I4-D10

An Abstract Machine for Xcerpt's Single-Rule Programs: Theory and Practice

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D10/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Massimo Marchiori and Jan Maluszinsky
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2006
Actual submission date:	31 December 2006

Abstract

Efficient evaluation of query languages has long been a core topic of database research. For Web query languages such as Xcerpt, however, many of the questions studied previously for traditional, centralized, relational databases and query languages have to be asked again and known answers affirmed, adapted, or entirely new answers developed. This deliverable presents in two parts some answers for efficient evaluation of Xcerpt, a novel rule-based, reasoning-aware query language for semi-structured data such as XML and RDF. These answers are focused on (a) an efficient data structures for Xcerpt's simulation unification, (b) a preliminary algebra utilising this data structure, and (c) algebraic construction of semi-structured query results.

Keyword List

evaluation,abstract machine,algebra,Xcerpt,XML,RDF

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

An Abstract Machine for Xcerpt's Single-Rule Programs: Theory and Practice

Tim Furche[1], **Michael Brade**[1], **François Bry**[1], **Benedikt Linse**[1], **Andreas Schroeder**[1]

[1] Institute for Informatics, University of Munich, Germany
Email: ⟨*first-name*⟩.⟨*last-name*⟩@ifi.lmu.de

31 December 2006

Abstract
Efficient evaluation of query languages has long been a core topic of database research. For Web query languages such as Xcerpt, however, many of the questions studied previously for traditional, centralized, relational databases and query languages have to be asked again and known answers affirmed, adapted, or entirely new answers developed. This deliverable presents in two parts some answers for efficient evaluation of Xcerpt, a novel rule-based, reasoning-aware query language for semi-structured data such as XML and RDF. These answers are focused on (a) an efficient data structures for Xcerpt's simulation unification, (b) a preliminary algebra utilising this data structure, and (c) algebraic construction of semi-structured query results.

Keyword List
evaluation,abstract machine,algebra,Xcerpt,XML,RDF

Contents

Chapter 1

Outline of this Deliverable

Efficient evaluation of query languages has long been a core topic of database research. For Web query languages such as Xcerpt, however, many of the questions studied previously for traditional, centralized, relational databases and query languages have to be asked again and known answers affirmed, adapted, or entirely new answers developed. This deliverable presents in two parts some answers for efficient evaluation of Xcerpt, a novel rule-based, reasoning-aware query language for semi-structured data such as XML and RDF. These answers are focused on (a) an efficient data structures for Xcerpt's simulation unification, (b) a preliminary algebra utilising this data structure, and (c) algebraic construction of semi-structured query results.

The deliverable is divided in two parts: The first part discusses a proposal for a data structure, near-optimal in memory and time, for storing intermediary results during evaluation of conjunctive queries against semi-structured data. This part of the deliverable is based on a recent paper at the ACM Int'l. Workshop on "Web Information and Data Management" (WIDM 2006). The second part of the deliverable focuses on an algebra and abstract machine for Xcerpt's simulation unification. The discussed operators present a preliminary set of operators that utilize the memoization matrix introduced in the first part for efficient evaluation.

Members of I4 are currently working on a further refinement of both the data structure (to optain optimal memory and time over all cases of tree and graph data) and the algebra. In particular the latter has been substantially revised. More detailed results will be published in further deliverables in year 4 of REWERSE.

Part I

Theory

1.1 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 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**:

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

1: A memoization technique for the compact representation of intermediary and final results of an n-ary conjunctive query is introduced in Section 1.3.
2: We introduce two algorithms for populating this matrix, one bottom-up in Section 1.4.1, one top-down in Section 1.4.2, and compare these algorithms w.r.t. to complexity.
3: We introduce two algorithms for matrix consumption (Section 1.5), 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 1.4 and Section 1.5 is complemented by an introspective experimental evaluation (Section 1.7) 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 1.1.

1.2 Preliminaries

1.2.1 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

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

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, $\mathscr{L} : N \to \Sigma_L$ the labeling function, and $\mathscr{V} : N \to \Sigma_V$ 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 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. $\mathscr{L}$ and $\mathscr{V}$. 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, $\mathscr{L}$ is assumed to be total, whereas $\mathscr{V}$ may be undefined for some nodes in the graph. In Fig. 1.1, 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 $O(|E| + |N|) = O(|E|)$.

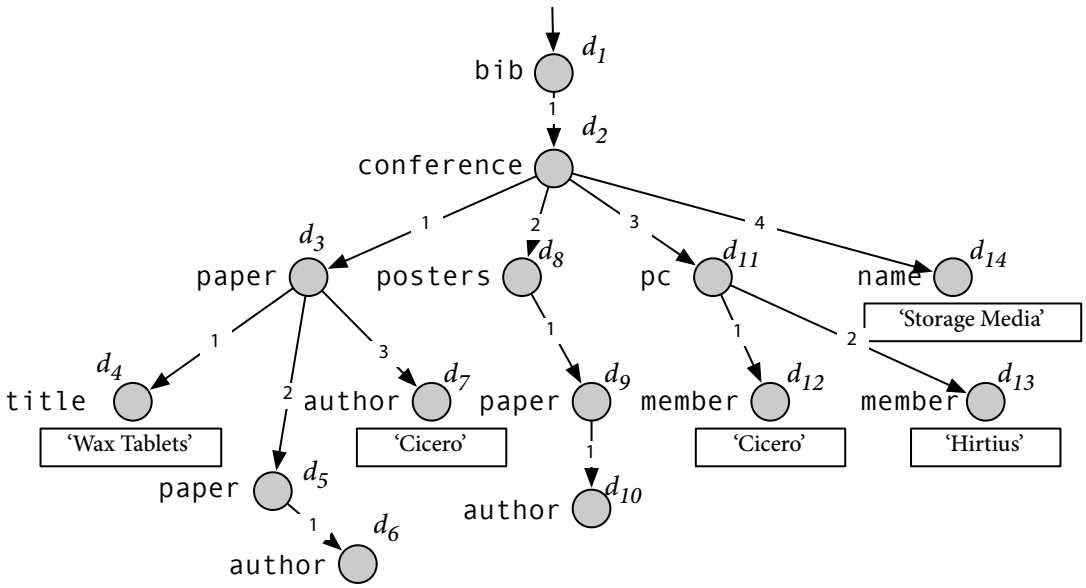

Figure 1.1: Data—Ordered Simple Directed Graph (order indicated by numbers on edges)

1.2.2 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 1.6 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 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_σ for all $\sigma \in \Sigma_L$ (i.e., for all possible labels) that restrict to nodes v where $\mathscr{L}(v) = \sigma$.

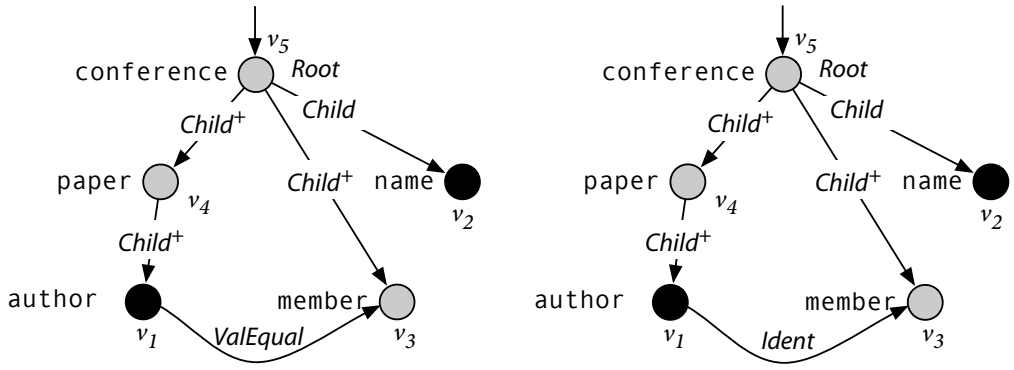

Figure 1.2: Exemplary Query Graphs (left: Q_1 with value join, right: Q_2 with identity join)

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

Table 1.2: Semantics of n-ary Conjunctive Queries

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 1.6, 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 $\mathscr{V}$. Queries with structural relations that form a graph can be rewritten to queries with structural relations in tree shape containing additional IDENT edges.

Fig. 1.2 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 1.2 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. 1.2 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 $\text{REL}(x, y)$ by the tree edge $\text{REL}(x, y')$, and add $\text{IDENT}(y, y')$ 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 $T(Q)$.

1.3 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 $\text{REL}(q_s, q') \in T(Q)$, it holds that $(n, n') \in [\![\text{REL}]\!]_D$.*

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 evaluation of structural relations

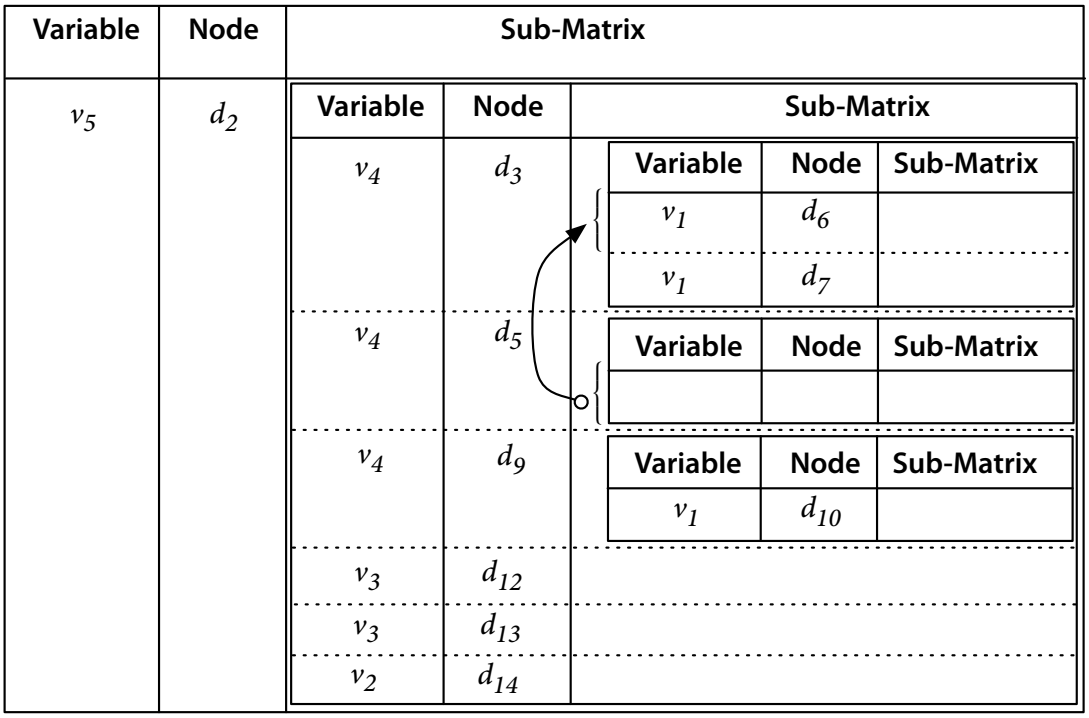

Figure 1.3: Memoization Matrix (on Data of Fig. 1.1)

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. 1.3 shows the memoization matrix for the evaluation of query Q_1 from Fig. 1.2 over the sample data (Fig. 1.1).

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 1 *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 1.5 traverse the matrix, enforce the remaining (non-hierarchical) relations, if there are any, and create the output according to the query semantics introduced above.

1.4 Matrix Population

The compact memoization matrix introduced in the last section can be produced bottom-up ($\mathsf{Match}_{\uparrow}$, Section 1.4.1) or top-down ($\mathsf{Match}_{\downarrow}$, Section 1.4.2), 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 1.7 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.

1.4.1 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.

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

Proof 2 *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* $(n, n') \in [\![\,\mathrm{REL}\,]\!]_D$ *is in* $O(e)$ *for structural*

Algorithm 1.1 $\mathsf{Match}_{\uparrow}(Q, D)$

```
1:  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)
```

relations (cf. Section 1.2), 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 $O(q)$.

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 1.7 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.

1.4.2 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).

