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

François Bry   Tim Furche   Benedikt Linse   Andreas Schroeder

Institute for Informatics, University of Munich
Oettingenstraße 67, 80538 Munich, Germany
(firstname.lastname)@ifi.lmu.de

ABSTRACT

N-ary conjunctive queries, i.e., queries with any number of answer variables, are the formal core of many Web query languages including XSLT, XQuery, SPARQL, and Xcerpt. Despite a considerable body of research on the optimization of such queries over *tree-shaped* XML data, little attention has been paid so far to efficient access to *graph-shaped* XML, RDF, or Topic Maps. We propose the first evaluation technique for *n*-ary conjunctive queries that applies to both tree- and graph-shaped *data* and retains the same complexity as the best known approaches that are restricted to tree-shaped data only. Furthermore, the approach treats tree and graph-shaped *queries* uniformly without sacrificing evaluation complexity on the restricted query class. The core of the evaluation technique is based on dynamic programming using a memoization data structure, called "memoization matrix". It can be populated and consumed in different ways. For each of population and consumption, we propose two resp. three algorithms each having their 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*

General Terms: Algorithms, Performance, Languages

Keywords: query evaluation and optimization, conjunctive queries, memoization, semi-structured data, XML, RDF

1. INTRODUCTION

Semi-structured data in the form of XML or RDF nowadays 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 or asset management, news aggregation, and information classification.

Web query languages such as XSLT, XQuery, SPARQL, or Xcerpt [12] provide convenient means to access such data.

Efficient evaluation of queries over XML data has received considerable attention in recent years [1, 6] including extensive studies of complexity of query evaluation for XPath [6], XQuery [9], and general conjunctive queries over trees [7].

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 XML's 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 over semi-structured data, tree- or graph-shaped, as *n-ary conjunctive queries over unary and binary relations*. Conjunctive queries over tree data form a common formal basis for the query core of a large set of XML query languages such as XPath [6], and thus XQuery; conjunctive queries over graph data for RDF query languages such as SPARQL and general semi-structured query languages such as Lorel and Xcerpt [12].

Compared to full semi-structured query languages, the main restrictions of *n*-ary conjunctive queries are twofold: **(1)** They disallow *result construction*: The result of an *n*-ary conjunctive query is just a set of tuples of bindings for the n result 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 [9], 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, cf. [9].

An extension of our results that drops the first restriction is straightforward and covers *composition-free core XQuery without negation* as defined in [9]. Indeed, the algorithms presented in this paper reaffirm the complexity results from [9] on tree data and extend them to graph data.

For the evaluation of *n*-ary conjunctive queries, we present two algorithms both based on a compact data structure, called *"memoization matrix"*, for memorizing 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

---

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

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

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 path-wise population inspired by [11]).

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, graph queries require exponential time for evaluation. This is in line with complexity results in [6, 7] that show that evaluation of graph queries even over tree data is already NP-complete.

This article is organized along its **contributions**:
1: A memoization technique for the compact representation of intermediary and final results of an n-ary conjunctive query is introduced in Section 3.
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. to complexity.
3: We introduce two algorithms for matrix consumption (Section 5), one for tree queries, and one for graph queries that enforces the remaining non-hierarchical relations that have not been considered during matrix population.
4: Careful complexity analysis of the algorithms in Sections 4 and Section 5 is complemented by an introspective experimental evaluation (Section 7) that confirms the complexity results and shows that the algorithms are competitive. This result applies over all three classes of queries considered, viz. n-ary path, tree, and graph queries. A summary of the complexity results is given in Table 1.

2. PRELIMINARIES

2.1 Graph Data Model

From the perspective of the used 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 over 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, $E \subset N \times \mathbb{N} \times N$ the set of edges, $R \subset N$ the set of root vertices, $\mathcal{L} : N \rightarrow \Sigma_L$ the labeling function, and $\mathcal{V} : N \rightarrow \Sigma_V$ the value assignment function.

