A Logical View of Structured Files*

Serge Abiteboul
I.N.R.I.A., Rocquencourt
France
(Serge.Abiteboul@inria.fr)

Sophie Cluet
I.N.R.I.A., Rocquencourt
France
(Sophie.Cluet@inria.fr)

Tova Milo
Tel-Aviv U.
Israel
(milo@math.tau.ac.il)

Tel: (33) 1 39 63 56 79 Fax: (33) 1 39 63 53 30

March 16, 1997

Abstract

Structured data stored in files can benefit from standard database technology. In particular, we show here how such data can be queried and updated using declarative database languages. We introduce the notion of *structuring schema* which consists of a grammar annotated with database programs. Based on a structuring schema, a file can be viewed as a database structure, queried and updated as such.

For *queries*, we show that almost standard database optimization techniques can be used to answer queries without having to construct the entire database. For *updates*, we study in depth the propagation to the file of an update specified on the database view of this file. The problem is infeasible in general and we present a number of negative results. The positive results consist of techniques that allow to propagate updates efficiently under some reasonable *locality* conditions on the structuring schemas.

Keywords: Database, Textual data, File System, Query, Update, Query and Update Optimization.

1 Introduction

Database systems are primarily concerned with structured data. However, structured data is still often stored in an unstructured manner (in files) even when it does have a strong internal structure (e.g., electronic documents or programs). Unfortunately, data stored in files cannot benefit from database technology and in particular, has to rely on very primitive linguistic support for queries and updates. In this paper, we show how *structured files* can be queried and updated using high-level database languages.

In this paper, we are primarily concerned with data organized according to so-called *data exchange formats* (DXF) (e.g., SGML, HTML, ASN.1, STEP/EXPRESS, etc.) and stored in file systems. A long term goal is the migration of such data to systems providing database-style "physical-level" functionalities such as recovery, versioning, concurrency control, etc. At the logical level, the way we use

*Part of this work was done while the third author visited INRIA and U. of Toronto. The work was partially supported by European Communauty Projects Fide2 and GoodStep, by the Institute for Robotics and Intelligent Systems in U of Toronto, and by the Israel Academy of Sciences and Humanities. Some preliminary versions of part of this work can be found in [2, 3]

such data is not satisfactory either. Files with structured data are typically accessed using editors (e.g. emacs), programming language (e.g., perl, the acronym for Practical Extraction and Report Language), or graphic interfaces in the style of Mosaic. In the best cases, these tools know of the inner structure of data, e.g., SGML editors. However, in terms of querying facility, they rarely provide more than pattern matching. We believe that a high-level database language is a central component for accessing data organized according to DXF's.

To continue with the motivations, an important advantage of offering high level language access to files in DXF's is that it provides a uniform interface to these files. The users and application designers are relieved from mundane tasks such as learning the syntax of specific formats or using application specific interfaces for particular DXF's.

We therefore consider the problem of querying/updating files that have some grammatical structure. More precisely, we are concerned with *structured files* (to be defined formally further on) which form our main tool for describing the implicit structure of files and defining an abstract interface to their information. A *structuring schema* consists of a grammar annotated with database programs and of a database schema. A structuring schema specifies how the data contained in a structured file should be interpreted as a database.

Based on a structuring schema, one can parse the file and load data into a database. It is also possible to understand a structuring schema as a specification of a *database view* of the file. In this alternative approach, it is not necessary to load the entire file in order to answer a query. Indeed, variations of standard optimization techniques from relational databases can be used to avoid any unnecessary data construction. The development of these optimization techniques is a major contribution of this paper.

Structuring schemas provide a uniform way of accessing data stored in many files possibly based on different DXF's: use high-level database languages such as SQL or OQL on the database view of the file. Similarly, structuring schemas provide a uniform way of updating such files: updates are specified on the database view of the file. First, a *naive* approach to the implementation of these updates is considered: (i) the database view of the file is materialized; (ii) the update is performed on the database; and (iii) the database is "unparsed" to produce an updated file. For this, we develop an *unparsing* technique. The problems that we meet while developing this technique are related to the well-known view update problem. (See, for instance [14, 15, 18, 31].) The technique relies on the existence of an inverse mapping from the database to the file.

The *naive* technique presents major drawbacks. It is inefficient: it entails intense data construction and unparsing, most of which in general dealing with data not involved in the update. Moreover, it may entail unnecessary loss of information from the file that is not recorded in the database. A major contribution of this paper is a combination of techniques that allows to minimize both the work (the data construction and the unparsing) and the information loss. We show how query optimization techniques can be used to focus on the relevant portion of the database and avoid constructing the entire database. We also show that for a class of structuring schemas satisfying a *locality* condition, it is possible to carefully circumscribe the unparsing thereby minimizing the loss of information.

Some of the results in the paper are negative. They should not come as a surprise since (i) we are dealing with complex theoretical foundations (language theory for parsing and unparsing, and first-order logic for database languages) and (ii) our problem is related to two very complex issues in databases, namely data restructuring and view update. However, we do present positive results for particular classes of structuring schemas. We believe that the restrictions imposed on these schemas are very acceptable in practical situations.

For the grammar, we assume standard knowledge of context-free languages and in particular of the parsing of such languages. (In rare cases, we do brief digressions to SGML grammars since they were the prime motivation of this work and provide some interesting features.) For databases, we use here a complex value model (or similarly the nested relation model) [1]. We sometimes do brief excursions to object-based models. Then the particular model we have in mind is O_2 [16]. More important than the specific data model, the target query language is OQL [9] both for the complex value model or the object-oriented model.

The problems of extracting data from files and encoding data onto files (to a lesser extent) have been popular database topics since the early days of the field. One should notably cite the Express system in the 70's for data extraction and restructuring [26]. Two languages were proposed there: *define* to specify the structure of the file content and *convert*, an algebraic language for restructuring the data. Our contribution obeys the same motivations since the problem clearly did not disappear over the years and is even reflourishing with the popularity of DXF's and the explosion of the use of Internet. We use now standard database technology tools (e.g., optimization techniques, object-orientation) to provide a modern answer to the old problem. Motivations and various proposals can be found in [12, 25, 10, 22, 17, 5, 20, 7, 13].

The present work is part of an important implementation effort at INRIA in the general direction of data integration. Data is imported from relational and object-oriented DBMS, and from various DXF's. Data restructuring is based on the work presented here (structuring files with database query/update facilities) and on the O_2-View system[27] developed in our group. The integrated data can be queried/updated using an extension of OQL with information retrieval and "navigation" facilities described in [10] and using standard hypertext tools such as Mosaic/Netscape and full-text indexing facilities.

The paper is organized as follows. In the next section, we define structuring schemas, discuss this notion and introduce the problems that we study. Structuring schemas specify the loading of file data into the database. The inverse mapping (the unloading from the database to a file) is briefly considered in Appendix B. Section 3 deals with the query optimization technique; and Section 4 with updates. In Section 5, we present an optimization technique for updates. Section 5 is based on some technical tools developed in Appendix A. The last section is a conclusion.