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

Proof 3 *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 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), $\mathsf{Match}_{\downarrow}$ is in $O(q \cdot v \cdot e)$ combined time complexity.*

Algorithm 1.2 $\mathsf{Match}_{\downarrow}(x, n)$

```
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)}
```

As section 1.7 shows, the algorithm $\mathsf{Match}_{\downarrow}$ 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.

1.5 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. 1.2: The memoization matrix only enforces the structural relations, but does not consider

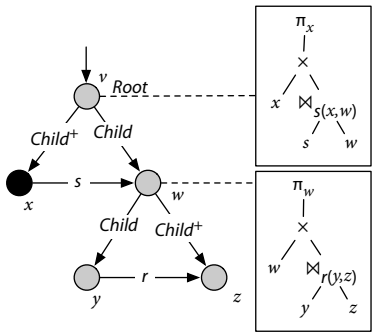

Figure 1.4: Join and Projection Specification

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 1.5.2).

1.5.1 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. 1.4 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 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.*

1.5.2 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

1.6 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 1.2.1.

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'.

1.7 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 $\texttt{Match}_{\downarrow}$ 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. 1.5), because several common sub-matrices are built repeatedly. In contrast, Fig. 1.6 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. 1.7 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 $\texttt{Match}_{\downarrow}$, 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. 1.8 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 $\texttt{Match}_{\downarrow}$ 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 $\texttt{Match}_{\downarrow}$ scales nicely with the data size and is very competitive even in the basic form implemented for this experiment.

1.8 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

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

[2] http://www.dbis.informatik.uni-goettingen.de/Mondial/

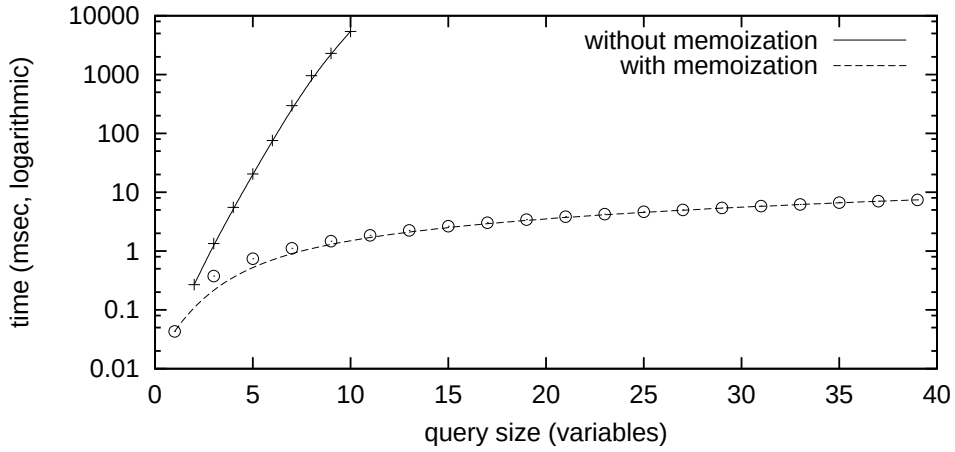

Figure 1.5: Effect of Memoization over Query Size (data synthetic, uniform, deeply nested; bindings for query variables overlap considerably)

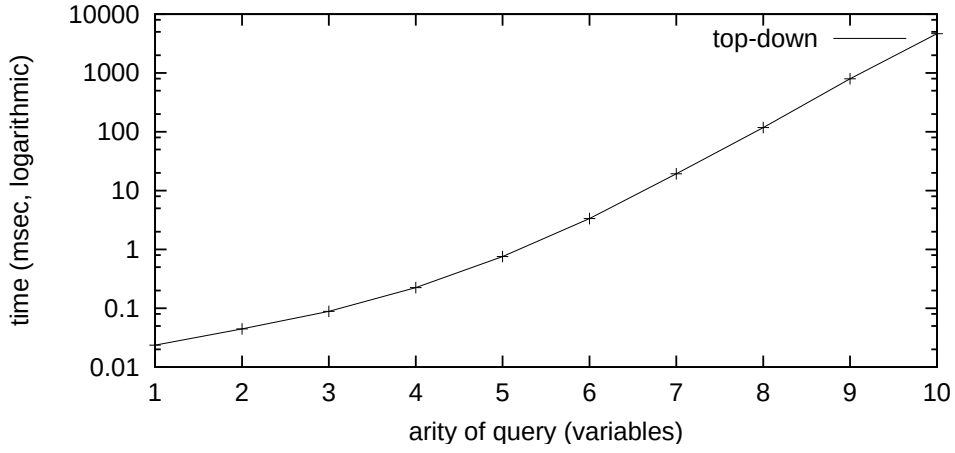

Figure 1.6: Worst-Case Effect of Query Arity (data and query as before)

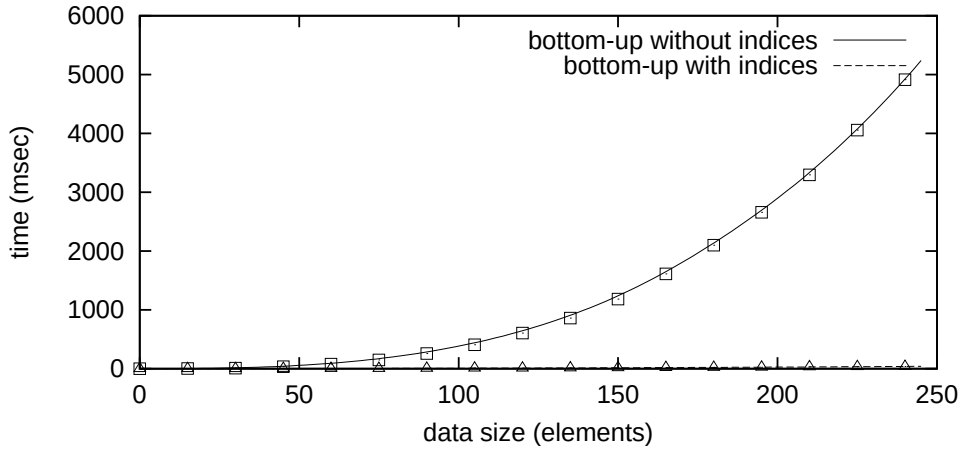

Figure 1.7: Comparison Top-Down and Bottom-Up (data synthetic, size increased by adding in depth; query small, containing many descendants)

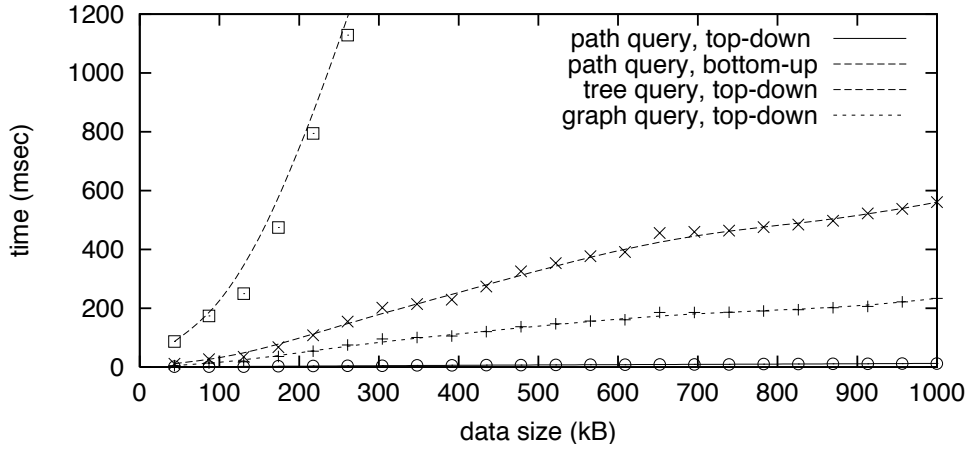

Figure 1.8: Query Classes over Real-life Data (Mondial; queries simple, but with arity > 1)

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.

1.9 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 $\underline{\text{X}}$-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 $\underline{\text{X}}$-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.

1.10 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.

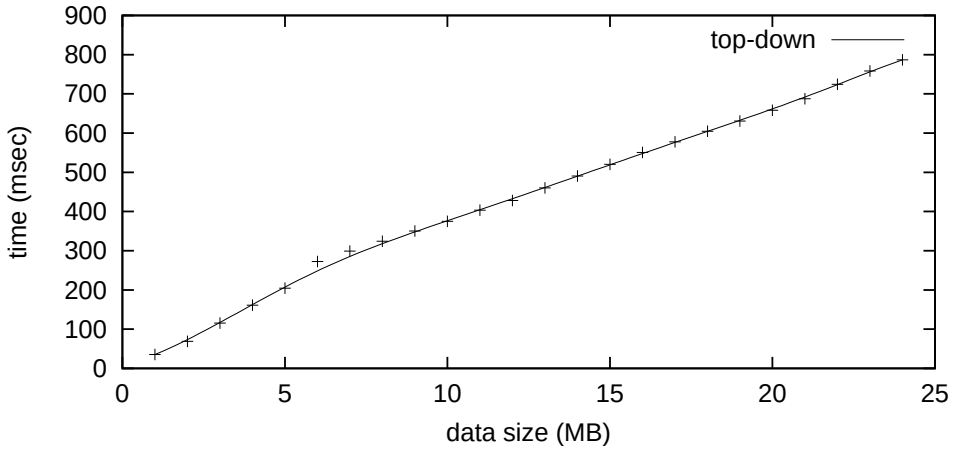

Figure 1.9: Top-Down over Large, Real-life Nasa Data (binary tree query)

Bibliography

[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.

Part II

Practice

Chapter 2

Design and Implementation of an Abstract Machine for Single-Rule Xcerpt

2.1 Introduction

Xcerpt [13] is getting more and more attention these days. It is a powerful and impressive new web query language with an easy to understand and declarative syntax. Xcerpt has now been in development for more than 5 years. But to really gain momentum, a programming language needs to get out of the research status and be used by real-world applications. For that to be possible, efficient and easy to use and understand implementations need to exist. Without that, nobody will use a language, no matter how beautiful it is.

Up to this point in time there is no abstract machine available for the Xcerpt language. The only directly applicable research in this area is the predecessor of this chapter—a chapter which develops the first steps towards an abstract machine that processes Xcerpt's simulation unification. This abstract machine is called "SUAM", short for Simulation Unification Abstract Machine [3].

To work towards a complete abstract machine for Xcerpt, this diploma chapter presents an abstract machine, which is designed to evaluate an Xcerpt single-rule program and is part of the AMAXOS project (Abstract Machine for Xcerpt on Semi-structured Data). One of its goals is to be as simple and easy to understand as possible.

For that reason, and although the AMAXOS is supposed to extend the SUAM, it is not designed on quite the same low level on which the SUAM is designed. The SUAM defines instructions that worked on the level of registers, bytes, the stack and the heap. In contrast to that, the AMAXOS does not provide a strict definition of its own registers, heap, or stack management. It is implemented in Java and C++, both of which are languages with good support to hide these low-level details.

The focus of this chapter is on the bigger picture of the supported machine operations. More refined and low-level implementations, if needed, can be based on this research.

Another difference of the abstract machine in this chapter compared to the SUAM in the earlier paper, is a further developed matrix model. During the research of the SUAM it became clear that its matrix handling and storage is consuming more memory than strictly necessary. A first paper regarding the new matrix has been published already [5], another one based partially on the results of

this chapter will follow shortly.

2.1.1 Overview

The remainder of this chapter is divided into three distinct parts. The first part sets forth the theoretical foundation of Xcerpt, the AMAXOS, and the evaluation models of the AMAXOS. The second part deals with the set-based evaluation model as implemented for this chapter a little more in-depth and the third part is concerned with the implementation details of the Java and C++ implementations of the AMAXOS.

Of the first part, Section 2.2 is an introduction to Xcerpt, Section 2.2.1 shows Xcerpt's features relevant to this work, and Section 2.3 briefly describes the design principles of the AMAXOS, like the memory model, the data model, the matrix model, and the evaluation models.

