Reihe Informatik
2 / 1998

The Evaluation of Content-Based Web Queries

Thorsten Fiebig and Guido Moerkotte

The Evaluation of Content-Based Web Queries

Thorsten Fiebig *Guido Moerkotte*

Lehrstuhl für Informatik III

Fakultät für Mathematik und Informatik

Universität Mannheim

Seminargebäude A5

68131 Mannheim

Germany

email: *thorsten/moer*@pi3.informatik.uni-mannheim.de

web: http://pi3.informatik.uni-mannheim.de

fon: +49 621 292 5403

fax: +49 621 292 3394

Abstract

We introduce the notions of syntactically and semantically structured data to refine the notion of semi-structured data. As we will see, most data found on the Web is syntactically structured. In order to evaluate content-based Web queries, semantically structured data is needed. The problem occurs to transform syntactically structured data into semantically structured data.

Syntactically and semantically structured data can be represented by trees. Our main contribution is a powerful restructuring mechanism that allows to express the transformation of trees representing syntactically structured data to trees that represent semantically structured data.

We embed our restructuring mechanism into RAW (Relational Algebra for the Web) and demonstrate its expressiveness by several example queries.

1 Introduction

Advanced applications demand queries to be evaluated against a variety of data sources. These data sources can either be traditional database systems or non-traditional data sources containing semi- or unstructured data. Furthermore, both kinds data sources are often distributed world-wide.

Several approaches exist to integrate different kinds of data sources for querying [1, 2, 20, 16, 21, 16, 25, 12, 11, 8]. Among the predominant approaches are wrappers that wrap data sources in order to provide a uniform interface for the integrating query engine [10, 29, 16]. One necessary requirement is a common data model into which wrappers transform the data of their wrapped data sources. Many existing approaches build upon tree-based data models for this purpose [7, 31, 5, 14, 15, 31, 3], others build on object-oriented data models [1, 2, 8]. Another major problem is to describe the query capabilities of wrappers or data sources and exploit these for query processing [25, 26, 32]. However, we are not going to discuss this issue here. Instead, we dig deeper into the problems involved in processing semi-structured data representable as trees.

The actual differences between tree data models are minor. Some allow for labels only at the leaf nodes [31, 5], others allow labels at edges only [14, 15]; some support only unordered trees, others ordered trees [3]. The main advantage common to all tree models is that data in all standard data models (e.g. hierarchical, relational or object-oriented data model) as well as semi-structured data can be represented using trees [6, 15]. In fact, it seems that the term semi-structured is often used in exactly this case, that is, if a tree-based data model is used, the data represented with it is called semi-structured. Hence, the notion of semi-structured data is not very useful if data sources have to be classified. To see the point, consider two examples. A HTML-page can be represented by its parse tree. The nodes (or edges) might be labeled by the syntactic categories (e.g. list item). These labels differ from the semantic labels like “restaurant” found in work on tree-based data models [6, 15]. Obviously, the former carries less semantics than the latter although there mostly exists a correspondence between the syntactic representation and the semantic interpretation. However, this correspondence is—without any further information—not easily accessible to computers. Hence, we distinguish between syntactically and semantically structured data (trees). Figures 1 and 2 represent a syntactically and a semantically structured data tree. The example is taken from the Ley server [27]. Since most data on the Web is syntactically structured, content-based Web queries—i.e. queries stated using semantic categories such as author name or publication date—demand the transformation of syntactically structured data to semantically structured data. However, note that todays query languages for the Web do not provide such a mechanism [28, 24, 23]. Instead, they rely on predicates based on string matching and syntactic structure. Other approaches directly work on semantically structured data only [1, 2].

The problem focussed on in this paper is to introduce a means to express the missing information needed to transform syntactically structured data into semantically structured data—as required for content-based Web queries. As a first step we introduce RAW data trees which are able to represent both kinds of data. Our working hypothesis is that tree restructuring mechanisms are capable of transforming syntactically structured trees into semantically structured data trees. Hence, we introduce a powerful restructuring mechanism. From RAW data trees representing semantically structured data, subtrees relevant to process a query must often be extracted. We present the extraction mechanism used within RAW. It supports the 10 different kinds of tree pattern matching discussed in [30]. Complex example queries against the Ley server illustrate how the restructuring and extraction mechanisms are embedded into RAW expressions that express content-based queries.

The rest of the paper is organized as follows. Section 2 reviews RAW. Its definition differs slightly from the original proposal [17]. The main contribution of the paper can be found in Section 3. The new domain RAW data tree—our variant of a tree-based data model— is introduced. Besides different constructors it provides for a powerful restructuring mechanism allowing to transform syntactically structured data trees into semantically structured data trees. Relevant parts of a data tree can be extracted using different tree matching algorithms producing variable (attribute) bindings. The restructuring mechanism and the extraction mechanism are both described in Section 3. In Section 4, we give several examples of complex queries against Ley's server. They demonstrate how the restructuring and extraction mechanisms are embedded into the algebra. Section 5 concludes the paper.

2 RAW

One major design goal of RAW[1] is to bridge the gap between web query languages and traditional query optimization since we believe that there is much unexploited optimization potential in the currently prevailing interpreter approach. As a consequence, RAW must be able to cope with syntactically structured data found on the biggest data source there is—the Web. Instead of designing a totally new approach, we decided to enhance the traditional relational algebra by as few new algebraic operators as possible and mainly by new domains. Each new domain type comes along with certain functions and, hence, can be thought of as an ADT. This will open a road between web queries and standard database query evaluation and optimization techniques.

2.1 RAW's Relational Model and Domains

A relational schema $\mathcal{R}$ is a finite set of attribute names $\{A_1, \dots, A_n\}$. With each attribute A_i a domain D_i is associated. A Relation R for a schema $\mathcal{R} = \{A_1, \dots, A_n\}$ is a subset $R \subseteq D_1 \times \dots \times D_n$. This is standard [4]. The only new algebraic operator in RAW is the *MAP* operator. This operator is rather standard in object-oriented systems where it occurs under a variety of names [22, 13].