D is an *ordered* graph. Since the order is relative to the parent and a single node can be child of several parents, the
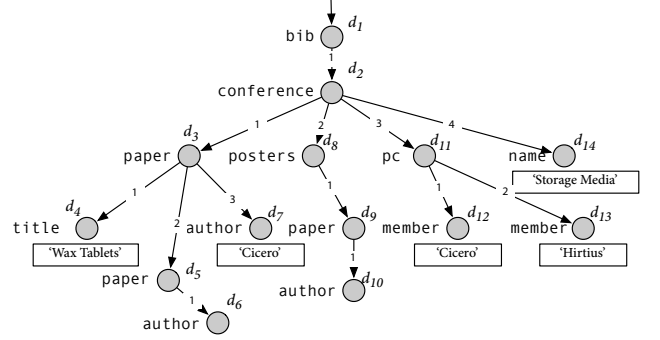

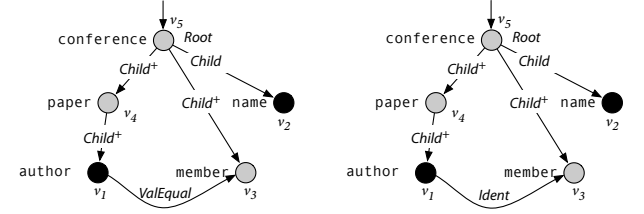**Figure 1: Data—Ordered Simple Directed Graph** (order indicated by numbers on edges)

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

order is associated with the edge rather than with the child node. Since the graphs are also simple, each child has a unique position in the order of its siblings.

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 Fig. 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. 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 $O(|E| + |N|) = O(|E|)$.

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 over. The query head is a list of answer variables, bindings for which form an answer. All answer variables must occur also in the body.

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).

Query Relations. Three types of relations may occur in conjunctive queries: unary "property" relations that restrict bindings to nodes with a certain property, binary "structural" relations that require pairs of nodes in the queried

$$\llbracket q(v_1,\ldots,v_n) \leftarrow atom_1,\ldots,atom_m \rrbracket_D$$
$$= \pi_{v_1,\ldots,v_n}(\llbracket atom_1 \rrbracket_D^q \cap \ldots \cap \llbracket atom_m \rrbracket_D^q)$$
$$\llbracket unary(x) \rrbracket_D^q = \{t \in N^q : t[x] \in \llbracket unary \rrbracket_D\}$$
$$\llbracket binary(x,x') \rrbracket_D^q = \{t \in N^q : (t[x],t[x']) \in \llbracket binary \rrbracket_D\}$$
$$\llbracket \text{ROOT} \rrbracket_D = R$$
$$\llbracket \text{LABEL}_\sigma \rrbracket_D = \{n \in N : \mathcal{L}(n) = \sigma\}$$
$$\llbracket \text{CHILD} \rrbracket_D = \{(n,n') \in N^2 : \exists i \in \mathbb{N} : (n,i,n') \in E\}$$
$$\llbracket \text{CHILD}^+ \rrbracket_D = \bigcup_{i>0}(\llbracket \text{CHILD} \rrbracket_D)^i$$
$$\llbracket \text{CHILD}^* \rrbracket_D = \bigcup_{i\geq 0}(\llbracket \text{CHILD} \rrbracket_D)^i$$
$$\llbracket \text{IDENT} \rrbracket_D = \{(d,d) \in N^2\}$$
$$\llbracket \text{VALEQUAL} \rrbracket_D = \{(d,d') \in N^2 : \mathcal{V}(d) = \mathcal{V}(d')\}$$

Table 2: Semantics of n-ary Conjunctive Queries

data graph to be in a certain structural relation, and binary "join" relations that compare nodes.

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 $O(|E|)$.

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$_\sigma$ 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 [10]) 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 $\mathcal{V}$. Queries with structural relations that form a graph can be rewritten to queries with structural relations in tree shape containing additional IDENT edges.

Fig. 2 shows the query graphs for two conjunctive queries. This representation of graphs is used throughout this paper: Labels and values, as well as root nodes are represented as in data graphs, but edges are annotated with structural or join relations. Answer variables are marked by the variable name and a node filled with black.

Query semantics. Let D be the data graph the query Q is evaluated over 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.