Of the second part, Section 2.4 is a detailed description of the operators that the AMAXOS understands and which make up the AMAXOS machine code, and Section 2.5 walks though the process of translating an Xcerpt query rule into AMAXOS machine code.

Finally, of the last part, Section 2.6.1 shows the design of the implementations, including the class hierarchy, some comments about certain design decisions, and some problems and their solutions. Section 2.7 gives a few comments regarding the Java implementation, Section 2.8 mainly concerns itself with the C++ implementation and the memory management, and Section 2.9 describes the performance of both implementations.

Since there is still a lot to be researched and improved, Section 2.10 gives a short outline of what can be done in the future regarding the AMAXOS and last, Section 2.11 summarizes a few of the most important points of this chapter.

2.2 An Introduction to Xcerpt

Xcerpt [13] is a declarative rule-based query language for both data on the standard Web and data on the Semantic Web. The data format is not important, Xcerpt can query and generate data in many different formats and is capable of further processing its own output, a feature known as chaining. Xcerpt is even "strongly answer-closed", as any answer to a query can itself be a query.

Xcerpt's queries are *pattern-based* instead of being path-based like in XQuery [2] and XSLT [9]. This provides for *incomplete query declarations* and makes it possible to specify only parts of the data to be found. This may be done in three non-exclusive ways:

- by specifying only a subset of the children of an element,
- by querying elements at arbitrary depths, then also called descendants,
- by declaring some parts of the query to be optional.

Xcerpt queries also provide the possibility to specify *unordered queries*. Those are queries where the order of the given elements need not match the order of the elements in the source document. To be able to process incomplete as well as unordered queries, Xcerpt utilizes a newly developed unification algorithm, called "Simulation Unification".

Xcerpt can process and generate documents that form *graphs*, that is, an element may have more than one parent node. To handle graphs correctly is also the task of the Simulation Unification.

Xcerpt is based on logic programming. An Xcerpt program consists of one or more *goals* and zero or more *rules*. This is to say that Xcerpt is a *rule-based* language. Goals are special rules in that a goal contains the answer construction of a multi-rule program.

A single rule can be regarded as a part of the whole query that the complete Xcerpt program represents. A rule in itself is a query already. As mentioned above, rules can be chained to make one rule process the output of another rule. However, this chapter focuses on single rule Xcerpt programs only and does not implement chaining yet.

Each Xcerpt rule contains a query part and a construction part, it is thus sometimes also called construct-query rule. Rules separate query and construction to enhance readability, consistency, and ease of use. The query part specifies the pattern of the data to be retrieved and the construction part builds the output document by reassembling the pieces from the query part using the document structure also specified in the construction part.

Last but not least, the Xcerpt construction part allows to group (some of) the results of a query part using the keywords **`all`** and **`some`**. Grouping constructs may be nested.

2.2.1 A Bird's Eye View on Xcerpt

The most central part of the evaluation of an Xcerpt rule is the Simulation Unification [14]. To make this new kind of unification efficient, intermediate solutions have to be saved—cached—for later reuse, and to make it space efficient those solutions have to be compressed as much as possible. To this end, the memoization matrix was invented. It stores an exponential number of solutions in a polynomial number of memory cells by storing equal parts of several solutions only once.

This chapter deals mainly with the somewhat low-level parts of Xcerpt, like the matrix. Nevertheless, the general Xcerpt syntax is important for the parser and the understanding of the operators. So the next section describes the supported Xcerpt syntax before the following two sections introduce the original and the revised matrix theory.

2.2.2 Syntax

The current AMAXOS only deals with single construct-query rules and defers the implementation of the chaining to a later point in time. A single rule in Xcerpt looks as follows:

```
GOAL
   <query term>
FROM
   <construct term>
END.
```

Query terms are enhanced data terms. They may contain additional variable declarations such as *`var X`*, variable restrictions such as *`var X as f{}`*, and descendant specifications such as **`desc f{}`**.

Construct terms are data terms that may contain the variables specified in the query term and the **`all`** keyword, which retrieves every single available binding for the specified variables.

2.2.2.1 Examples

The following examples have been taken from the description of the SUAM [3]. They are slightly modified to suit this chapter.

The first example is a real-world example that can be easily understood and that shows a fair few of Xcerpt's features: It is an XML document containing an address book. The corresponding *data term* will be:

```
<addressbook>
  <entry>
    <name>Robert Bart</name>
    <phone>
      <home>03529/3170</home>
      <mobile>0162/4576214</mobile>
      <work>03528/723712</work>
    </phone>
    <address>
      23 George Str, Village 3120
    </address>
  </entry>
  <entry>
    <name>Julia Flower</name>
    <mobile>0034-1252-6829</mobile>
    <email>flower@work.com</email>
  </entry>
  <entry>
    <name>Neil Fisher</name>
    <phone>
      <mobile>0174/3421390</mobile>
      <home>01282/709</home>
    </phone>
    <address>
      <street>45 Shorthorn Street</street>
      <postalcode>40307</postalcode>
    </address>
  </entry>
</addressbook>
```

In Xcerpt syntax the first part of the example would look as follows.

```
addressbook [
 entry [
  name [
   Robert Bart
  ]
  phone [
   home [
    03529/3170
   ]
   mobile [
    0162/4576214
   ]
   work [
    03528/723712
   ]
  ]
  address [
   "23 George Str, Village 3120"
  ]
 ]
 ...
]
```

Suppose the intention is to extract the mobile phone numbers. Then the query term could look like this one:

```
addressbook {{
 entry {{
  desc mobile [ var Mobile ]
 }}
}}
```

The query is partial and unordered (`{{ }}`) because only one element out of an unspecified number are known and required and the order of properties in an address book entry usually does not matter. The mobile phone number is a **`desc`**endant of the `<entry>` element, that is, it needs not to be a direct child of `<entry>`.

To create a new document with all available mobile phone numbers and email addresses, the below construct term can be used, which specifies how to create the new document:

```
result [
 mobiles [ all var Mobile ]
]
```

If this is put into a complete construct-query rule it constitutes the first working Xcerpt program:

```
GOAL
 result [
  mobiles [
   all var Mobile
```

```
   ]
  ]
FROM
  addressbook {{
    entry {{
      desc mobile [ var Mobile ]
    }}
  }}
END
```

The result of this query is the anticipated document:

```
result [
  mobiles [
    "0162/4576214",
    "0034−1252−6829",
    "0174/3421390"
  ]
]
```

The next example uses artificial labels for brevity. Note that the data term has two children with identical structure and labels:

```
f [
  g [ a, b ],
  g [ a, b ],
  h [ c, d ]
]
```

The query term shall use the unordered specification, the descendant construct, and a variable. Put into a construct-query rule using a very simple construct term, this yields:

```
GOAL
  result [
    third_term [
      var X
    ]
  ]
FROM
  f {
    g { a, b },
    desc b,
    var X
  }
END
```

The result of this program would be

```
result [
  third_term [
```

```
  h [ c, d ]
 ]
]
```

2.2.3 Matrix Theory

The original matrix—as defined in [13] and implemented by the project chapter [3] and the Xcerpt prototype [7]—has exponential complexity. Submatrices are not shared even where possible and for each level of the matrix an array is allocated with $q \cdot d$ entries. For instance, one of the observations is that most matrices have only a few cells filled with content, the rest of them contain just `false`. Those unnecessary `false`-cells are increasing the complexity and wasting space.

Two improvements discussed in the following section are sharing of submatrices if possible (which turns the matrix tree into a graph of matrices) and keeping only the non-`false` submatrices by using a list in the parent matrix instead of using an array.

2.2.4 Improved Matrix Theory

The more efficient matrix as shown in [5] and implemented in this chapter has a worst-case space complexity of $O(q \cdot d^2)$, q being the number of nodes in the query term and d the number of nodes in the data term.

Briefly, this version of the matrix shares submatrices on the same level if possible. Sharing is possible if the query node and the data node of one matrix match those of another. In this case the matrix for the unification of this query term/data term pair is only created once and pointed to by all other sources.

Furthermore, the use of the fixed-sized arrays is abandoned and replaced by a list of submatrices. Now a matrix only stores those submatrices in the list that are not `false` and it is easy to delete matrices that become `false` in the process of the matrix population. This improvement does not reduce the overall worst-case complexity but results in a reduced time and space usage of the average case.

For a more in-depth explanation, please see [5]. A further refinement of the memoization matrix discussed in [5] is currently under development. Partially based on the results of this chapter, it integrates the algebraic perspective of this chapter with an improved version of the memoization matrix. Details will be available at [1].

2.3 Design Principles of the Single-Rule Xcerpt Abstract Machine

This section briefly lays out the big picture of the inner workings of the single-rule Xcerpt abstract machine.

2.3.1 The Abstract Machine Core

The control loop of the abstract machine is quite simple: the abstract machine fetches one operator after another from the code segment and executes it. The operators access the data term and incrementally build the matrix and finally the result document.

Right now it is assumed that the data term is already on the heap. There are no operators that create an in-memory data term structure and there never will be. The connection of the parser to the abstract machine, which is supposed to create such an in-memory representation, is beyond the scope of this work but is currently under development.

Just like the SUAM, the abstract machine keeps one special feature. It not only allows for several data terms to be queried with one query term translation but also the other way round: It is possible to read and store one data term and then execute several queries on it without having to reparse the data term—none of the operators of the abstract machine modify the in-memory representation of the data term.

2.3.2 Memory Model

The abstract machine abstract machine utilizes two distinct and standard memory areas, a stack and a heap. The data term, the matrix, and the result document are all stored on the heap. The stack stores most of the temporary intermediate results that appear during the evaluation. The code segement is located at the beginning of the stack and is read-only.

2.3.3 Data Model

With the term "data model" this paper refers to the internal data structures and algorithms to represent and handle Xcerpt terms. For the most part, Xcerpt terms can be considered like XML [4] data. There are many XML data models available already today, but none of them is perfect for what the abstract machine needs. It was therefore necessary to implement a data model that fits the needs of this abstract machine.

Nevertheless, a few data models were examined and the design and implementation of the abstract machine data model was finally based on XOM. The following sections give a quick overview of some of the considered data and object models.

Keep in mind over the course of the following discussion that a refined data model has not been the focus of this chapter, but rather the design and implementation of the abstract machine operators. It is expected that a more carefully optimized data model for the abstract machine is going to be investigated in the future.

2.3.3.1 DOM

DOM [10] is a tree-based API. The W3C document object model (DOM) is an API that defines the logical structure of XML documents and the way a document is accessed and manipulated. Essentially, each part of the document like elements and attributes are represented by objects in memory. The objects are linked together using pointers. This is not specified by the W3C but most if not all implementations work that way.

Due to the huge memory requirements of DOM (about 8-20 times more memory than the raw XML document itself [15]), this memory model was out of question right from the beginning. Furthermore, to the best of my knowledge, DOM does not support graphs, it is limited to trees.

2.3.3.2 JDOM and DOM4J

JDOM [11] and DOM4J [12] were both better suited than DOM, but still provided too much functionality. For instance, the abstract machine does not need to modify an XML document at all. Since the

XOM code is easy to understand and contained only a minimal set of functionality, it was preferred to JDOM or DOM4J.

2.3.3.3 XOM

XOM seemed to be the most suited document object model. It is lightweight, easy to understand, and small, it just does not support graphs. So in the beginning the data model of the abstract machine is derived from XOM by stripping the superfluous code and adding support for graphs.

Up to now the code has evolved further so that the similarities with XOM shrunk. There is a lot of room for improvement, so in the future the code will most probably deviate even further from XOM. One of those extensions that need to be implemented is an efficient way to quickly retrieve all or a specific set of descendants of an element. There are a few interesting techniques that could be used, such as 2-hop labels [8], for instance.

2.3.3.4 C++ Data Models

Due to the relatively late decision to also implement the abstract machine in C++, the Java data model was ported to C++ without any significant changes. In contrast to the Java case, there is little consent on a proper XML data model in the C++ area.