Apart from the standard domains (int, float, string, bool) RAW features the following domains:

1. URL

2. Web-Path

3. HTML-Document

4. RAW Data Tree (RDT)

[1] see [17] for a full account of the design goals

A URL denotes an universal resource identifier (URI) as defined in [18].

A Web-Path is used to represent a path within the Web. It can be thought of as an alternating sequence of URLs and documents—except that instead of the actual documents special identifiers are used. For the purpose of the paper, the following two functions defined for Web-Paths suffice:

- *length*: Web-Path → int

  returns the number of URLs contained in a Web-Path.

- *get_last*: Web-Path → URL

  returns the last URL contained in the Web-Path

Web-Paths can be generated by applying the function

get_paths: URL, pred, patt → set-of(Web-Path)

Starting at the argument URL, it traverses the whole net to generate all possible paths starting at this URL. Since often not all paths are required, there exist two possibilities to restrict the number of actually generated paths. The first is a unary predicate that is evaluated on every generated path, only if the predicate is fulfilled, the path is extended and put into the result set. The second possibility to restrict the number of results is a pattern. This pattern describes those parts of the starting document where *get_paths* is allowed to look for URL's. Patterns are described in more detail in the next section.

2.2 The Algebraic Operators

As the common Relational Algebra RAW consists of a set of operations. The operator set contains the standard set operators union, intersection, set-difference. Further, it contains the standard relational algebra operators. Let us denote the type of relation with schema $\{A_1, \ldots, A_n\}$ by $\{[A_1 : \tau_1, \ldots, A_n : \tau_n]\}$. Each Attribute of the relation is represented by its name A_i and its domain type τ_i.

- SELECT_P: $\{[A_1 : \tau_i, \ldots, A_n : \tau_n]\} \rightarrow \{[A_1 : \tau_1, \ldots, A_n : \tau_n]\}$,

  where $P : [A_1 : \tau_1, \ldots, A_n : \tau_n] \rightarrow$ *bool* is the selection predicate

- $\mathit{PROJECT}_{\{A_i:\tau_i,\ldots,A_j:\tau_j\}}$:$\{[A_1 : \tau_1, \ldots, A_n : \tau_n]\} \rightarrow \{[A_i : \tau_i, \ldots, A_j : \tau_j]\}$,

  where $\{A_i : \tau_i, \ldots, A_j : \tau_j\} \subseteq \{A_1 : \tau_1, \ldots, A_n : \tau_n\}$

- JOIN_P:$\{[A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n}]\}, \{[B_1 : \tau_{B_1}, \ldots B_n : \tau_{B_n}]\} \rightarrow \{[A_n : \tau_{A_1}, \ldots, A_n : \tau_{A_n}, B_1 : \tau_{B_1}, \ldots, B_n : \tau_{B_n}]\}$,

  where $P : [A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n}], [B_1 : \tau_{B_1}, \ldots B_n : \tau B_n] \rightarrow$ *bool* is the join predicate

As selection and join predicates, RAW accepts arbitrary boolean expressions possibly contain functions for the new data types introduced above. Additionally the *MAP* operator belongs to the operator set of RAW.

$$\mathit{MAP}_{f,B}{:}\{[A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n}]\} \rightarrow \{[A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n}, B : \tau]\}$$

$$f{:}[A_1 : \tau_{A_1}, \ldots, A_n : \tau_{A_n}] \rightarrow \tau$$

For each input tuple the map operator extends the input relation by the attribute B, that is the result of the evaluation of the function f found in the subscript. Possibly the result of function f is a set, in this case the operator generates one output tuple for each element in the set or in case of multiple set valued expressions, one output tuple for each combination. In the latter case the *MAP* operator can be thought of as a standard *MAP* operator followed by an unnest.

3 From Syntactically to Semantically Structured Data

In order to facilitate content-based Web queries, more domains are needed to represent syntactically and semantically structured data. This sections introduces the RAW data tree (or RDT for short). The RDT is the new core domain used to represent syntactically and semantically structured data. RAW data trees will be presented in the next subsection. A typical RAW query works in several phases. First, the information is fetched from the net and presented as a RAW data tree. This

RAW data tree is typically syntactically structured. The conversion from syntactically structured data to semantically structured data is achieved by a powerful restructuring mechanism represented in subsection 3.2. Then the relevant parts of the semantically structured data tree are extracted and mostly represented as atomic attribute values. This part is introduced in subsection 3.3. Last, additional selection predicates are evaluated and the result is projected. Note that this is only a common pattern found in RAW expressions. Queries do not have to follow this pattern. Typical queries following this pattern are presented in Section 4.

3.1 RAW Data Trees and their Construction

A RAW data tree is a labeled ordered tree. Every node consists of a label and a unique identifier. Additionally, inner nodes have an ordered list of successor nodes. A node is termed leaf node if this list is empty. Example RAW data trees can be found in Figures 1 and 2. (We do not give the note identifiers since they are not relevant for our discussion here.)

For a given URL, the referenced document is transformed into a RAW data tree in two steps. First, the URL is dereferenced and the HTML-Document is parsed. As in traditional compilers, this results in an abstract syntax tree (AST) [9]. The domain HTML-Doc consists of the set of all possible abstract syntax trees possible for HTML-Documents. Second, the abstract syntax tree is transformed into a RAW data tree.

Why this indirection? If in the near future HTML will be replaced by XML then we only have to exchange the parser for HTML in order to enable RAW to deal with XML documents. Hence, the necessary changes to the RAW system remain small.

The functions supporting the transformation are:

- *fetch*: URL → HTML-Doc
- *build_tree*: HTML-Doc → RDT

The *fetch* function retrieves a HTML-Document referenced by a given URL from the Web and parses it. The result is its abstract syntax tree in the domain HTML-Doc. Within the AST representation, two kinds of nodes exist. Those representing HTML-Tags together with their attributes and plain text nodes representing the displayed text or contents of the tags. Consider for example the A-Tag. It consists of attributes (e.g. the href attribute) and its contents (the anchor text). The implementation of *fetch* is rather standard. After fetching the document via the hypertext transfer protocol [18], standard parsing takes place [9].