2 Structuring Schemas

In this section, we first introduce informally the notion of structuring schema. Then we formalize it, and finally present the problems studied in the paper.

Informal presentation

The problem is to obtain a database representation of a string with some inner structure. This inner structure is described in a natural way using a grammar. A first approach (that we will later abandon) is based strictly on the grammar. This requires the utilization of an object-based data model. The connection with the database is obtained in a straightforward manner by linking each vertex of the parse-tree to a database object. In other words, the parse-tree of the string is stored in the database. We illustrate this approach next, and explain why it is unsatisfactory.

Example 2.1 Bibliography files are examples of popular structured data. Consider for example a bibliography file in the BibTex format[19]. An entry in the file is a string of the form:

$$
\begin{array}{lll}
@Inproceedings & \{ & TLE90, \\
 & & author = \text{``}A.A.\ Toto\ and\ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{''}, \\
 & & title = \text{``}On\ Weird\ Dependencies\text{''}, \\
 & & booktitle = stoc, \\
 & & year = \text{``}1990\text{''}, \\
 & & pages = \text{``}1-12\text{''} \\
 & & notes = \text{``}A\ later\ version\ of\ this\ paper\ appears\ in\ JIR98\text{''}\}.
\end{array}
$$

Consider the following partial grammatical description of a BibTex file:

$$
\begin{array}{lcl}
\langle Ref\_Set \rangle & \rightarrow & \langle Reference \rangle\ \langle Ref\_Set \rangle \\
 & | & \epsilon \\
\langle Reference \rangle & \rightarrow & \text{``}@Inproceedings\{\text{''}\ \#String \\
 & & \text{``}author = \text{''}\ \langle Author\_Set \rangle \\
 & & \text{``}title = \text{''}\ \#String \\
 & & \ldots \text{``}\}\text{''}
\end{array}
$$

Nonterminals appear between brackets (e.g., $\langle Ref\_Set \rangle$). The grammar considers two kinds of lexical tokens: constant tokens between double quotes (e.g., "@Inproceedings {") and other tokens that are prefixed by the # symbol (e.g., $\#String$) and for which the lexical analyzer returns some typed values (e.g., string or integer).

Using default structuring, the grammar could correspond to the following class definitions (with standard methods such as display or edit attached to these classes):

$$
\begin{array}{lcl}
class\ Ref\_Set & = & tuple(reference : Reference, references : Ref\_Set) \\
class\ Reference & = & tuple(key : String, author : Author\_Set, title : String, \ldots)
\end{array}
$$

Attributes names were added for readability although, strictly speaking, such names are not mentioned in the grammar. A list of two references such as

$$
\begin{array}{lll}
@Inproceedings & \{ & TLE90, \\
 & & author = \text{``}A.A.\ Toto\ and\ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{''}, \\
 & & title = \text{``}On\ Weird\ Dependencies\text{''}, \\
 & & booktitle = stoc, \\
 & & year = \text{``}1990\text{''}, \\
 & & pages = \text{``}1-12\text{''} \\
 & & notes = \text{``}A\ later\ version\ of\ this\ paper\ appears\ in\ JIR98\text{''}\}. \\
@Inproceedings & \{ & L91, \\
 & & author = \text{``}H.R.\ Lulu\text{''}, \\
 & & title = \text{``}More\ on\ Weird\ Dependencies\text{''}, \\
 & & booktitle = focs, \\
 & & year = \text{``}1991\text{''}, \\
 & & pages = \text{``}120-124\text{''}\}.
\end{array}
$$

is then represented by an object o, where the association between objects and values is partially described below:

class name	*oid*	*value*
class Ref_Set :	o	$[reference : o_1, references : o_2]$
	o_2	$[reference : o_3, references : o_4]$
	o_4	$\perp$
class Reference	o_1	$[key : \text{“}TLE90\text{”}, author : o_5, \ldots]$
	o_3	$[key : \text{“}TLE90\text{”}, author : o_6, \ldots]$

where object o_4 with undefined value ($\perp$) represents the empty string (i.e., an empty list of references).

Observe that with this approach, *disjunctive types* naturally arise from non terminals defined by several production rules. Clearly, the use of disjunctive types can be avoided using complex coding but then the database is a much less direct representation of the parse-tree. (Similarly, one could use also a data model without objects but again at the cost of widening the gap between the parse-tree and its database representation.)

The default structuring leads to severe problems when querying the resulting database structure. One can consider two major origins for these problems:

1. *Presence of unnecessary information in the string/grammar.*
   Everything from the file has to be explicitly represented in the database. In particular, the database will record parsing information with perhaps little informative content that will in fact hide the "natural" structure of the data. Also, this approach often results in the presence of many unnecessary objects. E.g., in the above example, the natural structure of the list is hidden and o_2 and o_4 have no meaningful semantics.

2. *Absence of semantic information in the string/grammar.*
   For instance, the string above contains a list of references. This may be interpreted as a database list, but also as a set or a bag. Clearly, the intended meaning is application dependent and the grammar gives no indication to choose between these possible representations.

This approach based solely on the grammar clearly fails. This leads to consider a second approach requiring to explicitly state the links between grammatical constructs and their database counterparts. This is done again in a rather standard manner using a schema definition and an annotated grammar. We call the resulting couple a *structuring schema*. This solution is somewhat reminiscent of techniques used for automatic synthesis of editors for languages. They also use annotated grammars to construct editors and "unparsing" specifications to display programs. Of course, the problems studied are very different but some of the techniques they develop (e.g., incremental evaluation) are applicable to our context. (See [21].)

A *structuring schema* consists of a schema and an annotated grammar. We first illustrate the idea informally using an example. A formal definition is presented next. The annotated grammar specifies the relationship between the grammar non terminals and their database representation. More precisely, it associates to each derivation rule $A \rightarrow A_1, \ldots, A_n$ a statement describing how the database representation of a word derived from this rule is constructed using the database representations of the sub-words derived from $A_1, \ldots, A_n$.

The next example provides a possible (partial) structuring schema for a BibTex file.

Example 2.2 (BibTex continued) The schema is described first:

`/* Non terminals type definition */`
type $\langle Reference\rangle = \textbf{tuple}(Key : string, Authors : ...)$
type $\langle Ref\_Set\rangle = \textbf{set}(\langle Reference\rangle)$

The annotated grammar is given by:

`/* Annotated grammar */`

$$\begin{array}{lcll}
\langle Ref_Set\rangle & \rightarrow & \langle Reference\rangle\ \langle Ref_Set\rangle & \{\$\$:= \{\$1\} \cup \$2\} \\
 & | & \epsilon & \{\$\$:= \{\}\} \\
\langle Reference\rangle & \rightarrow & \text{“}@Inproceedings\{\text{”}\ \#String & \\
 & & \text{“}author = \text{”}\ \langle Author_Set\rangle & \\
 & & \ldots \text{“}\}\text{”} & \{\$\$:= [Key : \$1, Authors : \$2 \ldots]\}
\end{array}$$

Observe that a Yacc-like notation is used. In the example, the $$ symbol in the action part of a rule represents the data returned by this rule. A $i symbol represents the data returned by the ith nonterminal or non-constant token in the right hand-side of the rule. The non-constant tokens have values corresponding to database basic types that are returned by the lexical analysis. The whole BibTex file is represented by a set containing one element per bibliographical reference. A reference is represented by a tuple with one attribute per BibTex field.