Query classes. We distinguish graph, tree, and path queries. Tree queries are queries whose graphs are tree shaped. Path queries are queries whose graphs are in fact single paths. We introduce the class of *structural tree queries*, queries where the query restricted to unary and structural relations is a tree. There may be additional arbitrary join relations, so that the complete graph is no tree. The query graphs from Fig. 2 are structural tree queries, but no tree queries in the strict sense due to the non-tree join relations.

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 $SoureVars(Q)$ 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 $a \in Q$ and $Q\backslash\{a_1,\ldots,a_n\}$ to test for the occurrence of an atom a in Q, resp. 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 REL(x,y) by the tree edge REL(x,y'), and add 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 and quadratic in the number of variables in the query.

Irrespective of the nature of a rooted query graph Q, we denote a spanning tree of Q with $T(Q)$.

3. MEMOIZATION MATRIX

At the core of the proposed evaluation technique stands the "memoization matrix". It is a compact data structure holding intermediary results of the evaluation of an n-ary conjunctive query, inspired by a more limited and even on tree data exponential data structure from [13]. It assigns query nodes q with nodes $n \in N$ and *one* sub-matrix, containing for each child node q' of q the compatible bindings $n' \in N$ 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 over D is a recursive data structure representing all possible bindings of query variables in Q to nodes from D. Let $SourceVars(Q)$ be the variables in Q that are only occurring as sources in structural relations in Q. Then the memoization matrix for Q over D is a relation containing for each $q_s \in SourceVars(Q)$ 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 $Q\backslash SourceVars(Q)$ such that for each tuple $(q',n',M'') \in M'$ and each atom REL$(q_s,q') \in T(Q)$, it holds that $(n,n') \in \llbracket \text{REL} \rrbracket_D$.

Intuitively, the bindings for source variables in a sub-matrix M' must be structurally compatible with the binding of the source variable in the corresponding tuple of M.

Notice that only relations in the spanning tree of Q, $T(Q)$, are considered. This can lead to assigning structural relations to non-tree edges, which is unfavorable if no index exists that guarantees practical verification time. In such a case, a transformation from graph query to structural tree query is performed as discussed above.

Though the memoization matrix ensures that related bindings for different variables are consistent w.r.t. structural and property relations it does not ensure that related bindings are consistent w.r.t. join relations. This is to exploit the tree property of structural relations that does obviously not hold for join relations in structural tree queries: where join relations can relate arbitrary variables in the query, structural relations over variables form a tree, thus making a local

Variable	Node	Sub-Matrix				
v_5	d_2	**Variable**	**Node**	**Sub-Matrix**		
		v_4	d_3	**Variable**	**Node**	**Sub-Matrix**
				v_1	d_6	
				v_1	d_7	
		v_4	d_5	**Variable**	**Node**	**Sub-Matrix**
		v_4	d_9	**Variable**	**Node**	**Sub-Matrix**
				v_1	d_{10}	
		v_3	d_{12}			
		v_3	d_{13}			
		v_2	d_{14}			

Figure 3: Memoization Matrix (on Data of Fig. 1)

evaluation of structural relations possible: A full-match can be computed from local matches that consider parent and child variables in the structural tree query in isolation.

To avoid multiple computations of matches, 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.

It is assumed that the matrix is clustered by variables allowing linear access to all entries relating to a variable.

Fig. 3 shows the memoization matrix for the evaluation of query Q_1 from Fig. 2 over the sample data (Fig. 1).

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

LEMMA 1. *The **size** of the memoization matrix M for a query Q and a data graph D with nodes N is bounded by $(2q - 1) \cdot v^2$.*

PROOF. By structural induction over $T(Q)$.
Query leaves: It holds that $q = 1$, and the number of bindings 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 $q - 1 = \sum_{j=1}^{c} q_j$ (*). There are again at most v bindings of i. As tuples are shared, there is at most *one* tuple for each such binding. The size of the sub-matrix contained in the tuple itself is bounded by $c \cdot v$, as each child has at most v bindings. 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, using the induction hypothesis, $\sum_{j=1}^{c} (2q_i - 1) \cdot v^2 + c \cdot v^2 \overset{(*)}{=} (2(q-1) - c + c) \cdot v^2 \leq (2q - 1) \cdot v^2$. □