The function *build_tree* transforms the abstract syntax tree into the RAW data tree representation. For every AST node representing a HTML-Tag, a small subtree consisting of three nodes is generated. The root of the subtree is a node labeled by the HTML-Tag. Its successor nodes are the attribute node and the content node. Every AST node representing plain text is mapped to a single RAW data tree node representing this text. All nodes representing plain text are collected under a newly constructed nodes labeled by "Text". Again, the implementation is rather straight forward. For both functions, the Cocktail Tools [19] were used for the implementation.

3.2 Restructuring RAW Data Trees

The application of the *build_tree* function typically results in a syntactically structured data tree. The crucial and most difficult step is to transform this syntactically structured information into semantically structured information. Let us illuminate the occurring problems by an example situation. Ley's Logic Programming and Database server provides plenty of HTML-pages representing bibliographical data [27]. Among the different views given upon this data is the author page. For a given author, all publications of this author are presented on a single page. Let our task be to add this information for a given set of authors to a relational database containing bibliographical data. Obviously, doing so manually is a tedious task. Using RAW this task becomes less laborious.

Using the mechanisms explained so far, it is not difficult to retrieve the page for a given author and produce the according syntactically structured data tree. For every publication of the given author it contains a subtree representing this publication. Unfortunately, the syntactic structure

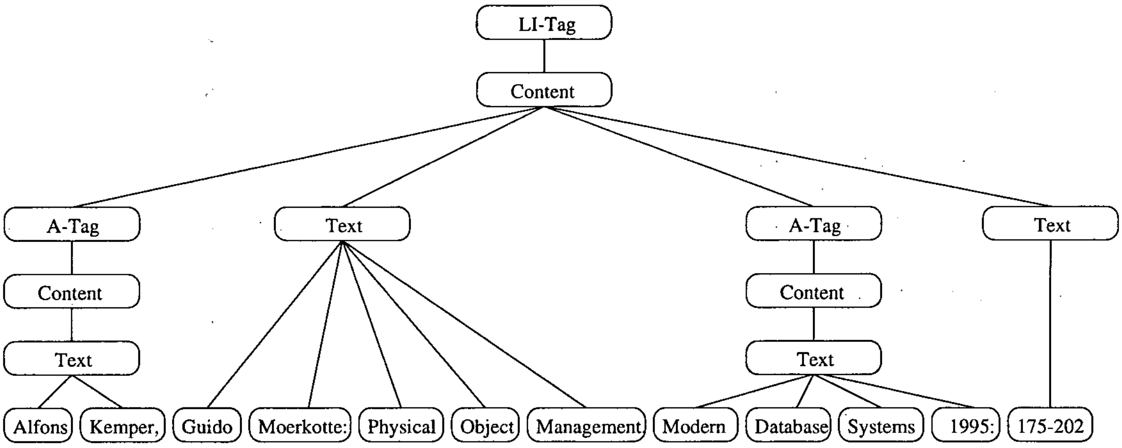

Figure 1: A Syntactically Structured Data Tree

does not match the semantic structure we would expect for a bibliographic reference. Figure 1 shows one subtree for a publication by the authors Kemper and Moerkotte as found on the page for "Moerkotte". As a default, while parsing a HTML-page, longer strings are cut into pieces whenever a blank or newline character occurs. For every coauthor, except the one to whom the page belongs, there is a reference to the coauthor's publication page. This (justified) irregularity is directly reflected within the data tree resulting from the abstract syntax tree for the entries: the tree in Figure 1 has a plain text node for the author and the title since this part is no further structured by HTML-Tags. The resulting problem is that a single node represents information on two semantic categories: coauthor and paper title. Hence, this node needs to be refined by splitting. The opposite can also occur: one semantic concept is represented by different nodes in the syntactically structured data tree. Take the whole list of authors in Figure 1 as an example.

A step towards a semantically structured data tree for the same entry can be found in Figure 2. The two main differences are that (1) there exists a single subtree for every semantic category and (2) the according root nodes are labeled by the category names. It should be clear that this tree could be gained from the one in Figure 1 by restructuring. It is our working hypothesis that

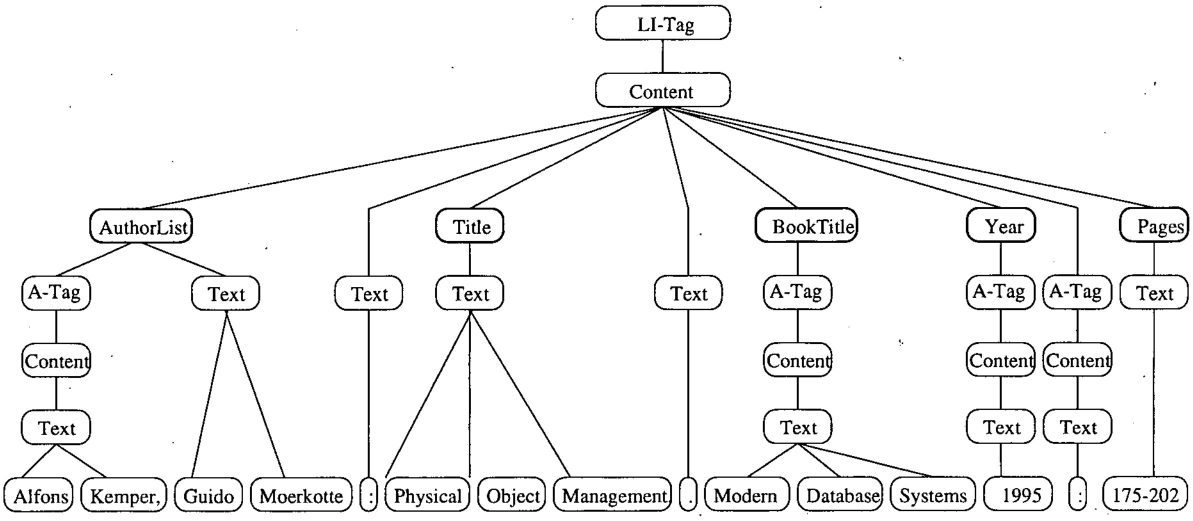

Figure 2: A Semantically Structured Data Tree

the transformation of syntactically structured data trees to semantically structured data trees can be achieved by a sequence of restructuring steps. Each step thereby enriches/restructures a given RAW data tree. In RAW, a single restructuring step corresponds to an application of the function *tree_restructure* which takes a RAW data tree and a restructuring specification as arguments. Restructuring specifications bear some resemblance to traditional context free grammars.