2.3.4 Evaluation Models

This section will give an overview of the possible evaluation models for Xcerpt. It implements the first of the three specified evaluation models. Due to time constraints it has not been possible to implement the other two models yet, but this chapter serves as proof of concept and basis for further work on the remaining evaluation models.

All three evaluation models consist of three main evaluation phases: The first phase is the matrix population phase where the matrix is constructed as a compact representation of all possible solutions.

The second phase is the solution generation phase where the matrix is unfolded to yield the full set of solutions. Then arbitrary joins are executed and only those solutions are selected that are needed to construct the result document in the third phase.

And thus, the third phase is the document construction phase where the solutions are embedded into the given result document structure.

The difference between the evaluation models is mostly the way those three phases interact.

2.3.4.1 Set-based Evaluation

The set-based evaluation is the simplest of the three evaluation models. All three evaluation phases are separate from each other, only when the previous phase is completely evaluated the next phase is started.

It immediately follows that this is the evaluation model that needs the most memory since the first phase cannot know which solutions are needed by the second phase and which solutions are not. The same holds for the second phase regarding the third phase.

2.3.4.2 Mixed Evaluation Model

The goal of this evaluation model is to have a transitional solution from the set-based evaluation to the lazy evaluation model. The fundamental idea is to make the solution generation phase implicit

but to keep the matrix population and the document creation phases separate. After the matrix has been created, the document creation phase queries the matrix only for those solutions that it explicitly needs. During this process joins are executed to verify solutions and those solutions are stored in the solution table. Thus, the solution table is incrementally and implicitly generated.

This evaluation model is not as complex as the lazy evaluation (see below), but it remedies the problem of having to unfold the whole matrix, thus reducing the impact of the existential query variables on the complexity of the query evaluation, cf. [5].

2.3.4.3 Lazy Evaluation

The lazy evaluation model integrates and merges all three phases, and, to some extent, reverses them. The reason is that only once the third phase—the document construction phase—requires a binding for an output variable it asks the second phase, which in turn queries the matrix (the first phase), which then goes on to generate any missing parts of the matrix. The result is sent back to the third phase.

The advantage of lazy evaluation is therefore quite obvious, in the best case no memory is wasted and no unnecessary computations are needed. Nothing is evaluated unless it is specifically needed. Theoretically, this evaluation model should yield the least amount of memory usage and the fastest execution. However, in fringe cases the additional overhead for managing and communicating partial evaluations may actually lead to an increase in processing time.

2.4 Operator Semantics under Set-based Evaluation Model

In the next few sections each of the operators that are understood by the abstract machine is defined and described. The following chapter ties these operators to the Xcerpt language as discussed in Section 2.2.1 on page 29 by illustrating the application of those operators for the translation of an Xcerpt query rule.

2.4.1 Overview

Each of the operators is first described more or less informally to give an idea of what the operator does, then the syntax of the operator is defined, and finally a more detailed semantics is given.

There is the query Q with the set of query variables $Vars(Q)$ and the data graph D with the set of nodes N. A single query variable is called v, a data term is called d, and strings are put in quotes like this "string".

A matrix is a triple (q, d, M'), where M' is the set of submatrices. The matrix that the abstract machine creates is called M_G.

2.4.2 Matrix Population

All matrix population operators have access to the one global matrix M_G that is created by them and to the cache σ that is used for fast access to all submatrices. Without σ the matrix would have to be traversed from its root for every access to some submatrix. σ provides constant access to any submatrix (q, d, M') that binds the query variable q to the data term d:

$$\sigma(q, d) = (q, d, M')$$

It is also possible to retrieve the set of matrices M_q that bind a data term to the query variable q:

$$M_q = \sigma(q) = \{(q, d, M') \in \sigma(q, d) | d \in N\}$$

This operation also has constant complexity.

Part of the matrix population operators are a few operators that seem redundant. Those include the "Labeled" version of the operators. Indeed, these operators are redundant in that they can be replaced by two or more other operators. However, they are not redundant in terms of performance. The "Labeled" versions are often preferable because they produce less submatrices (i.e., intermediary results), if the label is selective at all; see also Section 2.5.3.7 on page 52.

RootOperator

Syntax:

```
RootOperator $v_r$ $d_r$
```

Semantics:

The root operator creates the global root matrix $M_G = (v_r, d_r, \emptyset)$. Any previous matrix or cache is discarded. The root matrix maps the query root variable v_r to the root data term d_r. Then the new matrix M_G is stored in the cache, using the query variable and the data term as keys for fast lookups:

$$\sigma(v_r, d_r) = M_G$$

The `RootOperator` always has to be the first operator to be executed since operators cannot be freely reordered yet. See 2.5.3.8 on page 52 for more information on that.

ChildOperator

This operator is used to take all children of the data terms that are bound to the query variable v_p and bind them to the query variable v_c by creating a new matrix for each one of those children.

Syntax:

```
ChildOperator $v_p$ $v_c$
```

Semantics:

First, the child operator retrieves the set of matrices M_{v_p} from the cache σ that match the query variable v_p:

$$M_{v_p} = \sigma(v_p)$$

Each of those matrices has an associated data term the children of which are used to create a new matrix by binding v_c to each one of those children. The new matrices are stored as submatrices in the respective parent matrix that was responsible for those children.

LabeledChildOperator

The `LabeledChildOperator` is a specialized version of the `ChildOperator`. It is used to take only the children that are elements with label `"label"` of the data terms that are bound to the query variable v_p and then bind them to the new query variable v_c.

Syntax:

```
LabeledChildOperator $v_p$ "label"$v_c$
```

Semantics:

Just like the `ChildOperator`, the `LabeledChildOperator` retrieves all matrices from the cache that match the query variable v_p:

$$M_{v_p} = \sigma(v_p)$$

Only those of the associated data terms' children that are elements and match the given label are taken to create a new matrix by binding each of those children to v_c. The new matrices are stored as submatrices in the respective parent matrix that was responsible for those children.

RangedChildOperator

Together with the `NonRelationVerificationOperator`, this operator is used to translate a part of the negation in ordered queries.

Informally, this operator gathers for each data term d that is associated to one of the matrices $\sigma(v_p)$ the children of d that are between the data terms associated to the query variable v_{start} and v_{end}. Those data terms are bound to the query variable v_c with a new matrix.

Syntax:

```
RangedChildOperator $v_p$ $v_{start}$ $v_{end}$ $v_c$
```

Semantics:

This operator walks through all matrices $\sigma(v_p)$ and looks for submatrices that match v_{start}. Those submatrices need to have a further submatrix that matches v_{end}. All children between the data term of the former submatrices and the latter ones are stored in new submatrices of v_{end}.

The prerequisite for this operator to work is that the query node v_{parent} is ordered and that the matrix of v_{start} is a parent of the matrix for v_{end}—just like it is created by the `FollowingSiblingOperators`. This is to ensure that there really is a data term, referenced by the matrix v_{start}, that is a predecessor for the data term referenced by the matrix v_{end}. This prerequisite is guaranteed by the translator. If the prerequisite is not met, then this operator will do nothing.

In the same way the result is stored: every child that is found between v_{start} and v_{end} will be stored in a submatrix of the matrix v_{end} to record the relation.

LabeledRangedChildOperator

This operator is a specialized version of the `RangedChildOperator`. It only operates on those children that are elements and match the given label "label".

Syntax:

```
LabeledRangedChildOperator $v_p$ $v_{start}$ $v_{end}$ "label"$v_c$
```

Semantics:

For further details, see the `RangedChildOperator` above, apart from the additional restriction of the children, this operator has exactly the same semantics.

FollowingSiblingOperator

This operator is used to take those children of the data terms bound to the query variable v_p that follow the reference data term identified by v_{ref} and bind them to the query variable v_s by creating a new matrix for each one of those children.

Syntax:

```
FollowingSiblingOperator $v_p$ $v_{ref}$ $v_s$
```

Semantics:

This operator walks through all matrices in $\sigma(v_p)$ and looks for submatrices M' that match v_{ref}. Then all children of the current matrix of $\sigma(v_p)$ that are following siblings of the data term corresponding to M' are collected, bound to v_s, and stored under M' as further submatrices.

LabeledFollowingSiblingOperator

The `LabeledFollowingSiblingOperator` is a specialized version of the `FollowingSiblingOperator` above. It only takes those siblings that are elements with label "label".

Syntax:

```
LabeledFollowingSiblingOperator $v_p$ $v_{ref}$ "label"$v_s$
```

Semantics:

For further details, see the `FollowingSiblingOperator` above. This operator restricts the siblings to elements with the given label.

DescendantOperator

The `DescendantOperator` is used to retrieve all descendants of the data terms bound to the query variable v_p and to bind them to the query variable v_{desc}.

Syntax:

```
DescendantOperator $v_p$ $v_{desc}$
```

Semantics:

$$\{(v_p, d, M' \cup M'') : (v_p, d, M') \in \sigma(v_p) \wedge M'' = \{(v_{desc}, n, \emptyset) : n \in desc(d)\}\}$$

First, the `DescendantOperator` retrieves all matrices M_{v_p} from the cache σ that match the query variable v_p:

$$M_{v_p} = \sigma(v_p)$$

Each of those matrices has an associated data term d. This operator takes *all* the descendants of the data terms d and binds them to the query variable v_{desc} by creating a new matrix $M' = (v_{desc}, d, \emptyset)$. Those matrices are added as submatrices to that matrix of M_{v_p} the data term of which they belong to.

DescendantOrSelfOperator

The `DescendantOrSelfOperator` is a specialized version of the `DescendantOperator`. It not only retrieves all descendants of a given parent data term, but also includes the parent data term itself.

Syntax:

```
DescendantOrSelfOperator $v_p$ $v_{desc}$
```

Semantics:

$$\{(v_p, d, M' \cup M'') : (v_p, d, M') \in \sigma(v_p) \wedge M'' = \{(v_{desc}, n, \emptyset) : n \in (desc(d) \cup d)\}\}$$

LabeledDescendantOperator

The `LabeledDescendantOperator` is a specialized version of the `DescendantOperator`. It only retrieves those descendants of a given parent data term that are elements that match a given label.

Syntax:

```
LabeledDescendantOperator $v_p$ "label"$v_{desc}$
```

Semantics:

$$\{(v_p, d, M' \cup M'') : (v_p, d, M') \in \sigma(v_p) \wedge M'' = \{(v_{desc}, n, \emptyset) : n \in desc(d) \wedge label(n) = "label"\}\}$$

LabeledDescendantOrSelfOperator

The `LabeledDescendantOrSelfOperator` is a specialized version of the `DescendantOrSelfOperator`. It only retrieves those descendants of a given parent data term, including the parent data term itself, that are elements that match a given label.

Syntax:

```
LabeledDescendantOrSelfOperator $v_p$ "label"$v_{desc}$
```

Semantics:

$$\{(v_p, d, M' \cup M'') : (v_p, d, M') \in \sigma(v_p) \wedge M'' = \{(v_{desc}, n, \emptyset) : (n \in desc(d) \cup d) \wedge label(n) = "label"\}\}$$

Note that not only the true descendants have to match the given label, but also the parent node d is only bound to v_{desc} if it matches the label "label".

Verification Operators

All operators in the remainder of this section are part of the matrix population operators, but serve a special need: they verify if the created matrix associated with a particular query variable meets certain requirements. If this is the case, nothing else is done, if this is not the case, then the affected matrix is set to `false`.

TypeVerificationOperator

The `TypeVerificationOperator` verifies that the data terms bound to the given query variable are of a given type.

Syntax:

```
TypeVerificationOperator $v$ $type$
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that their type matches the given type $type$.

The type $type$ can be either one of the following:

- Node
- Document
- Element
- Text
- Comment
- ProcessingInstruction
- DocType

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

LabelVerificationOperator

The `LabelVerificationOperator` verifies that the data terms bound to the given query variable match a given label.

Syntax:

```
LabelVerificationOperator $v$ "label"
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that they are elements with label "label".

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

ContentVerificationOperator

The `ContentVerificationOperator` verifies that the data terms bound to the given query variable are text nodes with a specific content.