Based on the populated matrix, the algorithms discussed in Section 5 traverse the 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 (Match$_\uparrow$, Section 4.1) or top-down (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 (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 Match$_\uparrow(Q, D)$

1: $V_Q \leftarrow vars(Q);$ $N \leftarrow nodes(D)$; $\rho \leftarrow \emptyset$
2: $root \in SourceVars(Q)$
3: **while** $\rho(root) = \emptyset$ **do**
4:    take $x \in V_Q : \rho(x) = \emptyset \wedge$
         $\forall \text{REL}(x, x') \in T(Q) : \rho(x') \neq \emptyset$
5:    $M \leftarrow \emptyset$
6:    **for all** $n \in N$ **do**
7:       **if** $\exists \text{REL}(x) \in Q : n \notin [\![\text{REL}]\!]_D$ **then**
8:          **continue** n
9:       $M_S \leftarrow \emptyset$
10:       **for all** $\text{REL}(x, x') \in T(Q)$ **do**
11:          $M_R \leftarrow \emptyset$
12:          **for all** $(x', n', M') \in \rho(x')$ **do**
13:             **if** $(n, n') \in [\![\text{REL}]\!]_D$ **then**
14:                $M_R \leftarrow M_R \cup \{(x', n', M')\}$
15:          **if** $M_R = \emptyset$ **then**
16:             **continue** n
17:          $M_S \leftarrow M_S \cup M_R$
18:       $M \leftarrow M \cup \{(x, n, M_S)\}$
19:    $\rho(x) \leftarrow M$
20:    **if** $\rho(x) = \emptyset$ **then**
21:       **return** $\emptyset$
22: **return** $\rho(root)$

The algorithm uses a helper data structure 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).

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 Match$_\uparrow$). *Let $v = |N|$, $q = |vars(Q)|$, and $e = |E|$. Then, Match$_\uparrow$ has $O(q \cdot v \cdot e)$ combined time and $O(q \cdot v^2)$ combined space complexity.*

PROOF. There are q variables, so that the outer loop (l. 3) is bound by q. The loop over all nodes (l. 6) is bound 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 $q - 1$ structural relations that need to be tested in the iteration starting at l. 10. As each binary relation is visited only once (when the source variable of that relation is processed), the loop (l. 11–17) is executed $(q-1) \cdot v$ times. Since there are at most v bindings for each variable, the iteration in l. 12 is bound by v. As verifying $(n, n') \in [\![\text{REL}]\!]_D$ is in $O(e)$ for structural relations (cf. Section 2), the overall time complexity is in $O(q \cdot v^2 \cdot e)$. However, we can assume that the structural relations are precomputed in an index structure which provides constant verification time for the structural relation at a space cost of $O(v^2)$ and time cost $O(v \cdot e)$ per structural relation. This is an acceptable trade-off as there are usually only a small number of structural relations and as the memoization matrix already requires space in $O(q \cdot v^2)$. 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 precomputed relations are in $O(v^2)$ and the size of ρ is in $O(q)$. $\square$

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, bottom-up algorithms need efficient index structures on structural relations occurring in the query. A further performance increase might be obtained by evaluating groups of structural and property relations at once using holistic tree queries, cf. [1]. 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 to be evaluated at once. Further investigation of the use of such holistic schemes for n-ary conjunctive graph queries is required, but out of the scope of this paper.

4.2 Top-Down Approach

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 structural relations must be guaranteed in $O(e)$ time (cf. l. 11). 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 were checked and did not 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 Match$_\downarrow(x, n)$