A restructuring specification consists of a set of non-terminal symbols NT and a set of rules. As building blocks rules contain non-terminal symbols and regular expressions to denote tokens. The regular expressions have to be enclosed by quotation marks. Further, two types of rules exist. The first type consists of rules with no non-terminal symbol on the left-hand side (Type-1). These rules are merely used to describe patterns to be found in the data tree. The second type of rules exhibits a non-terminal symbol on the left-hand side (Type-2). These rules additionally introduce new nodes labeled by their left-hand side's non-terminals into the data tree.

Here is part of the restructuring specification needed for author pages of the Ley server:

1. $NT = \{$ *AuthorList, Title, BookTitle, Pages, Year* $\}$

2. *:= AuthorList “:” Title “.” BookTitle Year “:” Pages;*

3. *AuthorList :=“[¬:]+”;*

4. *Title := “[¬.]+”;*

5. *BookTitle := “[¬.]+”;*

6. *Pages := “[0-9]+-[0-9]+”;*

7. *Year := “[0-9]+”;*

The right-hand sides of the rules consist of a sequence of non-terminal symbols and regular expressions. Within the regular expressions, we preceded a character c by ¬ in order to denote any character but c. As usual, “+” denotes non-empty repetition. The complete right-hand sides of a rule constitute a pattern to be matched against the given data tree. During the matching process, the actual boundaries of the plain text fragments in leaf nodes are crossed if necessary. For example, the first rule specifies that the author list and the title must be separated by a colon. This colon is not present as a singular node in the data tree, instead it is contained in the node labeled by “Moerkotte:”. Further, the complete author list is contained in four data tree nodes. Though the rules are easy to specify, the matching process is rather complex. After a succesful match, new nodes are created for those rules whose left-hand side is non-empty. The label of the node directly corresponds to the left-hand side's non-terminal.

More specifically, restructuring takes place in three steps. Step 1 expands Type-1 rules by replacing the non-terminals by the according rules. This requires the rule set to be non-recursive. If there exist several rules with the same left-hand side, all combinations are generated. Step 2 tries to match the expanded rules against the data tree. Thereby, the total sequence of leaf nodes is considered as a single string. The goal of the matching step is to find fragments of the data tree that correspond to the regular expressions of the expanded rules. A fragment consists of a sequence of leaf nodes corresponding to exactly one regular expression in the expanded rule. During this

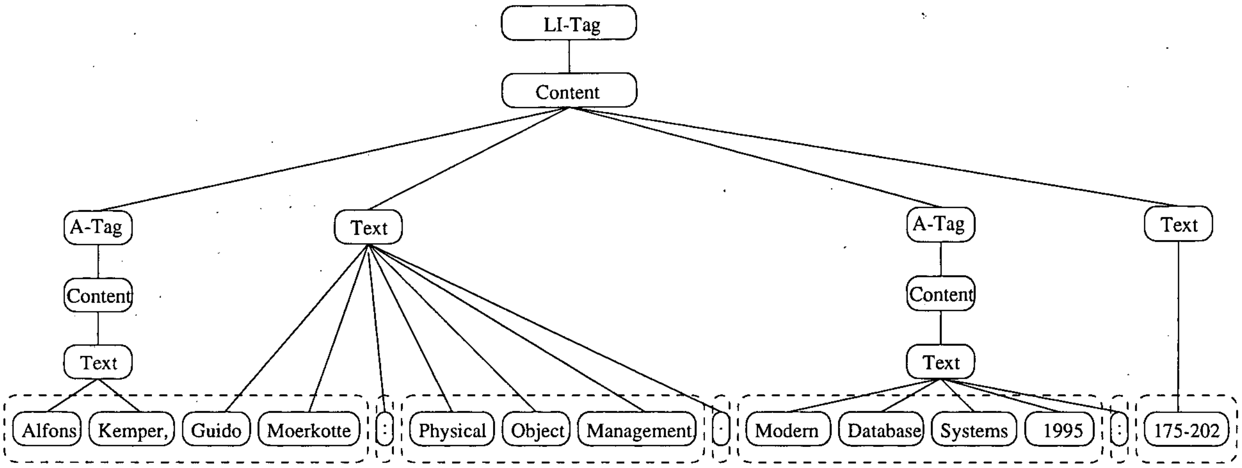

Figure 3: Intermediate State I

step leaf nodes are splitted if several regular expressions share the label of a node. The shared label is then distributed among the nodes resulting from the split. Inner nodes are splitted, if they point to more than one fragment. For splitted inner nodes, the label is duplicated. This step takes place iteratively until no more inner node points to more than one fragment. Step 3 creates new nodes for the matching rules with non-empty left-hand side. Their label is the non-terminal on the left-hand side of these rules.