The schema describes the database types. One may argue that the type information can be partially or totally derived from the annotated grammar using type inference. However, the issue of type inference can be viewed in a larger context and is not the topic of the present paper.

To see a more complex examples, suppose that we want the database to record both the set of references and the set of authors. The two “anchors” can be viewed as the fields of a unique tuple describing the entire database. This is specified in the following structuring schema:

`/* Non terminals type definition */`
type $\langle BibTex\rangle = \textbf{tuple}(Refs : \textbf{set}(\langle Reference\rangle), Authors : \textbf{set}(\langle Author\_Set\rangle))$
...

```
/* Annotated Grammar */
```

$$
\begin{array}{lcl}
\langle BibTex \rangle & \rightarrow & \langle Reference \rangle \; \langle BibTex \rangle \\
& & \qquad \{\$\$:= [Refs : \$2.Refs \cup \{\$1\}, \\
& & \qquad\qquad Authors : \$2.Authors \cup \$1.Authors]\} \\
& | & \epsilon \\
& & \qquad \{\$\$:= [Refs : \{\}, Authors := \{\}]\} \\
\langle Reference \rangle & \rightarrow & ... \\
& & \text{“}author = \text{”} \; \langle Author\_Set \rangle \\
& & \ldots \\
& & \qquad \{\$\$:= [\ldots, Authors : \$2, \ldots]\} \\
\langle Author\_Set \rangle & \rightarrow & \langle Author \rangle \text{“,”} \langle Author\_Set \rangle \\
& & \qquad \{\$\$:= \{\$1\} \cup \$2\} \\
& | & \langle Author \rangle \\
& & \qquad \{\$\$:= \{\$1\}\} \\
\langle Author \rangle & \rightarrow & \#String \\
& & \qquad \{\$\$:= \$1\}
\end{array}
$$

Observe the action for the $\langle BibTex \rangle$ nonterminal: the tuple returned by parser $\langle BibTex \rangle$ of the body of the rule is projected to its components and the values of these components are used to construct the new tuple.

We next define structuring schemas formally.

Formal definition

A *structuring schema* (*SS* in short) consists of a context-free grammar annotated with one semantic action per rule, and one type per nonterminal. We assume that the grammar is reduced. More precisely, for each grammar G that we consider and each rule in G, we assume that there is a derivation of some word in $L(G)$ that uses this rule. We now describe the types and actions that we consider in this paper.

The database types are defined by the following abstract syntax:

$$
\begin{array}{lll}
base\_type & :- & int \mid real \mid bool \mid string \\
\tau & :- & base\_type \mid [A_1 : \tau_1, ...A_n : \tau_n] \quad \% \text{ tuple type} \\
& & \mid \{\tau\} \mid \langle \tau \rangle \mid \{\!\{\tau\}\!\} \qquad\qquad \% \text{ set/list/bag type}
\end{array}
$$

A value of type *int*, *real*, *bool* or *string* is an element of the corresponding sorts. We also assume that *int*, *real*, *bool* or *string* can occur as tokens in the grammars, and that the lexical analyzer then returns a value of the appropriate sort. A value of type $\{\tau\}$ is a finite set of values of type τ, and similarly for bags, lists and tuples. To simplify, we assume that empty collections are explicitly typed (e.g., an empty strings set is different from an empty integers set). Structuring schemas were first introduced in [2] with a more general data model. To simplify, we mostly ignore here some aspects considered in the original framework: union of types, objects/classes/inheritance. We informally consider them in sections devoted to extensions.

We next define the semantic actions that we consider in this paper. Actions are defined using functions over database types. In the following definition, some of the functions are described explicitly (e.g. set/tuple constructors). Others are only referenced as belonging to a predefined set F of typed functions. The reason for distinguishing between these two classes of functions will become clear in the sequel. At this point, we may assume that F includes addition, subtraction, division, multiplication over int or real; and the entire complex value algebra [1].

Definition 2.3 The *actions* are terms formed as follows:

1. each $\$i$ is an action, each database constant is an action.

2. under obvious type restrictions, if $a_1, ..., a_n$ are actions

   - $\{a_1, ..., a_n\}$ is an action (set construction), and similarly for lists and bags;
   - $[A_1 : a_1, ..., A_n : a_n]$ is an action (tuple construction);
   - $cons(a_1, a_2)$ is an action; it adds an element a_1 to the (set, list or bag) collection a_2;
   - $a_1 \cup a_2$ is an action (union of sets/bags, concatenation of lists);
   - $a_1 || a_2$ is an action (string concatenation);
   - $f(a_1, ..., a_n)$ for $f \in F$ is an action.

In the paper, we mostly prove results for $F = \emptyset$. Sometimes, we consider problems occurring from introducing some operations (such as projection or join) in F. When considering extensions such as objects, we will have to enrich the action language.

A structuring schema defines a mapping, denoted *parse*, from the set of strings accepted by the grammar to the set of databases of appropriate type. (This assumes that the parsing always terminates which is reasonable since the language is context-free.)

Note that there may be several parse-trees for the same file (due to ambiguities in the grammar). From a practical viewpoint, one can assume that *parse* tries the rules in some predefined order and so, that *parse* is indeed a mapping. From a theoretical view point, it is difficult to investigate properties of structuring schema taking into account the order of rules, since even very simple properties of *parse* then become undecidable[1].

We therefore concentrate on a relation that ignores the order of rules. Given a SS S, a nonterminal T, a file f and a value v, $f \leadsto_{S,T} v$, indicates that there exists a parse-tree of word f rooted at T and *yielding* the database value v.

We consider context-free grammars. Intuitively, the terminal symbols corresponds to the constant tokens (e.g., "author =") and non-constant tokens (e.g., $\#String$). The nonterminals correspond to the various parsers. To define formally, the relation $\leadsto$, we need the auxiliary function *build_string*. Intuitively, for a rule r and an assignment of strings to the nonterminals and the non-constant tokens of the r, *build_string* returns a string accepted by r. More precisely, let r be a rule with tokens $P_1, ..., P_m$ in that order, nonterminals and non-constant tokens $T_1, ..., T_n$ in that order, and values $f_1, ..., f_n$ for $T_1, ..., T_n$, $build\_string(r, [f_1, ..., f_n])$ is the string $f = w_1 || ... || w_m$ where for each j:

1. either P_j is a constant token w_j;

2. or P_j is the non-constant token or a nonterminal T_i and w_j is f_i.