Syntax:

```
ContentVerificationOperator $v$ "content"
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that they are text nodes with content matching the string "content".

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

ArityVerificationOperator

The `ArityVerificationOperator` verifies that the data terms bound to the given query variable have a fixed number of children.

Syntax:

```
ArityVerificationOperator $v$ $arity$
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that they are elements with exactly $arity$ number of children. $arity$ may be any positive integer, including 0.

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

OrderVerificationOperator

The `OrderVerificationOperator` verifies that the data terms bound to the given query variable have their children ordered or unordered.

Syntax:

```
OrderVerificationOperator $v$ $ordered$
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that they are elements the children of which are either an ordered or unordered sequence. $ordered$ may be either `true` or `false`.

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

NamespaceVerificationOperator

The `NamespaceVerificationOperator` verifies that the data terms bound to the given query variable match the given namespace declaration.

Syntax:

```
NamespaceVerificationOperator $v$ $URI$ $prefix$
```

Semantics:

Upon execution this operator retrieves the data terms associated with the matrices $\sigma(v)$ and verifies that the namespace of those nodes matches the given namespace definition: the namespace name or prefix must be equal to $prefix$ and the associated URI must be equal to URI.

If the verification of any of those data terms fails, the corresponding matrix is set to `false`.

RelationVerificationOperator

The `RelationVerificationOperator` sets itself a little apart from the other verification operators: it verifies that all matrices identified by a given query variable have at least one submatrix for each further given query variable v_r.

Syntax:

```
RelationVerificationOperator $v$ $v_{r_1}, v_{r_2},\ldots$
```

Semantics:

Upon execution this operator checks that all matrices $(v, d, M') \in \sigma(v)$ contain submatrices that bind the query variables $v_{r_1}, v_{r_2}, \ldots$:

$$\forall (v, d, M') \in \sigma(v) \quad \forall v_r \in \{v_{r_1}, v_{r_2}, \ldots\} \quad \exists d' : (v_r, d', M'') \in M'$$

If the verification of any of those matrices fails, the matrix is set to `false`.

NonRelationVerificationOperator

The `NonRelationVerificationOperator` is the opposite of the above `RelationVerificationOperator`. It verifies that none of the matrices identified by a given query variable have a submatrix that binds one of the given query variables v_r.

Syntax:

```
NonRelationVerificationOperator $v$ $v_{r_1}, v_{r_2},\ldots$
```

Semantics:

Upon execution this operator checks that none of the matrices $(v, d, M') \in \sigma(v)$ contains a submatrix that binds one of the query variables $v_{r_1}, v_{r_2}, \ldots$:

$$\forall (v, d, M') \in \sigma(v) \quad \forall\, v_r \in \{v_{r_1}, v_{r_2}, \ldots\} \quad \nexists d' : (v_r, d', M'') \in M'$$

If the verification of any of those matrices fails, the matrix is set to `false`.

2.4.3 Relational Solution Evaluation

The pure relational solution evaluation is used by the set-based evaluation model. This section describes the operators that create the relational solution table.

The operator that has always to be executed first in this evaluation phase is the `CartesianProductOperator`. It creates the initial relational table on which the following operators perform their work.

CartesianProductOperator

The `CartesianProductOperator` flattens the matrix M_G generated by the matrix population operators and stores the results in the relational solution table.

Syntax:

```
CartesianProductOperator $v_i, v_j,\ldots$
```

Semantics:

Create the set of *all* possible solutions that the matrix represents. The list of variables $v_i, v_j, \ldots$ determines which variables are supposed to be output variables, that is, which variables the caller is interested in.

A single solution is a set of value assignments, one binding for each variable.

Each (sub)matrix represents a set of (partial) solutions. The matrix itself contains just the one variable-node-binding that it was created for. But there are many other variable bindings that depend on this very binding, and those are represented by the matrix's submatrices and their combinations.

Those submatrices are grouped by their respective variables; each group represents one set of alternative bindings for that variable and its corresponding subvariables (submatrices).

All of those binding alternatives that one submatrix represents have to be combined with all of the alternatives of each one of the other variable groups. All those combinations together comprise the possible solutions for the matrix and its submatrices.

It is therefore obvious that the final table of solutions needs exponential space compared to the matrix from which it is created (and thus compared to the document size).

IdentitySelectionOperator

This operator is a purely relational operator. Its only input is the relational solution table and its only output is the modified solution table.

Syntax:

```
IdentitySelectionOperator $v_1$ $v_2$
```

Semantics:

This operator selects only those tuples of the result table that have identical bindings for the variables v_1 and v_2 and removes the other tuples from the table.

ValEqualSelectionOperator

Just like the `IdentitySelectionOperator`, this operator is a purely relational operator. Its only input is the relational solution table and its output is the modified solution table.

Syntax:

```
ValEqualSelectionOperator $v_1$ $v_2$
```

Semantics:

This operator selects only those tuples of the result table that have value equal bindings, i.e. matching terms, for the variables v_1 and v_2 and removes the other tuples from the solution table.

2.4.4 Document Construction

The document construction is the phase that takes the computed variable bindings and builds the result document according to the construct operators. These operators work a little different than the matrix population operators and the relational operators. The document construction operators are nested, their work depends on the parent operator. The parent operator calls its child operators with the appropriate element $parent$ that needs to be extended.

Thus, the document construction operators form a tree. The root of that tree is the operator that creates the root element of the result document.

The document construction operators keep a global reference to the current tuple c in the solution table. It is only changed by the `AllIterationOperator` and read by the `VariableOperator` so that the `VariableOperator` can access a(nother) binding for its query variable.

The implemented document construction is solely based on the relational solution table and requires the matrix to be completely unfolded by the `CartesianProductOperator`.

ElementOperator

The `ElementOperator` creates a new element in the context of its parent operator.

Syntax:

```
ElementOperator "label"$ $ $ordered$ ( $[DCOP_1$ $DCOP_2$ $\ldots]$ )
```

Semantics:

This operator creates a new and empty element with the label "label". The new element is appended to the list of children of the element $parent$, implicitly given by the parent operator. The new element serves as parent element for the suboperators $DCOP_i$. The argument $ordered$ can be `true` or `false` and specifies if the order of the children matters, that is, if the children are ordered or not.

AttributeOperator

The `AttributeOperator` creates a new attribute for the element of its parent operator.

Syntax:

```
AttributeOperator $name$ $value$
```

Semantics:

This operator adds a new attribute with the identifier $name$ and the content $value$ to the element $parent$, which is given by the parent operator.

This operator cannot have any further suboperators.

VariableOperator

The `VariableOperator` reads the current binding for the given query variable and adds the referenced data term to the list of children of the element given by the parent operator.

Syntax:

```
VariableOperator $v$
```

Semantics:

This operator takes the data term that is bound to the query variable v by the current tuple c and appends it to the list of children of the element $parent$, implicitly given by the parent operator.

A `VariableOperator` operator cannot have any further suboperators.

AllIterationOperator

This operator iterates through all distinct combinations of bindings of the given query variables in the solution table and executes all its suboperators for each of those combinations.

Syntax:

```
AllIterationOperator $v_{i_1}, v_{i_2}, \ldots$ ( $[DCOP_1$ $DCOP_2$ $\ldots]$ )
```

Semantics:

The `AllIterationOperator` (also called the **`all`**-iterator) executes its suboperators on all tuples of the solution table that are unique in the bindings and their combination of the given set of variables $\{v_{i_1}, v_{i_2}, \ldots\}$. Before executing the child operators on a selected tuple t, the **`all`**-iterator sets the current tuple c to this selected tuple t.

In addition to that, the **`all`**-iterator operates in the context of its nearest ancestor **`all`**-iterator, if there is such an operator. That means that if there is a parent `AllIterationOperator` that iterates over some given variables, those variables constitute the context for the closest following child or descendant **`all`**-iterator. This child **`all`**-iterator then operates only on those tuples of the solution table that have the same bindings for the context variables as the current tuple c.

2.5 Translation of Xcerpt Single Rule Programs

This chapter is a more or less informal description of the translation of an Xcerpt single rule program. It is not a mathematically precise definition of a translation function. However, this chapter should be sufficient to understand the translation, to implement it, and to create the formal function.

First of all, recall that the abstract machine only handles Xcerpt single rule programs. In the following, such a program is simply called *rule*.

A rule consists of a query term and a construct term. Currently, the abstract machine supports only one query term in the rule since the constructs **`and`** and **`or`**, which can combine several query terms, are not supported.

The translation process of a rule is three-fold. First, the spanning tree of the query term is translated, then the remaining graph relations are translated, and finally the construct term is translated.

The query term binds certain subterms of the data term to some variables and with the help of the document structure given in the construct term, the construction part reassembles those data terms into the result document.

2.5.1 Basics

Every query term can be represented by a graph, called query graph. Each node in such a graph is assigned a so-called query variable. In the following, the query variables are called v_i, with $0 < i <= n$, and n is the number of nodes in the query term.

Variables that appear in the query term are called output variables and since those output variables correspond to a node as part of the query graph, they too are assigned a query variable.

Note, however, that even if one and the same output variable occurs more than once in the query term, it is still assigned two different query variables. Query variables are unique and refer to one particular node in the query graph. This also means that variables in the query term (the output variables) are treated just like any other query node. All of the translation is based on this concept.

Also note that due to the different operation of the document construction operators, the nodes of the construct term are not assigned any query variables. Rather, the generated machine code of the construct term *uses* the query variables already assigned to the query nodes.

The remainder of this chapter is divided according to the three evaluation phases. They are represented by the translation and the operators, and the following three sections correspond to the matrix population phase, the solution generation phase, and the document construction phase.

2.5.2 Scope and Naming

This section specifies which parts of the Xcerpt language are understood by the abstract machine and can be translated, and some of the naming used in this chapter.

All simple query terms without special Xcerpt keywords are fully supported and can be translated, including graphs, ordered and unordered child specifications, and complete and incomplete term specifications.

Variables and variable restrictions can be used, and the "$*$"-wildcard for labels is supported.

Regarding Xcerpt keywords, the following subset is fully supported: **`desc`**, **`descendant`**, *`var`*, **`variable`**, *`as`*, **`all`**. Only partially supported is the **`without`** keyword.

Generally, single Xcerpt rules with query part and construction part are supported. Chaining of rules is not covered by this initial version of the abstract machine.

Currently, only element nodes, attribute nodes, and text or content nodes are supported. Comments, document types, and processing instructions are not implemented, but are straightforward to add using the existing code as a starting point.

Irrespective of the children of an element being specified completely or incompletely, the set of children of an element is called a *sequence*. Depending on the order, the sequence is called *unordered* sequence or *ordered* sequence.

2.5.3 Translation of the Query Part

The resulting machine code of the translation of the query part of an Xcerpt query rule constitutes the first phase of the evaluation, the matrix population phase. The matrix population operators are divided into two groups: operators that create and add new submatrices and operators that set matrices to `false` and delete them if certain properties are not met. The first group of operators includes the following operators:

- `ChildOperator`
- `FollowingSiblingOperator`
- `RangedChildOperator`
- `DescendantOperator`

together with their specialized operators. The second group of operators are all those operators the names of which end with `VerificationOperator`.

2.5.3.1 Translation of the Query Root Node

The root node of the query term is always translated with the `RootOperator` and it is always translated first. The arguments for the `RootOperator` are the query variable for the root node of the Xcerpt query term v_1 and the root node of the data term.