Let us illustrate the three steps using the example. During the first step, the first rule is expanded to

:= “[¬:]+” “:” “[¬.]+” “.” “[¬.]+” “[0-9]+” “:” “[0-9]+-[0-9]+”;

The second step matches this expanded rule against the given data tree. The first two regular expressions (“[¬:]+” and “:”) match the four nodes labeled “Alfons”, “Kemper,”, “Guido”, and Moerkotte:”. Hence, since “Moerkotte:” contains parts for two regular expressions, it must be split into a node labeled “Moerkotte” and a node labeled “:”. Likewise, the node labeled “1995:” must be split. Figure 3 shows the tree after splitting the leaf nodes. Fragments corresponding to a regular expression in the expanded rule are framed by dotted lines. Note that the first node and

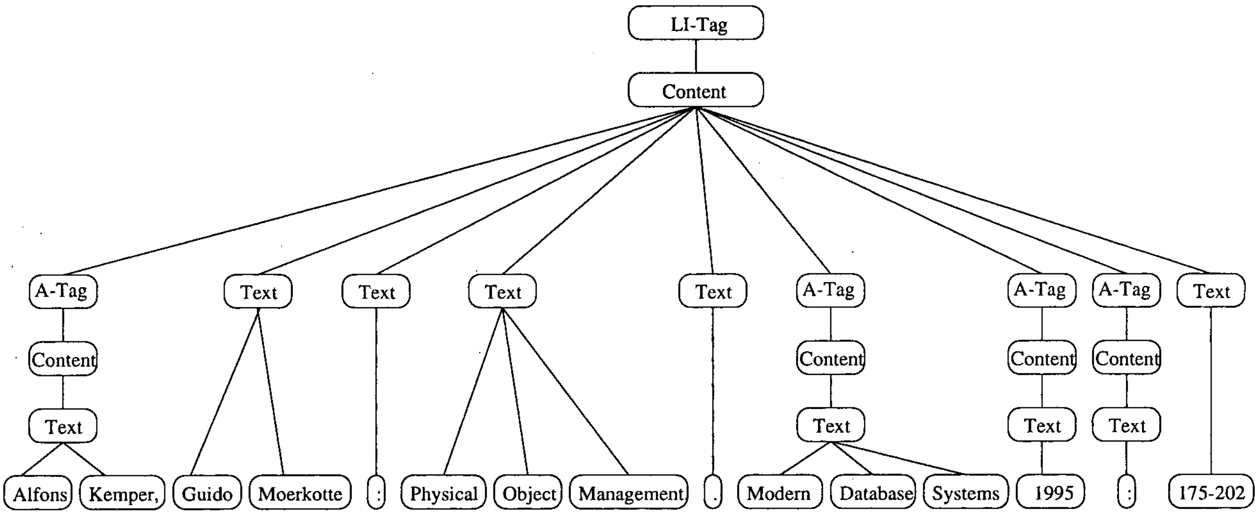

Figure 4: Intermediate State II

the second node labeled "Text" point two more than one fragment. Hence, they must be split. The result of splitting/duplicating the inner nodes is shown in Figure 4. In the third step, new nodes with labels "AuthorList", "Title", "BookTitle", "Year", and "Pages" have to be introduced (see the last 5 rules of the restructuring specification). The least upper node whose subtree points to all fragments comprising the body of the rule is labeled "Content". Hence, the new nodes are introduced directly below this node. Figure 2 shows the result after introducing the new nodes.

3.3 Extraction of Raw Data Subtrees

Often the interesting information is placed only in a small part of the data tree. In order to extract the relevant parts of a data tree, the function

- *tree_match*: RDT, RPT → VariableBinding (= Tuple)

is introduced. It takes a RAW data tree and a RAW pattern tree as arguments. A RAW pattern tree (RPT) is an adorned RAW data tree. Nodes in RAW pattern trees can be labeled by variables. Additional adornments allow to express different kinds of tree matchings (see below).

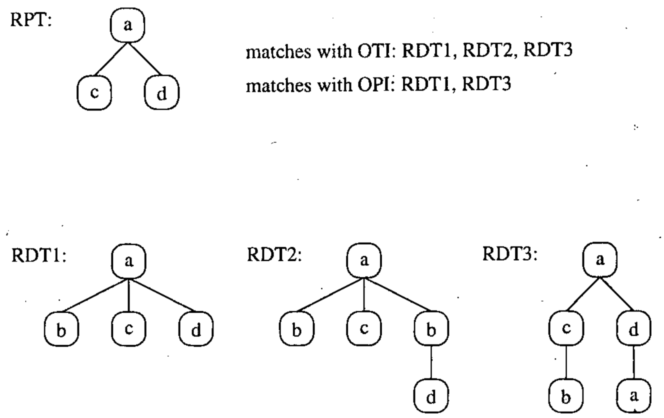
RPT:
a
c
d
matches with OTI: RDT1, RDT2, RDT3
matches with OPI: RDT1, RDT3
RDT1:
a
b
c
d
RDT2:
a
b
c
b
d
RDT3:
a
c
d
b
a

Figure 5: Different Kinds of Tree Matching

The function *tree_match* matches the RPT against the given data tree. A *match* consists of a correspondence relation between the nodes of the data tree and the nodes of the pattern tree. For every successful match a variable binding is produced for every variable found in the pattern tree. More specifically, if a pattern node n_p is labeled by a variable v and for a successful match n_d is the corresponding node in the data tree, the variable v is bound to the subtree rooted at n_d. For every variable binding a tuple is constructed where the attribute names equal the variable names and the attribute's value are the bindings of the corresponding variables.