1: **if** $\rho(x, n) = \perp$ **then**
2:    **return** $\emptyset$
3: **if** $\rho(x, n)$ defined **then**
4:    **return** $\{(x, n, \rho(x, n))\}$
5: **if** $\exists \text{REL}(x) \in Q : n \notin [\![\text{REL}]\!]_D$ **then**
6:    $\rho(x, n) \leftarrow \perp$
7:    **return** $\emptyset$
8: $M_S \leftarrow \emptyset$
9: **for all** $\text{REL}(x, x') \in T(Q)$ **do**
10:    $M_R \leftarrow \emptyset$
11:    **for all** $n' \in N : (n, n') \in [\![\text{REL}]\!]_D$ **do**
12:       $M_R \leftarrow M_R \cup \text{Match}(x', n')$
13:    **if** $M_R = \emptyset$ **then**
14:       $\rho(x, n) \leftarrow \perp$
15:       **return** $\emptyset$
16:    $M_S \leftarrow M_S \cup M_R$
17: $\rho(x, n) \leftarrow M_S$
18: **return** $\{(x, n, M_S)\}$

The top-down algorithm performs a recursive descent over the query structure. It has two parameters, a query x and a data node n, and computes the memoization matrix for these two nodes. For each pair, a matching is computed at most once. 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 over D.

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

PROOF. The use of matrix memoization (l.1–3, 14, 17) guarantees that Match$_\downarrow$ is executed at most once for each combination of variable and data node (x, d). Testing unary predicates takes constant time. As each of the $q-1$ relations is visited at most once, the loop over all binary relations (l. 9) is performed at most $(q - 1) \cdot v$ times. Enumerating all values of any structural relation is in $O(e)$, and thus 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) takes constant time (memoization in ρ amortizes the recursive call), Match$_\downarrow$ is in $O(q \cdot v \cdot e)$ combined time complexity. $\square$

As section 7 shows, the algorithm Match$_\downarrow$ is a competitive algorithm with linear time complexity in many real world scenarios, even without any index structures. Streaming schemes [2] and similar techniques could be used to refine the algorithm further and speedup the average runtime, but are beyond the scope of this paper.

5. MATRIX CONSUMPTION

The consumption of a memoization matrix for the evaluation of a query Q over 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, 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 the fact that

the nested matrices consists of bindings for several relations and must be hence decomposed into partitions before the flattening takes place, and that a sub-matrix tuple can be referenced by several matrices.

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. Since there are no such remaining relations in tree queries, the matrix consumption algorithm is a simple flattening of the nested memoization matrix to produce the output. Since the output size is larger than every intermediate result when evaluating tree queries, the time and space complexity of the consuming algorithm is bound 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 remaining binary relations not in query spanning tree. Thus, the matrix consumption for graph queries has exponential combined time complexity. To illustrate this, consider the queries from Fig. 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.

As stated above, the consumption algorithm Output_T for tree queries is a simple flattening of the (nested) memoization matrix to a set of tuples of variable bindings. During the flattening dynamic programming is used to avoid duplicate construction, thus space and time complexity are bound by the result size. For space reasons, the full algorithm is omitted and only complexity results are given.

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.*

In the following section, we take a closer look at the matrix consumption algorithm for graph queries outlining briefly the benefits and drawbacks of a nested loop join for secondary storage processing (Section 5.2).

5.1 Matrix Consumption for Graph 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 in 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 can be distributed over Cartesian products the consumption algorithm for tree queries performs and combined into joins—with possibly non-atomic conditions, if more than one relation must be verified in one Cartesian product.

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 *join and projection specification* for each variable, and of a function that applies these join specification to a set of valuations. The join and projection specification is created by a query planner; Fig. 4 shows an example of a join and projection specification.

The new algorithm however exhibits exponential worst case runtime in that it may perform at worst $q - 3$ Cartesian
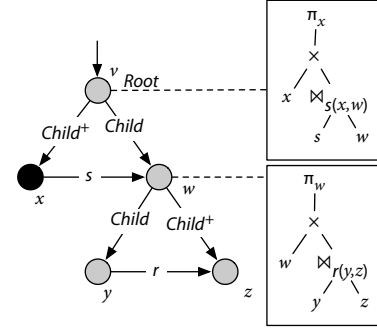

Figure 4: Join and Projection Specification

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 each join divides the result size by at least $|N|$, the overall runtime can be approximated 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.*

5.2 Incremental Matrix Consumption

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, and on graph queries* $O(j(|N|) \cdot |N|^q)$ *time and* $O(q \cdot n^2)$ *space complexity.*

The advantage of Output_{NLP} is its low space complexity which is essentially bound by the size of the memoization matrix. However, this advantage is paid for by an exponential time complexity in almost all cases. Furthermore, it is reached in many practical cases

6. ORDER RELATIONS ON GRAPH DATA

In the previous sections, graph shaped data is considered equivalent in querying aspects to tree shaped data. Although the worst case complexity of the matrix population

algorithms is the same in both cases, the order of two sibling nodes is context dependant in graphs, cf. Section 2.1.