```
RootOperator $v_1$ <pointer_to_data_term>
```

2.5.3.2 Translation of a Node

There are a few different cases as to how a query node is translated. The translation mainly depends on the type of the node.

Text Nodes

Syntax: "content"

A text node (also called content node), which was assigned the query variable v_c, is translated with the following code:

```
ContentVerificationOperator $v_c$ "content"
```

`content` is a conceptual variable and is replaced by the actual string.

Elements

Syntax: `id@nsprefix:label [sequence]`

If the node is an element having the query variable v_e assigned, and with a label, a namespace, an ID, and possibly a sequence if it has children, then the translation is:

```
TypeVerificationOperator $v_e$ ElementType
NamespaceVerificationOperator $v_e$ URI nsprefix
LabelVerificationOperator $v_e$ label
```

[2]

Note that the ID is not directly translated, the translator just stores the mapping from the ID to the query variable of the query node and replaces this ID with the query variable when another term references the ID. For the translation of the children of the element, see below.

2.5.3.3 Translation of the Children of an Element

Syntax: `id@nsprefix:label [sequence]`

The translation of the children of an element query node v_p depends on if the children are ordered or not. The children are contained in a sequence:

```
$v_p$ [ $c_1, c_2,\ldots, c_n$ ]
```

with n being the number of children in the sequence or the arity of the parent query term v_p. As a reminder, a sequence's properties are specified using different brackets: ordered and total sequences are denoted by `[]`, ordered and partial sequences are denoted by `[[]]`, unordered and total sequences are denoted by `{}`, and unordered and partial sequences are denoted by `{{}}`.

Translation of Unordered Sequences

```
OrderVerificationOperator $v_e$ false
```

If the sequence is total, i.e., completely specified, then the following additional code is added:

```
ArityVerificationOperator $v_e$ $n$
```

[1]

As an optimization, in case of an incomplete sequence specification a new operator could be added that checks for an arity greater than or equal to the given arity n.

Now, assuming that no further Xcerpt keywords appear in the sequence, all children are translated using the `ChildOperator`. Thus, for each child v_c of the n children the following code is generated:

```
ChildOperator $v_p$ $v_c$
```

Following that is the code of the translation of the child term itself. If one or more of the children is declared to be a descendant using the **`desc`**-keyword, then the code above is replaced by

```
DescendantOperator $v_p$ $v_c$
```

Nothing else changes.

Translation of Ordered Sequences

```
OrderVerificationOperator $v_e$ true
```

If the sequence is total, i.e., completely specified, then the following additional code is added:

```
ArityVerificationOperator $v_e$ $n$
```

Again, assuming that no further Xcerpt keywords are used in the sequence of children, the resulting code of the translation of the sequence is:

```
ChildOperator $v_p$ $c_1$
FollowingSiblingOperator $v_p$ $c_1$ $c_2$
FollowingSiblingOperator $v_p$ $c_2$ $c_3$
$\dots$
FollowingSiblingOperator $v_p$ $c_{n-1}$ $c_n$
```

Of course, for each of the children the code for translating that child is inserted as well.

Since in the ordered and total case there is only exactly one matching possibility, instead of using the `FollowingSiblingOperator`s, a new kind of operator could be introduced, the `ImmediatelyFollowingSiblingOperator`. This operator would only create new matrices for the immediately following sibling, not all following siblings, like the `FollowingSiblingOperator` does.

And just like in the unordered case, an optimization for the incomplete or partial sequence specification would be a new operator that checks for an arity greater than or equal to the given arity n.

For the case of descendants in an ordered sequence, please refer to the next section.

2.5.3.4 Translation of the Descendant Construct

There are two cases in the translation of the **`desc`** keyword: the descendant construct can be part of an unordered sequence or it can be part of an ordered sequence.

As mentioned already, the descendant translation in unordered sequences is straightforward. Instead of the `ChildOperator`, the `DescendantOperator` is used.

However, the ordered case is a little bit trickier. It has to be made sure that the descendant term that was found is a descendant of one of the nodes that come *after* the previously translated child (or descendant) in the order of the sequence. One way to ensure that is to introduce an additional query variable v_t that is bound to the following siblings of the previously translated term. Then, the actual query variable of the descendant construct to be translated is bound to the descendants of those siblings, or the siblings themselves, which were bound to the additional query variable v_t. Therefore, the translation of the following sequence part:

```
$v_p$ [ $\ldots$, $v_x$, desc $v_y$, $\ldots$ ]
```

results in the following code:

```
$\dots$
<translation of the child $v_x$>

FollowingSiblingOperator $v_p$ $v_x$ $v_t$
```

```
DescendantOrSelfOperator $v_t$ $v_y$
$\dots$
```

And then follows the translation of the descendant term itself, and after that the translation of the rest of the sequence.

2.5.3.5 Back-Propagation of `false` Matrices

In the process of building the matrix, all operators of the `ChildOperator`-family, the `DescendantOperator`-family, and the `FollowingSiblingOperator`-family do not create any new submatrices if the corresponding data term does not fulfill the requirements of those operators.

Similarly, the operators of the `VerificationOperator`-family may set a submatrix to `false`.

At the end of the translation of the query part of the Xcerpt query rule, the generated matrices that are `false` or the matrices that are invalid matrices because they should have but do not have certain submatrices, need to be propagated back up in the matrix tree. Naturally, back-propagation needs to be done in reverse order, that is, the leaves of the matrix tree need to be checked first, and `false` matrices need to be propagated up until at last the root is reached.

For each `ChildOperator`, `RangedChildOperator`, `DescendantOperator`, and `FollowingSiblingOperator` there has to be at least one `RelationVerificationOperator`, or `NonRelationVerificationOperator` in case of the `RangedChildOperator`.

The `RelationVerificationOperator`s have to be in reverse order of the order of their corresponding child- or descendant-operators.

Therefore, as an example, if the query to be translated is

```
$v_1$ {{ $v_2$, $v_3$ {{ $v_4$ }} }}
```

then the resulting code is:

```
ChildOperator $v_1$ $v_2$
ChildOperator $v_1$ $v_3$
ChildOperator $v_3$ $v_4$

RelationVerificationOperator $v_3$ $v_4$
RelationVerificationOperator $v_1$ $v_3$
RelationVerificationOperator $v_1$ $v_2$
```

2.5.3.6 Translation of Negated Subqueries

Negated subterms are only partially supported. Only one special case has been implemented: negated subterms must be part of a partial and ordered sequence and must not contain any output variables.

The **`without`** keyword is translated using the `RangedChildOperator` and the `NonRelationVerificationOperator`.

The following sequence

```
... $v_1$ [[ a, without b, var X as b ]]
```

is translated to

```
ChildOperator $v_1$ $v_2$
LabelVerificationOperator $v_2$ "a"

FollowingSiblingOperator $v_1$ $v_2$ $v_4$
```

```
LabelVerificationOperator $v_4$ "b"

RangedChildOperator $v_1$ $v_2$ $v_4$ $v_3$
LabelVerificationOperator $v_3$ "b"
NonRelationVerificationOperator $v_4$ $v_3$

RelationVerificationOperator $v_2$ $v_4$
RelationVerificationOperator $v_1$ $v_2$
```

Intuitively, the query means: "Get all those bs that immediately follow an a in the sequence of children of v_1 and bind them to the output variable X."

The matrices for the query variable v_4 (the second "b") have to be created before v_3 can be computed because v_3 depends on v_4.

The `NonRelationVerificationOperator` ensures that no term is bound to v_3 as a submatrix of v_4.

2.5.3.7 Optional Optimizations During Translation

There are a few optimizations the translator can but need not perform. One optimization concerns the label verification of children and descendants. Whenever code like this is generated as per the above logic:

```
ChildOperator $v_p$ $v_c$
LabelVerificationOperator $v_c$ label
```

i.e., whenever a `ChildOperator` is followed by a `LabelVerificationOperator`, the code can be replaced by

```
LabeledChildOperator $v_p$ label $v_c$
```

The same holds for the `FollowingSiblingOperator`, the `DescendantOperator`, the `DescendantOrSelfOperator`, and the `RangedChildOperator`. This optimization can in some cases be quite drastic: instead of creating matrices for *all* children first and then discarding those matrices the children of which are not elements or do not match the label, this optimization only creates matrices if the nodes are elements that match the given label.

In the future, there may be more shortcuts like these. It should be noted though that the lazy evaluation model will render those kinds of optimizations obsolete.

Another optimization is the increased use of the `RelationVerificationOperator`. Instead of only verifying a relation at the end of the query term translation, the `RelationVerificationOperator` can additionally be used right after any one of the matrix population operators that create new submatrices. This pushes the selection further up in the operator tree and eliminates irrelevant tuples earlier.

As it was mentioned in Section 2.5.1 on page 47, the abstract machine treats all variables the same. It could be beneficial to treat query subterms without output variables existentially. That is, as soon as one matching data term is found any further matrix creation is stopped for this subquery and the corresponding matrix branch is set to `true`.

2.5.3.8 Limitations for the Translator

Currently, the translator does not perform all optimizations one could possibly think of. The most prominent limitation is the free reordering of operators: with the current operator implementation it

is not possible to execute the operators in an arbitrary order. An operator that needs a query variable to compute its output, like the `ChildOperator`, for instance, cannot be moved in front of the operator that creates the matrices for that query variable.

A further refinement of the operators and the matrix that makes it possible to freely reorder the operators is currently under consideration.

2.5.4 Assigning the Output Variables

Translation of the output variables means creating the code that evaluates the graph relations of the Xcerpt query term. All the code up to this point dealt with the spanning tree of the query term.

In the set-based evaluation model, this produces the code that represents the second phase of the evaluation, which is the solution generation phase. It completely unfolds the matrix and stores the resulting solutions in the relational solution table.

The first operator of the translation of the output variables is always the `CartesianProductOperator` with the query variables for all output variables.

```
CartesianProductOperator $qOut$
```

Recall that each output variable is assigned a query variable. $qOut$ are all those query variables that stand for an output variable.

Only if there are joins to be computed further code is generated. Joins are the graph relations of the query term and are identified by the same output variable occurring more than once in the query term. Recall that two identical output variables are assigned two different query variables for two different positions in the query graph. Those two corresponding query variables have to be "joined", that is, only those tuples of the relational solution table are selected that have the same binding for the two query variables. The type of equality can be selected; available types are value equality and true identity.

For value joins of two query variables, the translated code is

```
ValEqualSelectionOperator $q_x$ $q_y$
```

and for identity joins it is

```
IdentitySelectionOperator $q_x$ $q_y$
```

2.5.5 Translation of the Construct Part

The last part of the translation is marked by the translation of the construct part of the Xcerpt query rule. This section deals with the creation of the result document, and the generated abstract machine code constitutes the last phase of the abstract machine, the document construction phase of the evaluation. The solutions of the previous phase are assembled into the result document using the document structure given in the construct part of the query rule.

Compared to the translation of the other two parts the operators of the document construction are nested and make up a hierarchy. This is required because the nodes of the construct term are not assigned to query variables, nor is the construct term's tree structure recorded anywhere.

Elements, attributes, and namespaces are translated into `ElementOperator`, `AttributeOperator`, and `NamespaceOperator`, respectively:

```
  f[ m[ a, b ] ]
```

is translated to:

```
ElementOperator "f" true (
      ElementOperator "m" true (
            ElementOperator "a" true ()
            ElementOperator "b" true ()
      )
)
```

Variables are translated into `VariableOperator` and the **`all`** is translated into `AllIterationOperator`. The arguments to the `AllIterationOperator` are all those variables in the construct term that are a descendant to the **`all`** and do not have a closer parent **`all`**. See the next section for some example translations.

2.5.6 Example Translation

After all the theoretical translation knowledge, this section shows an example for the translation of some of the supported features.

Query Rule

The Xcerpt query rule to be translated is:

```
GOAL
   f[ all m[ var X, all var Y ] ]
FROM
   f[[ a[[ var X as b ]], var Y as c, var X ]]
END.
```

abstract machine Code

```
// first phase
RootOperator $v_1$ f
LabelVerificationOperator $v_1$ "f"
OrderVerificationOperator $v_1$ true

LabeledChildOperator $v_1$ "a" $v_2$
OrderVerificationOperator $v_2$ true
RelationVerificationOperator $v_1$ $v_2$

ChildOperator $v_2$ $v_3$
LabelVerificationOperator $v_3$ "b"

LabeledFollowingSiblingOperator $v_1$ $v_2$ "c" $v_4$
FollowingSiblingOperator $v_1$ $v_4$ $v_5$

RelationVerificationOperator $v_4$ $v_5$
RelationVerificationOperator $v_2$ $v_3$ $v_4$
RelationVerificationOperator $v_1$ $v_2$

```

```
// second phase
CartesianProductOperator $v_3$ $v_4$
ValEqualSelectionOperator $v_3$ $v_5$