With adornments, RAW pattern trees are able to express different kinds of matching. These different kinds of matching express variations in the degree of correspondence that has to be achieved during the matching process. For example, if the order of the nodes in the pattern tree might be relevant or irrelevant for the matching process. Another distinction is whether the subtree of the data tree must be isomorphic to the pattern tree or whether it must contain the pattern tree. Let us explain the difference between ordered tree inclusion and ordered path inclusion. For ordered tree inclusion, every node of the pattern tree must match with some node in the data tree but a child node of the pattern tree does not have to match with a child of the matching node of its parent pattern node. Instead the matching successor node can be several levels down. Consider for example the pattern RPT and the three RAW data trees RDT1 to RDT3 in Figure 5. Under ordered tree inclusion, the pattern tree RPT matches with all three data trees even with the second one. Obviously, the root node of the pattern tree matches with the root node of the data tree RDT2. The left child of the pattern tree (labeled by "c") matches the middle child of RDT2's root. The right child (labeled "d"), however, does not match any direct child of RDT2's root. Instead, it matches the grandchild labeled "d". For tree inclusion matching this is a regular situation. Hence, tree inclusion can be used to traverse unknown paths down the tree. In situations where this implicit traversal of paths is not allowed, path inclusion matching is applied. Here, the children of a pattern tree must match the children of the corresponding node in the data tree. Other possible restrictions exist. For example, one might want the match to be complete, that is, every node in the data tree must also be present in the pattern tree. In total, ten variants of tree matching are distinguished [30]:

- Unordered Tree Inclusion (UTI)
- Ordered Tree Inclusion (OTI),
- Unordered Path Inclusion (UPI),
- Ordered Path Inclusion (OPI),
- Unordered Region Inclusion (URI),
- Orderd Region Inclusion(ORI),
- Unordered Child Inclusion (UCI),
- Ordered Child Inclusion (OCI),
- Unordered Subtree Problem (USP), and
- Ordered Subtree Problem (OSP).

With according adornments, RAW pattern trees are able to express which kind of matching is demanded at a certain node. The necessary matching algorithms have been implemented according the lines of [30]. However, some modifications in order to retrieve the variable bindings and to enhance performance had to be made. But these algorithms as well as a detailed description of the different matching variants is well beyond the scope of the paper. For the purpose of the paper, it suffices that relevant parts of a data tree can be extracted using RAW pattern trees together with the function *tree_match*.

4 Example Queries

This section presents three queries against the Ley server [27]. Every query will first be stated in natural language. Then we give their algebraic equivalent. We are aware of the fact that these algebraic expressions exhibit optimization potential but optimizing them is beyond the scope of this paper.

Query 1 Query 1 retrieves the names of all authors that have published at least a SIGMOD or a VLDB paper. Figure 6 shows the RAW expression for this query. We explain the expression from bottom to top. The lowest line—representing the input relation—contains a single tuple with a single attribute. This attribute points to the author index of the Ley server. Here, authors are put into small alphabetical groups. For example, the second group contains authors from *Abr* to *Ade*. Every group is represented as a hyperlink referencing the list of according authors. These are hyperlinks pointing to the actual author pages. On this page, all the publications of the author are collected.

The second line of the RAW expression retrieves all the links on the start page and dereferences them two times. The step is implemented by applying the *get_paths* function via the *MAP* operator. This function returns a set of Web-paths. Since it is set-valued, the *MAP* operator unnests this set. For every element in the set, an output tuple is constructed. These output tuples differ from

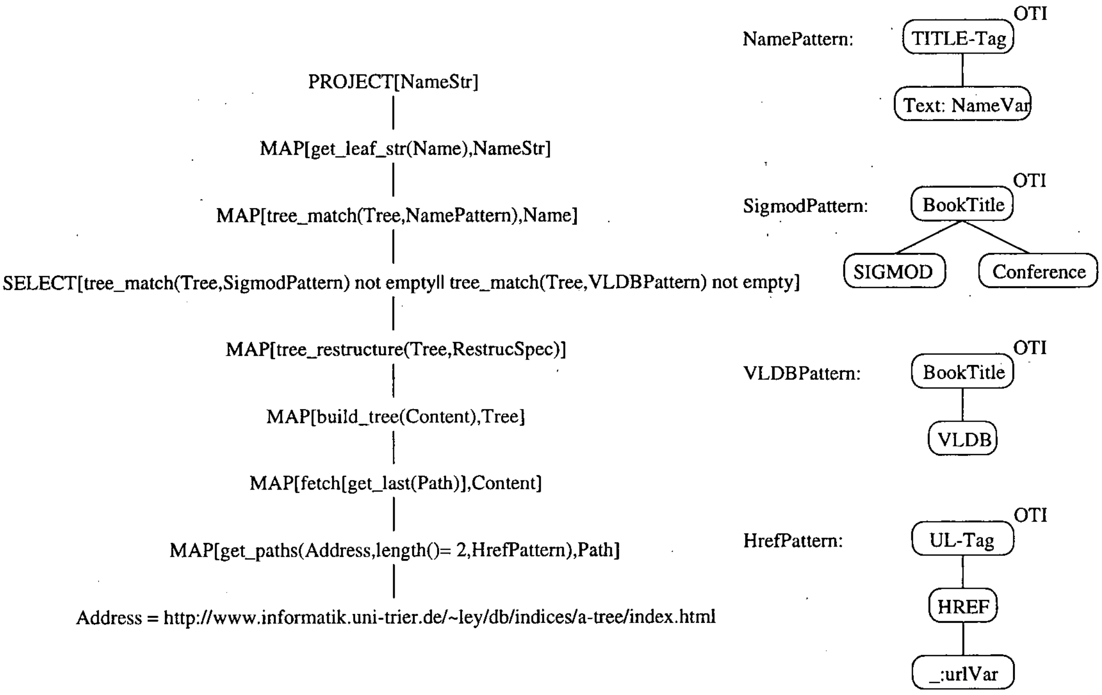
PROJECT[NameStr]
MAP[get_leaf_str(Name),NameStr]
MAP[tree_match(Tree,NamePattern),Name]
SELECT[tree_match(Tree,SigmodPattern) not empty|| tree_match(Tree,VLDBPattern) not empty]
MAP[tree_restructure(Tree,RestrucSpec)]
MAP[build_tree(Content),Tree]
MAP[fetch[get_last(Path)],Content]
MAP[get_paths(Address,length()= 2,HrefPattern),Path]
Address = http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html
NamePattern:
OTI
TITLE-Tag
Text: NameVar
SigmodPattern:
OTI
BookTitle
SIGMOD
Conference
VLDBPattern:
OTI
BookTitle
VLDB
HrefPattern:
OTI
UL-Tag
HREF
_:urlVar

Figure 6: The RAW expression for Query 1

the input tuple in that a single attribute *Path* is added that holds one of the paths contained in the result set of *get_paths*.