[1]For example, testing if there is a successful parsing of a word (taking into account the order of rules) that uses a given rule r is undecidable. This can be proved by reduction from the undecidability of testing containment of context-free languages. Let G_1 and G_2 be two grammars with no common nonterminal and start symbols S_1, S_2. Let S be a new nonterminal and G the grammar with start symbol S and the following rules: The rules of G_1 and G_2 and the rules $r_1 : S \rightarrow S_1$, $r_2 : S \rightarrow S_2$ with precedence for r_1. Then, $L(G_2) \subseteq L(G_1)$ iff *parse* never uses the rule r_2.

The relation $f \rightsquigarrow_{S,T} v$ is defined inductively as follows. First for each τ in $\{int, real, bool, string\}$, and each data value v of type τ, $f \rightsquigarrow_{S,\tau} v$ if f is the string[2] denoting the data value v of type τ. Then, $f \rightsquigarrow_{S,T} v$ if S has a rule r with head T, a body containing the nonterminals and non-constant tokens $T_1, ..., T_n$ in that order, and and action t. And for some v_i, f_i $(1 \leq i \leq n)$:

1. for each i, $f_i \rightsquigarrow_{S,T_i} v_i$.
2. $f = build\_string(r, [f_1, ..., f_n])$;
3. v is $t[\$i \rightarrow v_i]$, i.e., is the value obtained by substituting each $\$i$ by the corresponding value and evaluating the resulting algebraic expression;

When S is understood, $f \rightsquigarrow_{S,T} v$ is simply written $f \rightsquigarrow_T v$. For T_0 the start symbol, $f \rightsquigarrow_{S,T_0} v$ is sometimes written $f \rightsquigarrow_S v$. When both S and the start symbol are understood, we sometimes use $f \rightsquigarrow v$ and the value v is called a *database view* of file f.

Note that one file may have several database views (due to ambiguity of the underlying grammar). Also, several different files may have the same database view. This happens when the database image of the file does not record all the information stored in the file (i.e. the mapping involves information loss). We shall consider these issues in depth is Section 4.

Database interface to files: the issues

A database view of a file provides a convenient interface to the data stored in the file. Consider the BibTex example. Suppose that we want to find the title of a paper of keyword TLE90. This can be formulated as a database query q:

$$\begin{array}{ll} \textbf{select} & r.Title \\ \textbf{from} & r \textbf{ in } References \\ \textbf{where} & r.Key = \text{“}TLE90\text{”}. \end{array}$$

where *References* denotes the set returned by the parsing.

For a more complex example, suppose that we want to check for each paper, whether there exists a more recent version, and add to the outdated reference a note pointing to the newer version. This can be easily formulated using a database update language and the join-like facilities built in such languages:

$$\begin{array}{ll} \textbf{update} & r1.Notes := concat(r1.Notes, \text{“}A\ later\ version\ appears\ in\ \text{”}, r2.Key) \\ \textbf{from} & r1 \textbf{ in } References,\ r2 \textbf{ in } References \\ \textbf{where} & r1.Title = r2.Title \textbf{ and } r1.Authors = r2.Authors \textbf{ and } r1.Year < r2.Year \end{array}$$

Note that we use here some rather trivial criteria to detect earlier versions. (In an OODB context, we could use an arbitrary method as predicate.) Even for such simple ones, the reader familiar with sophisticated editing tools may consider how this would be specified in his/her favorite one. E.g., how would you code this in an emacs macro?

We can use the SS to parse the file and load the result into the database. In the BibTex example, the database will then contain a set of tuples, one per reference. We may also use the SS to define a virtual "view" of the file. In this case, the set is not loaded into the database. In both cases, we would

[2] We assume that each atomic data value is represented by a single string. Another way to view this is that we do not distinguish between two files differing only in their representation of atomic values.

like to be able to query and update the references efficiently. If the data is fully loaded into a database, then the query optimization is done at the database level. On the other hand, when querying a view, we would like to avoid loading the entire file into the database, and need specific optimization techniques to achieve that. This aspect of optimization is the topic of Section 3. The issue of propagating updates specified at the database level to updates on the file, and the optimization of this process is the topic of Section 4.

3 Query Optimization

Assume that we use a SS to define a view of a file. A naive technique to evaluate a query on such a view is to materialize the view (i.e., parse the file, load all the data into the database), and then evaluate the query. This naive technique has a major drawback: It entails construction of the entire database, even if the the query actually uses only a small portion of it. This is time and space consuming.

The main contribution of this section is an optimization technique to avoid constructing the entire database to answer a query. We first focus on value-based databases, and consider the issue of object-based databases separately towards the end of the section. The technique is mainly based on modifying the *action* part of the SS, (although it may also lead to modifications of the grammar), and is based on standard query optimization techniques.

We first give some intuition through a very simple example and present the optimization algorithm. We then discuss some issues and limitations. Finally, we extend the technique to object-based data models.

3.1 The Core Technique

Consider a file containing BibTex references with the SS above. Suppose that we want to evaluate query q. The corresponding algebraic query is $\varphi(References)$ where:

$$\varphi \equiv \lambda X \cdot \Pi_{Title}(\sigma_{Key=\text{``}TLE90\text{''}}(X)).$$

Observe that we need to return a string and the construction of the entire database by the naive evaluation strategy seems really unnecessary. To reduce the data construction, we "push the selection down" inside the SS in the way selections are pushed down in relational algebraic query expressions by optimizers [31]. We leave the grammatical part of the structuring schema practically unchanged (the parser must still recognize the same file) whereas we modify the actions in an appropriate manner.

First, consider the rule used to compute the set of references:

$$\begin{array}{lcll} \langle Ref\_Set\rangle & \rightarrow & \langle Reference\rangle\ \langle Ref\_Set\rangle & \{\$\$:= \{\$1\} \cup \$2\} \\ & | & \epsilon & \{\$\$:= \{\}\}. \end{array}$$

The desired result can be obtained using the rule:

$$\textbf{(1)} \quad \begin{array}{lcll} \langle \varphi(Ref\_Set)\rangle & \rightarrow & \langle Reference\rangle\ \langle Ref\_Set\rangle & \{\$\$:= \varphi(\{\$1\} \cup \$2)\} \\ & | & \epsilon & \{\$\$:= \varphi(\{\})\} \end{array}$$

where $\langle\varphi(Ref\_Set)\rangle$ is a new nonterminal, indeed, the new start symbol.