// third phase
ElementOperator "f" true (
      AllIterationOperator $v_3$ (
            ElementOperator "m" true (
                  VariableOperator $v_3$ ()
                  AllIterationOperator $v_4$ (
                        VariableOperator $v_4$ ()
                  )
            )
      )
)
```

2.6 Implementation

2.6.1 Object-oriented Design of the Implementations

The following sections show the common design of the two abstract machine implementations, the Java and the C++ implementation.

2.6.2 Coding Conventions

Some coding conventions have been decided upon to make the code as readable as possible. Of those, the following are worth mentioning.

Member variables are prefixed with "`m_`" and static variables are prefixed with "`s_`". The rest of the code uses camel case. Classes start with a capital letter, methods and variables with a small letter.

2.6.3 Abstract Machine Core Cycle

The abstract machine Core Cycle is at the heart of the abstract machine and therefore rather simple. It executes one operator after another, which looks like this in Java:

```
for ( Operator op : query )
{
   op.execute();
}
```

And `query` is a container with the sequence of operators to be executed by the abstract machine.

2.6.4 Class-Hierarchy

Besides the main program, the abstract machine consists of the data model, the matrix, and the operators. The matrix and the operators together make up the conjunctive query pattern engine, which solves Xcerpt queries.

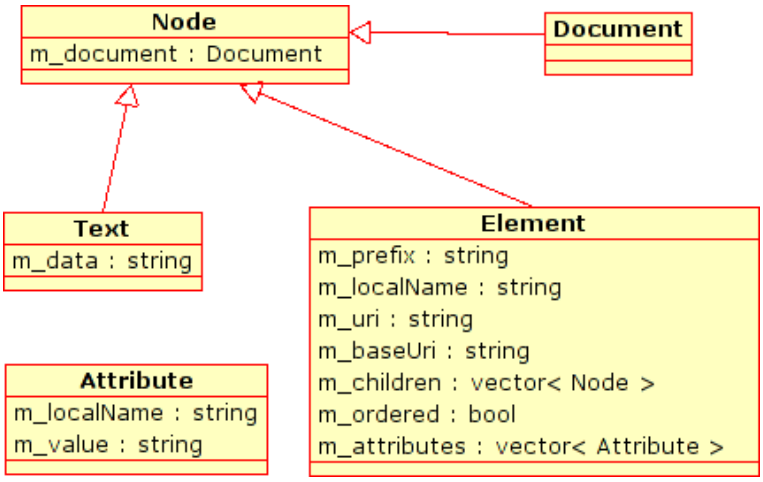

Figure 2.1: The Class Diagram of the abstract machine Data Model

The class graph of the main classes of the abstract machine is split in two: in Figure 2.1 can be found the UML diagram of the data model, Figure 2.2 on the next page shows the design of the engine.

2.6.4.1 Data Model

The central class of the data model is the `Node`-class. Except for the `Attributes`, all other classes inherit `Node` and, therefore, *are* `Nodes`.

To prevent endless casting of `Node` objects to `Element` or any of the other classes when calling, say, `Element` methods, most methods are available in the base class `Node` already. The default implementation of most of those methods throws an `UnsupportedOperationException` and is overridden by those classes where that particular method makes sense.

2.6.4.2 Engine

The engine is mainly based on four classes: `Matrix`, `MatrixPopulationOperator`, `RelationalOperator`, and `DocumentConstructionOperator`. The latter three classes represent the three evaluation phases and are derived from the interface `Operator`, since obviously those are all operators and a common superclass allows for all operators to be stored in one Java or C++ container. The actual operators are all derived from one of those three abstract operator classes.

2.6.5 Some Specific Operator Details

2.6.5.1 Document Construction Operators

The **`all`**-iterator (the `AllIterationOperator`) passes the current tuple via a class-global variable, a static class member called `s_currentTuple`. This is necessary because all suboperators (that are `VariableOperators`) of the `AllIterationOperator` at any depth need to be able to access it. Without the static member the current tuple would have to be passed through all the other operators and would make their API a bit ugly. The static class member is a cleaner solution.

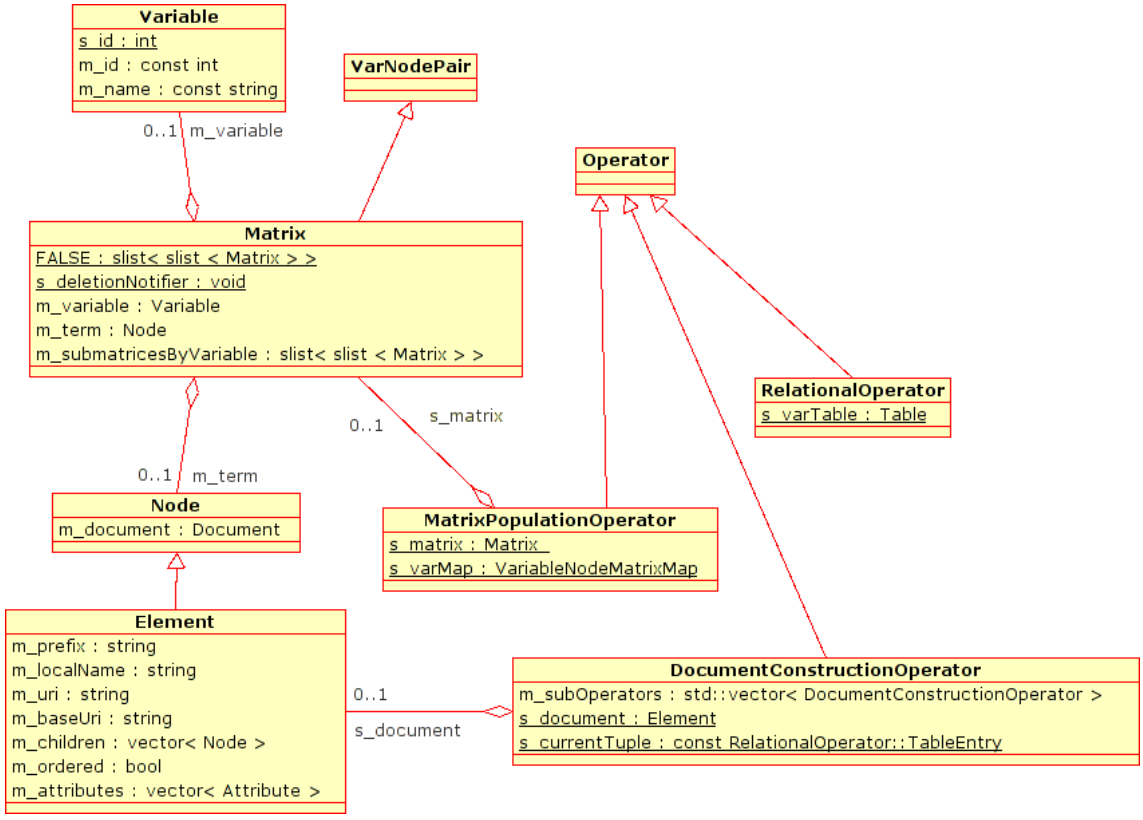

Figure 2.2: The Class Diagram of the abstract machine Engine

2.6.6 Problems Encountered and their Solutions

First and foremost was a problem that arouse during the implementation of the matrix expansion in the `CartesianProductOperator`. Recall that a solution is a list of variable bindings, which contains all query variables. All solutions are gathered in the solution table, which is also a list.

Generating this list of all possible solutions took far too much memory to be acceptable. This has first been observed in the Java implementation. The reason was that for every variable binding a new list element was created and for every solution all variable bindings were created again. On top of that, Java only provides a doubly-linked list, wasting memory with each and every variable binding and solution. It is not required to be able to walk backwards in either of those lists.

The solution to that was to implement a singly linked list with the possibility to share list tails, reducing heap memory allocation a lot. And it is possible to share the list tails because additional variable bindings are only prepended at the beginning of a list of bindings. Furthermore, the solutions do not have to be modified, making the list tails conceptually read-only.

This implementation improved the `CartesianProductOperator` dramatically, its speed increased by a factor of 2-4 and the memory usage was cut in less than half.

2.7 Java Implementation

The Java implementation uses the standard Java libraries, but mainly the containers from the `java.util` package.

The Java implementation has one special feature. It implements a singly linked list that supports tail sharing. This linked list implementation could probably be re-used for other projects as well.

Apart from that, the Java implementation conforms to the above design description with no deviations.

2.7.1 Drawbacks of the Java Implementation

Initially the first implementation of the abstract machine was done in Java only. However, it became clear that the exhibited performance cannot be the summit of possibility. Furthermore, some limitations of the Java language itself lead to the decision to port the implementation to C++ as well.

The main drawback of the Java implementation is its memory footprint and the execution speed. For a comparison of the Java and the C++ implementation see Section 2.9 on page 61.

Related to that is Java's limitation regarding objects: it is not possible to retrieve an object's physical address. If, say, we need a list of objects to be sorted so that equal objects are consecutive those objects need to be comparable. In case of data terms this is not possible, so one has to add a new integer class member to indicate the serial number and then sort according to the object's serial number. This is a little slower than the direct object addresses of C++ and needs more memory—the serial number is a per-object cost that is to be paid.

Other than that, the container library is not quite as flexible and powerful as the C++ STL, but that was no major problem to overcome.

2.8 C++ Implementation

The C++ implementation is based on the STL (Standard Template Library) and uses the stlport library, version 5.0. The reason for not using the GCC STL is efficiency—for some reason the GCC STL needs

dramatically more memory and is significantly slower. This concerns the GCC STL version 4.1.

The following section documents only those details that are peculiar to the C++ implementation.

2.8.1 A Singly-Linked List With Tail Sharing: `sslist`

The C++ implementation features a singly linked list implementation that supports tail sharing and is compatible with the STL. That means it supports the STL allocators, it uses the STL iterators, and can be used with many of the STL algorithms.

Care needs to be taken though when modifying an `sslist`'s structure. The `sslist` implementation does not check if the modified list element is shared or not. In the worst case, another list can unintentionally and implicitly be modified and a memory leak could be introduced.

Also, not all list operations specified by the ISO C++ standard have been implemented in the `sslist`. To reduce the resulting binary size, only those operations that are needed by the abstract machine are supported.

2.8.2 Memory Management

2.8.2.1 General Overview

In contrast to Java, C++ does not provide an automatic garbage collection. The advantage of that is performance, both in terms of speed and memory usage, and the obvious disadvantage is the additional work for the programmer with possible mistakes and resulting memory leaks. However, the fact that the programmer *can* implement his own garbage collection is also an advantage.

Manual garbage collection needs less overall memory (in fact, it needs the minimum amount of memory for that application) because the exact moment when an object can be deleted is known by the programmer. Thus, the problem of using excess memory when creating many temporary objects very fast cannot arise. Most of the time the execution speed of the program is also faster with manual garbage collection since the objects that can be deleted do not have to be searched.

To get those benefits sometimes significant work designing the memory management has to be spent. Mistakes are not excluded but tools like valgrind help a great deal verifying that the memory management is correct.

2.8.2.2 Data Structures Stored on the Heap

In the abstract machine there are five main data structures that need to be memory managed:

- the data term made up of subclasses of `Node`
- the result document, also made up of subclasses of `Node`
- the matrix
- the cache of submatrices
- the solution table

The first two are part of the data model, the latter three are part of the engine. Both parts are described in more detail in the following subsections.

2.8.2.3 Memory Management in the Data Model

The `Node` class of the data model, and thus all its derived classes, inherit from the `SharedData` class, the sole task of which is to do reference counting. When a node is added as a child to another node, the child node's refcount is increased by one and the parent node takes non-exclusive ownership.

When a parent node is deleted all of its children are `deref()`'ed. A node is only deleted when its reference count reaches 0. This way it is guaranteed that no nodes can be deleted that are children of some other, still existing, node. On the other hand, the top of the tree of nodes can be deleted even if some subtree is still used by another part of the program since the dereferencing makes sure that starting from the first still-in-use node nothing will be deleted.

This kind of reference counting also makes sure that nodes in data term graphs (and also shared matrices, see below) are not double-deleted. Graphs cause a node not to have one single unambiguous parent.

The abstract machine supports more than one query on the same data term. To prevent the rest of the abstract machine deleting parts of the data term, the root of the data term is `ref()`'ed in the main program.

2.8.2.4 Memory Management in the Engine

Just like the `Node` class, `Matrix` inherits `SharedData`, which takes care of the refcounting. A matrix is most likely a graph of matrices and thus reference counting is the only possibility to avoid double-deletions. Remember, one matrix can have two different parent matrices.

Since `Matrix` has a data term, it takes ownership of the data term by `ref()`ing it when the matrix is constructed and `deref()`s it when it is deleted. The matrix also `deref()`s (and deletes if needed) all its submatrices in the destructor.

The solution table is the reason for a little difference in the memory management of a matrix and a data term: All entries in the solution table are in essence `Matrix` objects. Those objects are only used to keep a reference of the variable binding to the data term, their submatrices are not needed anymore. Also, not all of the submatrices of the root matrix are used or needed by the solution table. Thus, it should be possible for the solution table to reference some matrices and delete all the others immediately, not only at the end of the abstract machine execution.

This is why the `Matrix` class provides a `derefRecursively()` method that not only dereferences and deletes its direct submatrices but walks all the way to the leaf matrices, dereferencing and deleting them all from the bottom up. Be careful though, this method should only be called exactly *once* for any matrix tree in conjunction with calling `ref()` on the root, since it works (and dereferences matrices) recursively.

The `Matrix::setFalse()`-Case

A special case is setting a matrix to `false`. Since a `false` matrix cannot have any submatrices, all submatrices are released and `deref()`'ed. In case no other object has a reference to one or more of those submatrices, they are deleted. However, it is not always possible to take definite ownership for a matrix although a reference to the matrix is needed.

An example is the cache used during the matrix population phase: The cache provides fast access to any submatrix in the matrix tree without having to walk through all the parent matrices starting from the root matrix. But the cache needs to be notified when a matrix is deleted so that it can discard it, otherwise the cache could return pointers to already free'd memory. The problem is that the caller of `Matrix::setFalse()` cannot know which submatrices will be deleted, if any.

So the `Matrix` class provides a way to register a function that is called back for all the matrices that are deleted by the `setFalse()` method and the `RootOperator`, which is a matrix population operator, registers the `removeFromCache()` function to keep the cache up-to-date with regards to the matrix tree.

In the Java version, this method is not strictly needed because as long as the cache refers to one of the matrices, this matrix is not deleted by the automatic garbage collection. The cache could be updated to provide for a faster execution though.

2.8.2.5 Freeing the Memory

`MatrixPopulationOperator::reset()` `deref()`s and may thus delete the matrix and it deletes the matrix cache.

`DocumentConstructionOperator::reset()` dereferences and deletes the result document.

`RelationalOperator::reset()` deletes the variable table. The contents of the table are untouched, the matrices are deleted by the `MatrixPopulationOperator`.

Instead of having to call those `reset()` methods, another design possibility would be to take the main data structures like the matrix, the cache, the document, etc. out of the main operator classes and put them into their own encapsulating class. This would centralize the memory management for them.

2.9 Benchmarks

This chapter shows some benchmarks of the implementations with some artificial data terms and query terms.

2.9.1 The Test Environment

All benchmarks were executed on a Toshiba Satellite Laptop with a Pentium IV CPU with 1800 MHz and 512 MB of memory. The operating system is a current Debian unstable distribution (August/September 2006) with Sun's Java 1.5.0, GCC 4.1.2, libstlport 5.0.2, and the Linux kernel version 2.6.17 with preemption enabled.

The data term used is an artificially created tree of nodes, it is a horizontal repetition of a core data term with a depth of 4 with the root defined to be at depth 1. The core data term is