The next two *MAP* operators fetch the documents and build the data trees. Then the restructuring specification is applied to transform the syntactically structured author pages into semantically structured author pages as discussed in the previous section. The *SELECT* operator uses the tree matching algorithms to verify that the paper is a SIGMOD or a VLDB paper. The according patterns are given on the right-hand side of the figure. Then, the name pattern is applied to retrieve the subtree of the data tree that represents the author's name. The function *get_leaf_str* then collects the tree labels and builds a string by concatenating them. Last, the name is projected.

Query 2 This query retrieves the years when "Carey" published a SIGMOD paper. The RAW expression can be found in Figure 7. The bottom four lines are identical with those for Query 1.

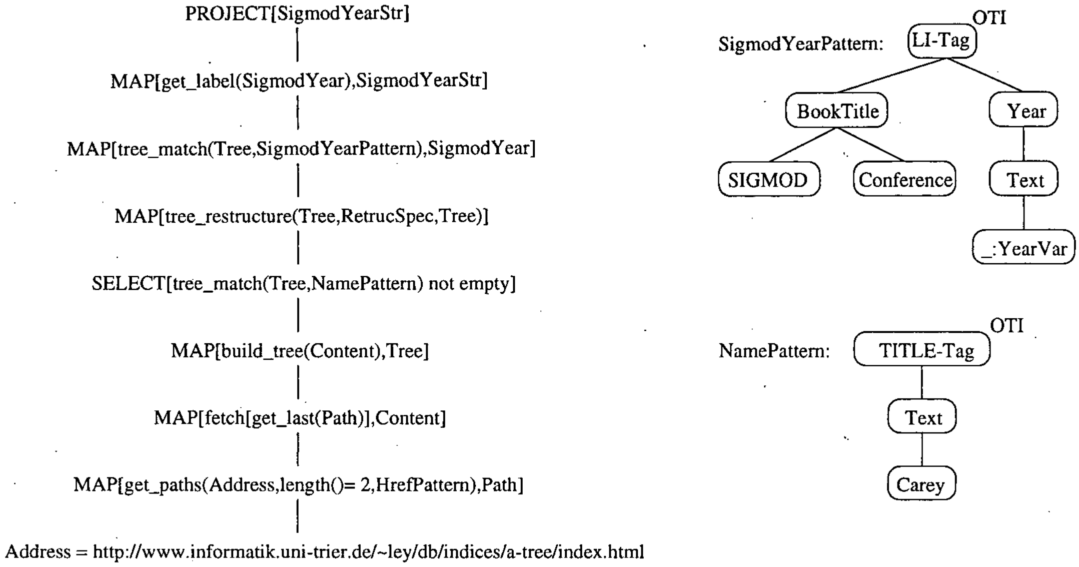
PROJECT[SigmodYearStr]
MAP[get_label(SigmodYear),SigmodYearStr]
MAP[tree_match(Tree,SigmodYearPattern),SigmodYear]
MAP[tree_restructure(Tree,RetrucSpec,Tree)]
SELECT[tree_match(Tree,NamePattern) not empty]
MAP[build_tree(Content),Tree]
MAP[fetch[get_last(Path)],Content]
MAP[get_paths(Address,length()= 2,HrefPattern),Path]
Address = http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html
SigmodYearPattern:
OTI
LI-Tag
BookTitle
Year
SIGMOD
Conference
Text
_:YearVar
NamePattern:
OTI
TITLE-Tag
Text
Carey

Figure 7: The RAW expression for Query 2

After their evaluation we have an attribute named *Tree* containing the semantically structured data tree for author pages. We then select only those pages where the pattern *NamePattern* matches. This represents the restriction to "Carey"'s page. The last three operators retrieve the years of SIGMOD publications.

Query 3 Query three is used to retrieve all the bibliographic information. This is not a typical user query. Instead, the purpose of such a query could be to fill a local database with information found on the Web. Query 3 retrieves the author list, title, booktitle, year, and pages for all references found. The bottom five lines of its algebraic representation (see Figure 8) correspond to those of Query 1. They provide an attribute *Tree* containing the semantically structured author pages. Then the pattern "BibPattern" is applied to look for the relevant bibliographical information within this page. The next five *MAP*s retrieve the bibliographical information from the subtrees resulting from the previous step.

All these queries refer to semantic categories like publication years, authors, titles, etc. Hence,