A strict application of this SS would result in creating the entire set of references and then applying the query to it. But, we can now apply some query rewriting to the action of rule 1. Observe first that

$$\varphi(\{\}) = \Pi_{Title}(\sigma_{Key=\text{``}TLE90\text{''}}(\{\})) \quad \leadsto \quad \{\}.$$

And in a less standard way, we have

$$\varphi(\{\$1\} \cup \$2) \rightsquigarrow \varphi(\{\$1\}) \cup \varphi(\$2) \rightsquigarrow$$
$$\Pi_{Title}(\sigma_{Key=\text{``}TLE90\text{''}}(\{\$1\})) \cup \varphi(\$2)$$
$$\rightsquigarrow$$
$$\{\Pi_{Title}(\sigma_{Key=\text{``}TLE90\text{''}}(\$1))\} \cup \varphi(\$2).$$

Part of this is rather standard, e.g., distributivity of selection/projection w.r.t. union. Part is less so, e.g., pushing projection/selection onto a single element since \$1 is not a set but a tuple. For this, we extend the algebra to have such operations also operate on singleton elements. We obtain the new rewrite rules by viewing such an element as a singleton set. In particular, the selection on a single element is defined as follows: if it succeeds, the result is the element itself; and if it fails, the result is the single element $\perp$ (which is viewed algebraically as the empty set).

For instance, one rewrite rule that we used is:

$$\sigma_{Cond}(\{A\})) \rightsquigarrow \{\sigma_{Cond}(A)\}$$

which can now be accepted also if A is an atomic value. The standard set of rewrite rules [6, 11, 23, 28] is extended in this manner.

The result of the rewriting of action of (1) leads to:

$$\textbf{(2)} \quad \begin{array}{lcll} \langle \varphi(Ref\_Set) \rangle & \rightarrow & \langle Reference \rangle \ \langle Ref\_Set \rangle & \{\$\$:= \{\varphi(\$1)\} \cup \varphi(\$2)\} \\ & | & \epsilon & \{\$\$:= \{\}\} \end{array}$$

Continuing with the example, we next "push" the query "down into the grammar". The query on \$1 and on \$2 in (3) is pushed down to the corresponding nonterminals. More precisely, instead of calling $\langle Ref\_Set \rangle$ and filtering its result with φ, we use the new parser $\langle \varphi(Ref\_Set) \rangle$ that also has the responsibility of applying φ. And similarly for $\langle Reference \rangle$. Thus, (2) is transformed into:

$$\textbf{(3)} \quad \begin{array}{lcll} \langle \varphi(Ref\_Set) \rangle & \rightarrow & \langle \varphi(Reference) \rangle \ \langle \varphi(Ref\_Set) \rangle & \{\$\$:= \{\$1\} \cup \$2\} \\ & | & \epsilon & \{\$\$:= \{\}\} \end{array}$$

We already have a definition for the nonterminal $\langle \varphi(Ref\_Set) \rangle$. We obtain a definition for $\langle \varphi(Reference) \rangle$ by transforming the rules for $\langle Reference \rangle$ in the following way:

$$\textbf{(4)} \quad \begin{array}{lcl} \langle \varphi(Reference) \rangle & \rightarrow & \text{``}@Inproceedings\{\text{''} \ \#String \\ & & \text{``}author = \text{''} \ \langle Author\_list \rangle \\ & & \text{``}title = \text{''} \ \langle \#String \rangle \\ & & \ldots \text{``}\}\text{''} \{\$\$:= \varphi([Key : \$1, Authors : \$2, Title : \$3 \ldots])\} \end{array}$$

Observe that here again we apply φ to a single element, i.e., a reference. We can use again a rewriting technique:

$$\textbf{(6)} \quad \begin{array}{c} \Pi_{Title}(\sigma_{Key=\text{``}TLE90\text{''}}([Key : \$1, Authors : \$2, Title : \$3 \ldots])) \\ \rightsquigarrow \\ \Pi_{Title}([Key : \sigma_{\lambda \cdot x(x=\text{``}TLE90\text{''})}(\$1), Title : \$3]). \end{array}$$

The selection operation has been pushed inside the tuple construction (that can be viewed as a product). We introduced a λ-expression in order to denote the element \$1 in the algebraic operation. It must be noted that, as before, we use selection on an element (here a string). If \$1 is not $TLE90$, the selection of \$1 is $\perp$, the tuple is also $\perp$ (since the standard product with an empty set is the empty set), and the value of the projection is $\perp$ as well (since the standard projection of an empty set is an empty set). Note also that the attribute *Authors* has been projected out and that \$2 is now simply ignored. The advantage of the above optimized construction is that it avoids the construction of irrelevant values.
The optimized SS for a query φ is obtained as follows:

1. add the new start symbol (φ applied to the previous start symbol) and the new rule for it.
2. apply the rewrite rules to generate alternative execution plans:
   (a) The *grammar rewriting rule* push queries down the grammar specification. If in a rule each occurrence of some $\$i$ is inside some unique algebraic query ψ (i.e., as $\psi(\$i)$), replace $\psi(\$i)$ by $\$i$ and replace the corresponding parser, say $\langle A\rangle$ by $\langle\psi(A)\rangle$. If such a rule does not exist yet, add a rule for parsing $\langle\psi(A)\rangle$ in the obvious way.
   (b) Apply algebraic *query rewriting rules* to the algebraic expressions in the action parts of the rules.

Clearly, heuristics (such as selection pushing) have to be applied as in standard query optimization. The grammar rewriting rule must be applied whenever possible. In the best case (e.g., the previous example), the only values that are constructed are the ones returned by the query. In the worst case, the query remains at the root of the parse-tree and the whole database is constructed.

We conclude this section with a remark on the reduction.

Remark 3.1 The *grammar rewriting rules* introduce new nonterminals and thus new rules. Some nonterminals may become obsolete as shown by the *BibTex* example (the nonterminal Ref_Set). Observe however a nonterminal $\langle V\rangle$ may be replaced locally by $\langle\varphi(V)\rangle$ in a portion of the grammar but continue to be used in another portion. We will not consider here the problem of reducing the grammar to remove obsolete nonterminals. □

We next sketch possible improvements of the technique for the same data model and finally, for object-based databases.

3.2 Improvements

We briefly consider possible improvements of the technique for the complex value model:

Saving more Data Construction

Consider the BibTex example and suppose that we want to select the reference with a certain keyword, say "AA92". Note that although the selection will be pushed to the leave of the query tree, the optimized SS still constructs the set of authors for all references (even when it is already "known" by the system that this reference is not of interest and will eventually be discarded). This is because we maintain some form of independence between the sub-parsers, e.g., the parser for $\langle Author\_Set\rangle$ does not know whether the selection on keyword succeeded. Clearly, one can achieve more optimization if data construction can be controlled more globally.

Modifying the grammar

For a given query, one can transform the grammar into a new grammar that will focus on the needed data and would avoid as much as possible parsing portions of the text not concerned with the query. For instance, for the selection of reference "AA92", a solution is to scan the file until we find the particular keyword and then resume to the normal parsing.