For the support of ordered queries in graph data, a ternary relation NEXT and its transitive, reflective closures NEXT$^+$, NEXT* are introduced. The semantic of NEXT is defined as $[\![\text{NEXT}]\!]_D = \{(c, c') : \exists (p, i, c), (p, i', c') \in E.\, i + 1 = i'\}$ with the closure relations defined as usually. 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.

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 of the data model is accessible. Assuming that $Next^*(x, y, y')$ must hold and that I is the set of positions of all y bindings and I' of y' respectively, it must obviously hold that $\forall i \in I.i \leq max(I')$ and $\forall i' \in I'.min(I) \leq i'$. When populating the bindings of y', the minimum position of bindings of y is known, so that the above condition can be imposed on all bindings of y'.

7. EXPERIMENTAL EVALUATION

The experimental evaluation is based on both synthetic and on real data. The set of structural relations is extended by the additional relations ATTRIBUTE and VALUE 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. All tests show the processing time without data parsing. Each measurement is averaged over 500 runs.

Synthetic data is used to 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.

The first experiment confirms on synthetic data how essential the memoization of intermediary results is, not only for the complexity but also for the experimental query evaluation time. The Match$_\downarrow$ algorithm without memoization of variable domains (i.e., the helper structure ρ) exhibits an exponential growth of time consumption in the size of the query (cf. Fig. 5), because several common sub-matrices are built repeatedly. In contrast, Fig. 6 depicts the effect of increasing arity in a worst-case scenario, where the query is unrestrictive, i.e., a binding for one answer variable is related to all bindings of another one.

Fig. 7 shows 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 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 and the bottom-up algorithm with CHILD* index. Without this index, the bottom-up approach exhibits a cubic runtime.

At least the top-down algorithm performs quite well even in its basic form discussed here in real query scenarios. Fig. 8 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 over the MONDIAL[2] database of geographical information. 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 Match$_\downarrow$ exhibits a linear runtime in all queries, even in the graph query experiment.

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

8. EXTENSIONS AND OUTLOOK

Though the experimental evaluation shows that even the basic form of the proposed algorithm performs quite 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*, as needed in Xcerpt [12]. This algorithm must handle negated and optional query parts as in general tree patterns [3].

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.

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 binding is replaced by a direct link. By this, 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. Furthermore, it has been shown (e.g., [8] and [5]), that queries over graph data remain tractable for the class of queries with bounded (hyper-) tree-width rather than just over tree-shaped queries. How to extend the presented algorithms to that larger class, remains an open issue. One step in this direction is the support of more expressive structural relations, e.g., conditional axis [10] that allow collapsing entire paths both for population and consumption of the matrix.

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 [7]. A restriction of the bottom-up algorithm discussed in this article to conjunctive *tree queries* is similar to the evaluation algorithm of [11] and has the same complexity.

Matching conjunctive queries over 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 $O(q \cdot v^2)$ [4], though this result assumes $O(1)$ verification time for all relations. However, the implication from arc to global constraint consistency used in this result, does not hold for graph-shaped constraint problems.