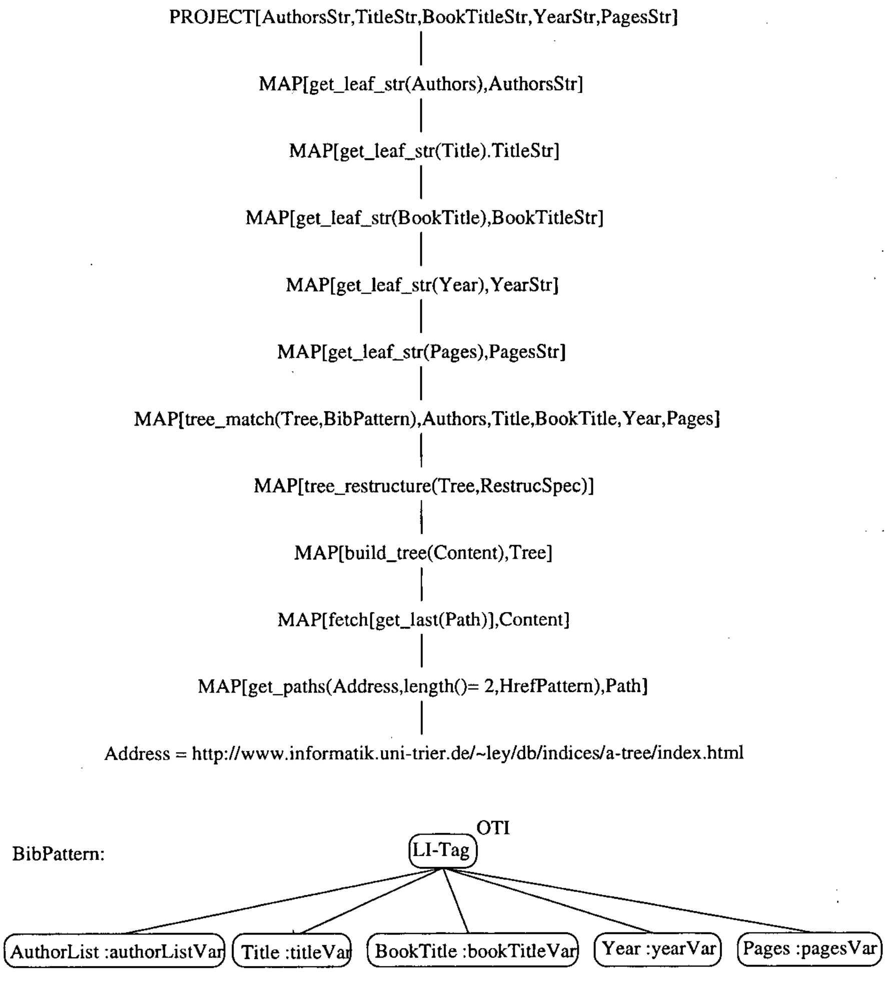
PROJECT[AuthorsStr,TitleStr,BookTitleStr,YearStr,PagesStr]
MAP[get_leaf_str(Authors),AuthorsStr]
MAP[get_leaf_str(Title).TitleStr]
MAP[get_leaf_str(BookTitle),BookTitleStr]
MAP[get_leaf_str(Year),YearStr]
MAP[get_leaf_str(Pages),PagesStr]
MAP[tree_match(Tree,BibPattern),Authors,Title,BookTitle,Year,Pages]
MAP[tree_restructure(Tree,RestrucSpec)]
MAP[build_tree(Content),Tree]
MAP[fetch[get_last(Path)],Content]
MAP[get_paths(Address,length()= 2,HrefPattern),Path]
Address = http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html
BibPattern:
OTI
LI-Tag
AuthorList :authorListVar
Title :titleVar
BookTitle :bookTitleVar
Year :yearVar
Pages :pagesVar

Figure 8: The RAW expression for Query 3

they are content-based and cannot easily be answered using other Web query languages [28, 24, 23]. Referring to these semantic categories is only possible due to the restructuring process. However, this step does not come for free: the restructuring specification must be provided. But although the restructuring mechanism is quite powerful, writing a restructuring specification is a rather easy task.

5 Conclusion

We introduced the distinction between syntactically and semantically structured data trees and argued that most of the data found on the Web is syntactically structured. In order to evaluate content-based queries a transformation of syntactically to semantically structured data becomes a necessity. A powerful restructuring mechanism achieving this task has been introduced. After restructuring, often relevant parts of the data tree must be extracted. For this purpose we developed a pattern-based extraction mechanism. Several queries demonstrated the potential of this approach.

References

[1] S. Abiteboul, S. Cluet, and T. Milo. Querying and Updating the File. In *Proceedings of the 19. VLDB Conference*, 1993.

[2] S. Abiteboul, S. Cluet, and T. Milo. A Database Interface for Files Update. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[3] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Translation for Heterogeneous Data. In *Proceedings of the 6. International Conference on Database Theory*, Delphi, Greece, 1997.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[5] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), November 1996.

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Digital Libraries*, 1(1), 1997.

[7] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, 1997.

[8] O. A.Bukhres and A. K. Elmagarmid, editors. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications.* Prentice Hall, 1996.

[9] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[10] N. Ashish and G. Knoblock. Wrapper Generation for Semi-structured Internet Sources. In *Proceedings of the Workshop on Management of Semistructured Data*, 1997.

[11] P. Atzeni and G. Mecca. Cut and Paste. In *Proceedings of the Symposium on Principle of Database Systems*, pages 144–153, Tucson, Arizona, 1997.

[12] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proceedings of the 23. VLDB Conference*, 1997.

[13] J. Blakeley, W. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[14] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, Electronic Workshops in Computing. Springer, 1995.

[15] P. Bunemann, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[16] M. Carey. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proc. IEEE RIDE-DOM*, 1995.

[17] T. Fiebig, J. Weiss, and G. Moerkotte. RAW: A Relational Algebra for the Web. In *Proceedings of the Workshop on Management of Semi-structured Data*, pages 34–41, 1997.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, January 1997. http://www.ics.uci.edu/pub/ietf/http.

[19] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. Technical Report 20, Dr. Josef Grosch, Datenverarbeitung Karlsruhe, 1990.

[20] R. H. Güting, R. Zicari, and D. M. Choy. An Algebra for Structured Office Documents. *ACM Transactions on Office Information Systems*, 8(4):121–157, April 1989.

[21] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, 1997.

[22] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the 16. VLDB Conference*, 1990.

[23] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-Wide Web. In *Proceedings of the 21. VLDB Conference*, 1995.

[24] L. V. S. Lakshmanan, F. Sadri, and T. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. of the 6th. International Workshop on Research Issues in Data Engineering, RIDE '96*, February 1996.

[25] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources using Source Descriptions. In *Proceedings of the 22. VLDB Conference*, 1996.

[26] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries using Views. In *Proceedings of the Symposium on Principle of Database Systems*, 1995.

[27] Michael Ley. Databases & Logic Programming. http://www.informatik.uni-trier.de/ ley/db/index.html.

[28] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. PDIS '96*, December 1996.

[29] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Schema for Rapid Implementation of Wrappers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1997.

[30] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, 1992.

[31] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying Semistructured and Heterogeneous Information. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD)*, 1995.

[32] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering Queries using Templates with Binding Patterns. In *Proceedings of the Symposium on Principle of Database Systems*, 1995.