Using Full-Text Indexing

Even if we modify the grammar (as just suggested), we still have to parse the entire file. The gain thereby obtained remains marginal compared to the gain provided by the core technique since data construction is typically more expensive than parsing. However, for large files, some important speed-up can be achieved [12] when the scanning of the file is replaced by an access via a full-text index. E.g., we could obtain immediately, in the BibTex example, the portion of the file pertaining to the specific keyword.

Join queries

The technique is well-fitted for "simple" queries, i.e., queries involving some projection/selection of collections stored in the database. Now, let us consider complex queries, and for instance queries involving also joins. This may involve joining data within a single file: e.g., a selection of citations in a LaTex file containing LaTex formatted citations. It may also involve joining data from several files and more than one SS: e.g., a selection of citations in a LaTex file of BibTex references from another file.

The problem that arises is similar to that of complex queries in any rewriting system: the query has to be rewritten to obtain a join involving several sub-queries on distinct elements. Once this is done, the sub-queries can be "pushed" into the SS building those distinct elements, using the optimization algorithm described above. Then, the join can be performed. In the OO context, algebraic equivalences for performing this style of rewriting can be found in [23, 24, 28].

3.3 Dealing with objects

In this section, we consider an extension of the technique to handle objects. To allow our SS's to construct objects, we extend the set of operations used in the action part of rules and add a *new* operation that allows the (virtual) creation of objects. For instance, we can define a class *References* with associated type $tuple(Key : string, Authors : \ldots)$, and modify the BibTex SS as follows: replace the action $\$\$:= [Key : \$1, Authors : \$2 \ldots]$ in the definition of $\langle Reference \rangle$ by

$$\$\$:= new(References, [Key : \$1, Authors : \$2 \ldots]).$$

Thus, the file will be represented by set of objects of class *References* (rather than by a set of tuples).

Suppose that we want to evaluate the OQL query

$$\begin{array}{ll} \textbf{select} & r \rightarrow reformat \\ \textbf{from} & r \textbf{ in } References \\ \textbf{where} & r.Key \; = \; \text{“}TLE90\text{”}. \end{array}$$

This query is very similar to the one considered at the beginning of Section 3.1, except that it now selects a set of objects and applies the method $reformat$ to these objects. Essentially, the same optimization technique can be applied, and will push the selection into the SS. It is important to observe that the rewriting is such that the resulting structuring schema (that answers to the query) creates objects only for references whose keyword is $TLE90$. This is achieved using the following rewriting (we focus only on the part of the optimization involving object creation):

$$\sigma_{Key=\text{``}TLE90\text{''}}(new(References, [Key : \$1, Authors : \$2 \ldots]))$$

$$\leadsto$$

$$new(References, \sigma_{Key=\text{``}TLE\text{''}}([Key : \$1, Authors : \$2 \ldots]))$$

Observe that object creation and selection commute. The advantage of pushing down the selection into the object creation is that when the selection fails, it returns $\perp$, and then object creation also fails and returns $\perp$. So, we avoid creating unnecessary objects which results in dramatic gain in performances.

The use of objects in the above SS is rather simple and does not involve an essential aspect of the object oriented model: object sharing. To conclude this section, we consider two main issues raised by object sharing:

1. How do we avoid duplication of objects?

2. How do we handle cyclic data?

Avoiding Duplicate Objects

As shown in [4], duplicate elimination is a crucial problem for languages allowing the creation of objects. We will see that it is also true in our context. To illustrate this, we visit again the BibTex example. Assume now that we want to represent authors by objects. For that, we define a class of Authors (where the state of each author is a string, namely the author name); and annotate the rule defining the nonterminal $\langle Author \rangle$ as follows:

$$\langle Author \rangle \quad \rightarrow \quad \#String \quad \{\$\$:= new(Authors, \$1)\}$$

Note however that in this SS every occurrence of an author name causes the creation of a new object. Thus an author who wrote many papers will be represented by many distinct objects. This is clearly not what is meant by the file. This leads to a more controlled operation for object creation: *c-new* (for conditional *new*). The *conditional new* primitive requires three parameters: the name of a class C, a predicate p and a value v. If an object of value v satisfying predicate p has already been built in class C, *c-new* returns that object. Otherwise, it creates a new object of value v.

It must be noted that this primitive requires that the system maintains the extent of the class. Observe also that now data constructions performed in non overlapping part of the parse-tree are no longer independent. This complicates optimization since an object needed for the query may be modified by side effects in a portion of data that is seemingly not relevant to the query. This is yet another demonstration of the difficulty of performing query optimization in presence of side effects.

Dealing with Cyclic Data

We next consider cyclic data. A more symmetric way to view references and authors is as objects with the following structure:

$$
\begin{array}{llcll}
Class & Reference & = & tuple(& key : string, \\
 & & & & authors : set(Author), \\
 & & & & title : String, \\
 & & & & \ldots \qquad\qquad) \\
Class & Author & = & tuple(& name : String, \\
 & & & & refs : set(Reference) \quad)
\end{array}
$$

This definition introduces a cycle between the references and the authors. A reference will be linked to a set of authors and an author to a set of references. In order to maintain this cyclic data, the user will have to create objects in one place, and update their state in another. This requires again using side effects in the SS's. The annotated rules corresponding to the new definition of Class *Authors* is as follows:

$$
\begin{array}{lll}
\langle Reference \rangle & \rightarrow & \ldots \\
 & & \text{``}author = \text{''} \langle Author\_Set \rangle \\
 & & \ldots \\
 & & \quad \{\$\$:= new(Reference, [\ldots, authors : \$2, \ldots]); \\
 & & \quad map_{\lambda x \cdot (x.refs := x.refs \,\cup\, \{\$\$\})}(\$2)\} \\
\langle Author \rangle & \rightarrow & \#String \\
 & & \quad \{\$\$:= c - new(Author, \lambda x \cdot \sigma_{x.name = \$1}, \\
 & & \qquad\qquad [name : \$1, refs : \{\}])\}
\end{array}
$$

In the rule defining $\langle Reference \rangle$, the variable \$2 denotes a set of authors. The *map* primitive is used to apply the update function on every elements of this set. The *map* statement adds the current reference to each author in the set of authors.

The fact that we have update operations, and allow the action part to contain a sequence of statements complicates the optimization process: under what conditions can a query be pushed into a rule involving updates? and when pushing is allowed, how is a query pushed on a sequence of updates?

Assume for example that we want to evaluate the query $\Pi_{Key}(References)$. Since the rule defining $\langle Ref\_Set \rangle$ contains only one assignment to \$\$, and no additional updates, the query can be pushed into this rule as before. After further (query) rewriting, we would like to push Π_{Key} into the rule defining $\langle Reference \rangle$. Now, having an added update after the assignment on \$\$, we have to check if this update effects the *Key* attribute. Since the update does not modify the key and is irrelevant to the query result, it can be removed from the rule, and the query can be pushed as before on the expression assigned to \$\$.