```
f[ x, a[b[1], h, b], c, c[i, j, k], g[ "cool", sth[k, b] ] ]
```

The set of children of f is repeatedly added a number of times r, yielding a data term with $16 \cdot r$ nodes (descendants).

The data term is built specifically this way to ensure that there is no trivial and small solution possible.

There are two queries that are executed on this data term as part of the benchmarks. Query 1 is a full Xcerpt query that creates an output using the query results, Query 2 is a pure query without result document creation to stress-test the `CartesianProductOperator`.

- Query 1:

```
GOAL
   f[ all m[ var X, all var Y ] ]
FROM
```

```
  f[[ a[[ var X as b ]], var Y as c ]] ]] ]]
END.
```

This query is a full Xcerpt query rule and is run on the data term with a factor of $r = 185$, which means that the queried data term has 2 960 elements. The solution table has 68 820 entries after the execution of the query. The result document contains 172 976 nodes.

- Query 2: `f[[ c[[ var X ]], desc var Y ]]` This query does not construct any documents, it just produces the solution table. It is meant as a stress test for the `CartesianProductOperator`, since that is probably the most complex and resource hungry operator of all.

  This query is run on the data term with a factor of $r = 300$, which means the data term has 4 800 elements. The solution table has 2 157 300 entries after the execution of the query. There is no result document created.

  Also note that the translation of this query retains all 5 query variables in the solution table, in an optimized translation only 2 would be needed.

2.9.2 The Java Implementation

Before presenting the benchmark results, it should be noted that the memory usage figures are not those of the whole Java virtual machine, but the real memory usage of the running application. This is done by reducing the maximum Java heap size using the `-Xmx<size>` option. However, this does not take into account the Java stack size, but since most of the memory is allocated with `new` and thus needed on the heap, it may not be a significant omission at all.

To measure the runtime, the Java system call `System.currentTimeMillis()` is used.

2.9.2.1 Test Results

- Query 1: The query term of the rule is evaluated in 0.5 seconds, the document creation takes about 15 seconds with 10 MB of heap space.

- Query 2: The query term is evaluated in 16.98 seconds of which 16.3 seconds are needed to generate the solution table. The maximum memory usage is about 248 MB.

2.9.3 The C++ Implementation

For the C++ implementation, the brilliant valgrind tool serves as profiler and memory analyzer. Timing data is taken by the `clock()` system function.

2.9.3.1 Test Results

- Query 1: The query term of the rule is evaluated in 0.07 seconds of which 0.05 seconds are needed to generate the solution table, the document creation takes 2.21 seconds, and the maximum memory usage is a little over 7 MB.

- Query 2: The query term is evaluated in 2.88 seconds of which 2.81 seconds are needed to generate the solution table. The maximum memory usage is about 170 MB.

2.9.4 Comparison of the Java and the C++ Implementations

Unsurprisingly, it is obvious that the C++ abstract machine implementation is significantly faster while using a lot less memory.

A probable reason for the additional space of the Java version is the `Object` class overhead and the automatic garbage collection. The latter is primarily the case because the garbage collection does not clean up objects immediately after they went out of use, so they take up needless space.

An observable reason for a significant loss of speed is the garbage collector in case the memory is almost completely used up. The garbage collector starts to look for unused objects to discard, and this takes the more time the more objects there are. The more memory is made available, the faster the Java implementation becomes, again most probably because the garbage collection has not much to do anymore compared to the case of a very limited memory.

Those facts make completely representative and objective statistics difficult. When measuring the memory usage the heap size has to be decreased, when measuring the time, the heap memory has to be increased to yield better results in the respective case. Where does one set the limit and trade-off?

2.10 Future Research Possibilities

This chapter is just the beginning of the development of the complete abstract machine. One of the next important steps would be the integration of the parser. At the moment the parser is able to parse the whole Xcerpt language, however, building the abstract syntax tree as specified in the "Initial Draft of a Language Syntax", cf. [6] is only marginally implemented, the transformation of the abstract syntax tree into the abstract machine machine language is completely missing. But for testing purposes it would be nice to have the possibility of querying any XML document without having to write its in-memory representation by hand.

Another important area is the evaluation model: with this chapter, the proof of concept for the set-based evaluation model is done, but for the mixed model and, most important, for the lazy evaluation model it has yet to be shown.

2.11 Conclusion and Summary

This chapter constitutes a further development of the SUAM—the first attempt at an implementation of an abstract machine for the simulation unification, cf. [3]—by expanding the scope to single-rule Xcerpt programs and by integrating the extended matrix model of [5].

Three possible evaluation models are identified: the set-based evaluation model, the mixed evaluation model, and the lazy evaluation model, of which the set-based evaluation model is examined in-depth in this chapter.

For the set-based evaluation, a new set of rather high-level operators is defined. This set of operators is able to represent an Xcerpt single rule program and constitutes an abstract machine machine program.

A first version of the abstract machine has indeed been implemented in two well-known languages, namely Java and C++. Due to time constraints only a part of the original vision could be completed, which is the set-based evaluation model. The other two evaluation models have to be developed in the future.

The set-based evaluation turned out to exhibit the theoretically expected drawback also in practice. Because it computes all possible solutions in every step it needs an exponential amount of time and

space with regards to the size of the query and the data. Nevertheless, it was possible to improve this situation with a few optimizations, most notably the list tail sharing when storing the solutions in the relational solution table.

Finally, it was shown experimentally that the C++ implementation is superior to the Java implementation both in terms of memory and execution time, although the algorithms and data structures are almost identical. On top of that, the C++ language allows for further optimizations that emphasize the advantages of the C++ implementation even more.

Generally, it is great to see that the basic design of the abstract machine works and is efficiently implementable.

Acknowledgements.

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`).

Bibliography

[1] Publications of the Chair for Programming and Modelling Languages. Institute for Informatics, University of Munich, http://www.pms.ifi.lmu.de/publikationen/index.html, accessed 2006/09/28.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. S. (eds.). XQuery 1.0: An XML Query Language. Candidate Recommendation, W3C, June 2006. http://www.w3.org/TR/2006/CR-xquery-20060608/, accessed 2006/09/28.

[3] M. Brade. Towards an Abstract Machine for Xcerpt's Simulation Unification. Projektarbeit/project thesis, Institute for Informatics, University of Munich, 2005. http://www.pms.ifi.lmu.de/publikationen/#PA_Michael.Brade.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Y. (eds.). Extensible Markup Language (XML) 1.0 (Fourth Edition). Recommendation, W3C, Aug. 2006. http://www.w3.org/TR/xml, accessed 2006/09/28.

[5] F. Bry, T. Furche, B. Linse, and A. Schröder. Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In *Proc. of 8th International Workshop on Web Information and Data Management (WIDM 2006)*, Nov. 2006.

[6] F. Bry, T. Furche, and S. Schaffert. Initial Draft of a Language Syntax. Deliverable I4-D6, Institute for Informatics, University of Munich, 2006. http://idefix.pms.ifi.lmu.de:8080/rewerse/index.html#REWERSE-DEL-2006-I4-D6.

[7] F. Bry and S. Schaffert. Xcerpt—Rule-Based Querying and Reasoning on the (Semantic) Web. http://www.xcerpt.org/, accessed 2006/09/28.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM Journal of Computing*, 32(5):1338–1355, 2003.

[9] J. C. (ed.). XSL Transformations (XSLT). Recommendation, W3C, Nov. 1999. http://www.w3.org/TR/xslt, accessed 2006/09/28.

[10] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. B. (eds.). Document Object Model (DOM) Level 3 Core Specification. Recommendation, W3C, Apr. 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, accessed 2006/09/28.

[11] J. Hunter. JDOM. http://www.jdom.org, accessed 2006/09/28.

[12] L. MetaStuff. dom4j: The Flexible XML Framework for Java. http://www.dom4j.org, accessed 2006/09/28.

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

[14] S. Schaffert, F. Bry, and T. Furche. Simulation Unification. Deliverable I4-D5, Institute for Informatics, University of Munich, 2005. http://idefix.pms.ifi.lmu.de:8080/rewerse/index.html#REWERSE-DEL-2005-I4-D5.

[15] D. Sosnoski. XML and Java technologies: Document models, Part 1: Performance, Sept. 2001. http://www-128.ibm.com/developerworks/xml/library/x-injava/index.html, accessed 2006/09/28.