms of the same ordering $<$) is consistent. It follows that the evaluation of n-ary graph queries with $\underline{X}$-relations is only exponential in the number of free variables. It can further be derived that the general problem is NP-complete and thus an algorithm as proposed here with worst case exponential runtime is the best achievable to present knowledge.

Another field of important related work are structural indexing techniques [5, 8, 18]. Indexes are an orthogonal aspect to the matrix method that can be used to improve the runtime of the presented algorithms. Considering entire paths or trees at once through physical operators such as twig joins [5] is a promising and widely researched technique for tree data. Their application to the discussed algorithms is straightforward. However, most structural indexing techniques do not exhibit obvious extensions to graph data.

10. CONCLUSION

With the rise of XML and RDF, processing both graph- and tree-shaped n-ary queries against semi-structured data becomes ever more important. The memoization matrix is a compact recursive non-redundant data structure that holds the solution sets to such queries. In case of tree queries, it contains the exact solutions to the queries, whereas in case of graph queries intermediary results: the solutions to the query represented by a spanning tree chosen for the population of the matrix. By separating the evaluation of n-ary queries in two phases, viz. the population and the consumption of the memoization matrix, we demonstrate several insights in tree- and graph-shaped query evaluation: **(1)** We show that the shape of the data, whether tree or graph, does for the most part not affect the query complexity for n-ary conjunctive queries. **(2)** We show that a unified algorithm for both tree- and graph-shaped semi-structured queries is feasible and both in worst-case complexity and in experimental performance competitive for reasonable queries and data. **(3)** By this, we extend previously known results for the evaluation of tree and graph queries against tree data to graph data and show where complexity and experimental performance is affected by the change.

11. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Boston, MA, USA, 1995.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wienerm. The Lorel Query Language for Semistructured Data. *J. on Dig. Libraries*, 1(1), 1997.

[3] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2), 1983.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Candidate Rec., W3C, 2005.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD*, 2002.

[6] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a Query Language and Algebra for Semistructured Data based on Structural Recursion. *VLDB J.*, 9(1), 2000.

[7] D. Chamberlin, P. Frankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Working draft, W3C, 2005.

[8] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In *Proc. ACM SIGMOD*, 2005.

[9] Z. Chen, H. V. Jagadish, L. V. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. Int'l. Conf. on Very Large Databases*, 2003.

[10] J. Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999.

[11] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Int. J. on Artificial Intelligence*, 34(1), 1987.

[12] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Proc. Annual IEEE Symposium on Logic in Computer Science*, 2002.

[13] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 2005.

[14] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *J. of the ACM*, 52(2), 2005.

[15] G. Gottlob, C. Koch, and K. Schulz. Conjunctive Queries over Trees. In *J. of the ACM*, 53(2), 2006.

[16] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach A Relational DBMS to Watch its (Axis) Steps. In *Proc. Int'l. Conf. on Very Large D.*, 2003.

[17] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. In *Proc. ACM SIGMOD*, 2005.

[18] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proc. Int'l. Conf. on Very Large Data Bases*, 2005.

[19] M. Marx. Conditional XPath, the First Order Complete XPath Dialect. In *Proc. ACM Symposium on Principles of Database Systems*, 2004.

[20] H. Meuss and K. U. Schulz. Complete Answer Aggregates for Treelike Databases: A Novel Approach to Combine Querying and Navigation. *ACM Trans. on Information Sys.*, 19(2), 2001.

[21] S. Paparizos and H. V. Jagadish. Pattern Tree Algebras: Sets or Sequences? In *Proc. Int'l. Conf. on Very Large Data Bases*, 2005.

[22] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Working draft, W3C, 2006.

[23] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Ph.D. thesis, University of Munich, 2004.

[24] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.