On the other hand, consider the query $\sigma_{Key=\text{``}TLE90\text{''}}(References)$. At some point of the optimization process, one wants to push the query into the rule defining $\langle Reference \rangle$. Note that this query returns the whole reference, including the *Authors* attribute. Thus clearly we can not remove the update to the set of authors. Furthermore, since the value of \$\$ is used to update the *Refs* attribute of the authors, the query must not be pushed on the expression assigned to \$\$.

When objects are involved, updates to one attribute can effect the value of other attributes. This is due to object sharing. The relationship between the various assignments and updates in a rule can be very tricky. One possible simple solution is to forbid pushing queries into rules involving updates. A

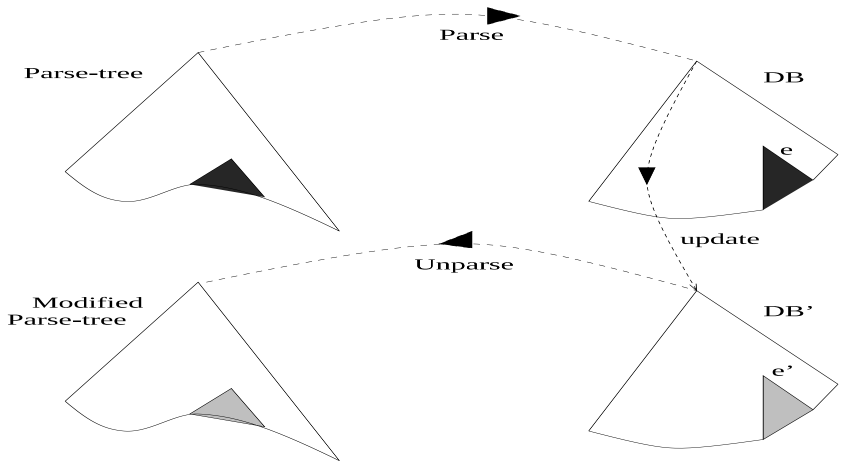


Figure 1: The Naive Propagation

more lenient solution is to distinguish special cases (like the one mentioned above) where query pushing is safe, and develop rewrite rules to handle such specific cases.

4 Updates

We considered queries and studied the optimization of queries specified on the database view of the file. In this section, we consider updates. We assume that updates are specified at the database level and translated into updates on the file. We next present a (not so simple) "naive" technique for update propagation based on some unparsing of database values. Section 5 introduces a locality condition, and presents a more efficient technique for propagating updates in structuring schemas satisfying this locality condition.

We are not concerned here by the language used to specify updates. (Any SQL-like language could be considered.) To simplify, we view database instances as trees, and elements in the database as particular subtrees. We also assume that updates are expressions of the form $replace(db, e, e')$ where (i) db is the tree representation of the database, (ii) e is a subtree rooted in some vertex of db and (iii) e' is a subtree of the same type. The result of the update is the tree obtained by replacing in db, the sub-tree e by e'.

The *naive* way of propagating an update from database to file is to perform the following three steps (See Figure 1): (i) materialize the database (i.e., parse of the file and construct db using the SS); (ii) update db; and (iii) unparse the new database to produce the updated file. We already know how to perform the first two steps. In this section, we present the unparsing technique required for the third step. This is an adaptation to our context of folklore techniques from parsing.

The goal of the unparsing of a database db (given a SS with start symbol T_0) is to produce a file f such that $f \rightsquigarrow_{T_0} db$. The unparsing process uses an auxiliary notion of *matching*, and a *matching algorithm* that we present next. Once this is done, we present the unparsing algorithm when the set of functions F used in the SS is empty, then consider the case when F is not empty. Finally, we study two properties of structuring schemas that are important for updates.

4.1 Matching for $F = \emptyset$

Let t be an action term containing constants and variables (i.e. $\$i$'s), and using some of the following constructors: set, tuple, bag, cons (for sets, lists or bags), union (for sets or bags), and concatenation (for lists or strings). Let v be a data value. A *matching ν for t and v* is an assignment of values to the $\$i$ variables occurring in t, such that $\nu(t) = v$. For instance, let $t = \{“a”||“b”||\$1, “b”||\$2\}$ and $v = \{“bcd”, “abc”\}$. The assignment $\nu(\$1) = “c”, \nu(\$2) = “cd”$ is a matching for t and v.

The notion of *matching* that we use here is rather standard. The only difficulty is that the matching must take into account the properties of the set, bag, list and string constructors (e.g., commutativity, idempotence and associativity of the set constructor). Let t be a term and v a data value. Then the *matchings* of t and v, denoted $match(t, v)$, are defined as follows.
below.

Basis: The base cases are as follows:
(1) If $t = \$i$ (for some $\$i$ variable), $match(t, v)$ succeeds and returns the assignment ν, such that $\nu(\$i) = v$ and $\nu(\$j)$ is undefined for each $\$j \neq \$i$. (More precisely, it returns a set consisting of a single matching.)
(2) If $t = v' \neq v$ (for some constant v'), then the matching fails (i.e. $match(v', v)$ returns the empty set.
(3) If $t = v$ then $match(v, v)$ succeeds and returns the assignment ν where $\nu(\$i)$ is undefined for all variables.

Recursion: The recursion works as follows:
To match a term $t = f(t_1, ..., t_n)$ against a value v, $match(t, v)$ tries to find sequences of values $v_1, v_2, \ldots, v_n$ such that (i) $v = f(v_1, ..., v_n)$, (ii) for each $i = 1 \ldots n$, v_i matches t_i, and (iii) the matchings are compatible (i.e., do not assign distinct values to the same $\$i$ variable).

Observe that, since $F = \emptyset$, for each data value v and each n-ary constructor f used in the action part, one can easily construct the finite set Δ of candidate sequences $v_1, ..., v_n$ such that $v = f(v_1, ..., v_n)$. For instance, if f is set construction, (i.e. $t = \{t_1, ..., t_n\}$), then either v is a set with more than n members, in which case no sequence satisfying (i) exists and Δ is empty; or v is a set containing n or less elements, in which case Δ contains all sequences of elements in v such that each element occurs in the sequence at least once. We then use the recursion to select the sequences in Δ satisfying (ii) and (iii).

The above matching algorithm returns a set of appropriate assignments for the variables in t (possibly empty if the matching failed).

We are now almost ready to define unparsing. We still need two auxiliary functions:

1. Let T be a nonterminal or non constant token in the structuring schema. Since we only consider reduced grammars, there exists a string that is accepted by T. The function $default$ takes a nonterminal or non constant token as input and returns some corresponding sting. (We will need this function when the variable $\$i$ corresponding to T does not occur in the action part of a rule, i.e., if there is no constraint on the string that T accepted.)

2. Function *build_string* has for input a pair with a rule r and an array $[w_1, ..., w_n]$ of strings where n is the number of nonterminals or non-constant tokens in the body of the rule. This function has already been defined in Section 2. (Recall particular that if each w_i is accepted by T_i, and T is the nonterminal in the head of rule r, then $build\_string(r, [w_1, ..., w_n])$ is accepted by T.)