In [7] 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 X-property (read: X-underbar) over an order $<$, arc-consistency is sufficient to guarantee that the minimal solution (in terms of the

---

[1] http://www.cs.washington.edu/research/xmldatasets/
[2] http://www.dbis.informatik.uni-goettingen.de/Mondial/
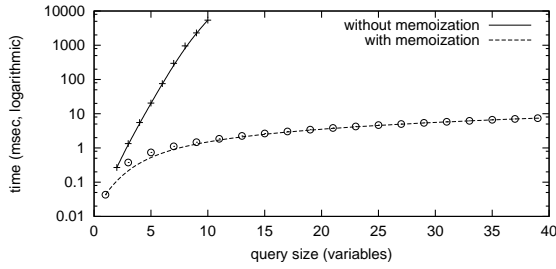
Figure 5: Effect of Memoization over Query Size
(data synthetic, uniform, deeply nested; bindings for query variables overlap considerably)
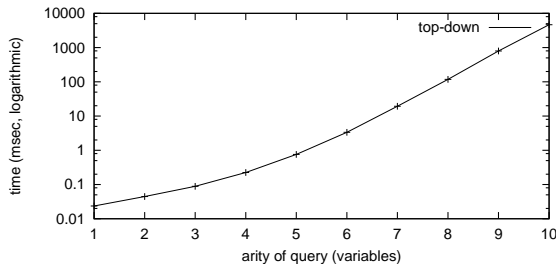

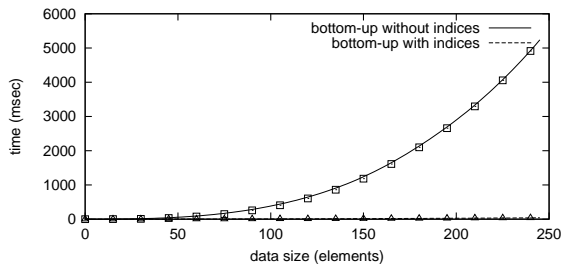**Figure 6: Worst-Case Effect of Query Arity** (data and query as before)

Figure 7: Comparison Top-Down and Bottom-Up
(data synthetic, size increased by adding in depth; query small, containing many descendants)
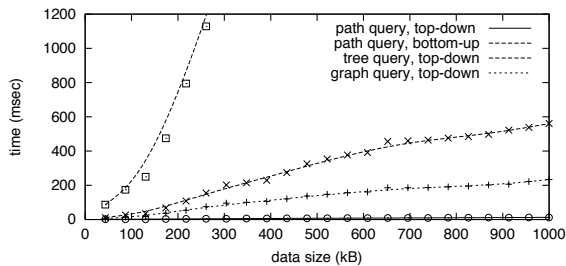

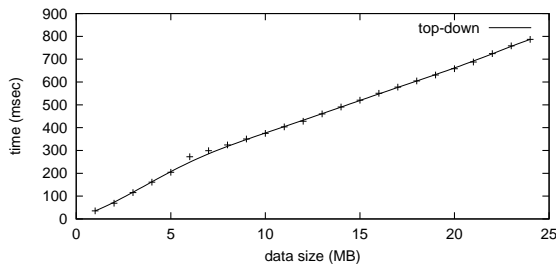**Figure 8: Query Classes over Real-life Data** (Mondial; queries simple, but with arity > 1)

Figure 9: Top-Down over Large, Real-life Nasa Data
(binary tree query)

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 answer 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 in the number of variables is the best achievable.

Another field of important related work are structural indexing techniques [1, 2]. 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 [1] is a promising and recently widely researched technique for tree data. On such data, their application to the discussed algorithms is straightforward.

10. CONCLUSION

The memoization matrix is a compact recursive 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. Based on this data structure, we show **(1)** that the shape of the data has little or no effect on the query complexity for the chosen relations; **(2)** that a unified algorithm for both tree- and graph-shaped semi-structured queries is feasible and competitive, both in worst-case complexity and in experimental performance.

11. REFERENCES

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

[2] 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.

[3] 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.

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

[5] J. Flum, M. Frick, and M. Grohe. Query Evaluation via Tree-Decompositions. *J. of the ACM*, 2002.

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

[7] G. Gottlob, C. Koch, and K. Schulz. Conjunctive Queries over Trees. In *Proc. ACM PODS*, 2004.

[8] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions and Tractable Queries. In *Proc. ACM PODS*, 1999.

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

[10] M. Marx. Conditional XPath, the First Order Complete XPath Dialect. In *Proc. ACM PODS*, 2004.

[11] 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.

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

[13] S. Schaffert. Xcerpt: A Rule-Based Query and Transformation Language for the Web. PhD Thesis, Institute for Informatics, University of Munich, 2004.