4.2 Unparsing for $F = \emptyset$

Let S be a structuring schema. We define a function *unparse$_S$(T,v,history)* that takes as argument a nonterminal T of S, a database value v, and a "history" and returns, (i) if it succeeds, a pair $[true, s]$ such that $s \leadsto_T v$; and (ii) otherwise $[false, ``"]$. The *history* parameter is a stack used to record previous unparsing attempts and prevent infinite loops. To unparse a database db, we compute $unparse(T_0, db, -)$, where T_0 is the start symbol of S and $-$ is the empty stack.

> **function** $unparse_S(T, v;$ **var** $history)$ **returns** $[boolean, string]$
> $success := false$;
> If $[T, v]$ is in $history$ return $[false, ``"]$;
> push$(history, [T, v])$;
> For each rule r defining T and while not $success$
> $[success, s] := unparse\_rule_S(r, v, history)$;
> pop$(history)$;
> return $[success, s]$
> **end**

The function *unparse$_S$* uses a function *unparse_rule$_S$*. Given a database value v and a rule r, the function *unparse_rule$_S$*, returns, if it succeeds, a string s, such that there exists a parsing of s (with the structuring schema S) starting with the rule r, that constructs the value v.

> **function** $unparse\_rule_S(r, v, history)$ **returns** $[boolean, string]$
> let t be the action of r;
> let $st[1..n]$ be an array of n strings
(where n is the number of nonterminals or non constant tokens in r;)
> $matchings := match(t, v)$;
> $success := false$;
> For each matching m in $matchings$ and while not $success$
>   { $success := true$;
>   For each $\$i$ and while $success$
>   Let A_i be the nonterminal corresponding to $\$i$.
>   if $\$i$ is used in the action part
> $[success, st[i]] := unparse_S(A_i, m(i), history)$;
>   else ($\$i$ is not used in the action part)
> $st[i] := default(A_i)$;
>   }
> if $success$ return $[true, build\_string(r, st)]$;
> else   return $[false, ``"]$;
> **end**

Theorem 4.1 For $F = \emptyset$, the above unparsing algorithm terminates on each input db and succeeds returning a string s such that $s \leadsto_{T_0} db$ iff such a string exists.

Proof: The termination of the algorithm comes from the fact that *unparse* is called recursively with values of smaller and smaller size and loops are detected. The correctness essentially results from the careful case analysis performed by the algorithm. □

We illustrate the algorithm using the BibTex SS presented in Section 2. Consider the set of references $v = \{ref_1, ref_2, \ldots, ref_n\}$. Assume that we want to unparse v w.r.t the root literal $\langle Ref\_Set\rangle$ of the BibTex grammar. There are two rules defining $\langle Ref\_Set\rangle$:

$$\begin{array}{lll} \langle Ref\_Set\rangle & \rightarrow & \langle Reference\rangle\ \langle Ref\_Set\rangle \quad \{\$\$:= \{\$1\} \cup \$2\} \\ & | & \epsilon \qquad\qquad\qquad\qquad\qquad\quad \{\$\$:= \{\}\} \end{array}$$

To unparse v we try to use the rules. In each rule we try to match v against the term in the action part. We start with the first rule. One possible matching for v and $\{\$1\} \cup \$2$ is $\nu(\$1) = ref_1$, $\nu(\$2) = \{ref_2, \ldots, ref_n\}$. Our next step is to try and unparse the value assigned to each $\$i$ w.r.t. the corresponding non terminal. If the unparsing terminates successfully, we will have strings representing the two values, and the function *build_string* will concatenate them and obtain a string representing the whole set of references.

The variable $1 corresponds to the non terminal $\langle Reference\rangle$, and $2 corresponds to $\langle Ref\_Set\rangle$. We start by unparsing ref_1 w.r.t $\langle Reference\rangle$. Assume that ref_1 is the tuple

$ref_1 = [Key : TLE90, Authors : \{$"*A.A. Toto*", "*H.R. Lulu*", "*A.B.C.D. Eux*"$\} \ldots,]$

To unparse ref_1 the procedure considers the rules defining $\langle Reference\rangle$. There is one such rule:

$$\begin{array}{lll} \langle Reference\rangle & \rightarrow & \text{“}@Inproceedings\{\text{”}\ \#String \\ & & \text{“}author = \text{”}\ \langle Author\_list\rangle \ldots \\ & & \qquad \{\$\$:= [Key : \$1, Authors : \$2 \ldots]\} \end{array}$$

When ref_1 is matched against the action part of the rule, the matching assigns "TLE90" to $1, {"*A.A. Toto*", "*H.R. Lulu*", "*A.B.C.D. Eux*"} to $2, etc. The algorithm now proceeds by unparsing these values w.r.t. to the corresponding non terminals. When the process terminates, we have strings corresponding to each such value. The strings are then combined together to get a string representing ref_1. The result will have the form

$$\text{“}@Inproceedings\{TLE90, author = \text{“}A.A.\ Toto\ and\ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{”}, \ldots\}\text{”}$$

Now consider the unparsing of $\{ref_2, \ldots, ref_n\}$ w.r.t. the non terminal $\langle Ref\_Set\rangle$. The unparsing proceeds as above, unparsing recursively smaller and smaller sets, until we are left with the empty set. When this happens, the set can only be matched against the second rule of $\langle Ref\_Set\rangle$, and the corresponding empty string ϵ is returned.

When the unparsing procedure terminates, we obtain a BibTex file containing entries for all the references in the unparsed set v.

Note that the unparsing algorithm may require time exponential in the size of the unparsed value. This clearly motivates studying techniques to minimize the size of elements being unparsed. We will present such techniques in Section 5.

4.3 Unparsing for $F \neq \emptyset$

The above unparsing algorithm applies to SS's with F empty. We next consider more general actions. Not surprisingly, unparsing becomes more complicates, and sometimes undecidable.

The problem of deciding for a structuring schema S whose start symbol is T_0 and a value v whether there exists a string s for which $s \leadsto_{T_0} v$ is called the *unparsing problem.* The unparsing algorithm presented above solves the problem for the case where F is empty. (The algorithm succeeds in unparsing v iff such s exists). The difficulty of the problem for non empty F is demonstrated by:

Theorem 4.2 The unparsing problem is undecidable for structuring schemas where F contains the operations *join* and *projection*.

Proof: The proof is by reduction from the problem of testing if the intersection of two context-free grammars G_1 and G_2 is empty, known to be undecidable. Let G_1, G_2 be two context free grammars. We construct a SS $S(G_1, G_2)$ and a value v such that v can be unparsed w.r.t. $S(G_1, G_2)$ iff $L(G_1) \cap L(G_2)$ is nonempty.

We assume w.l.o.g. that the alphabets of nonterminals in G_1, G_2 are disjoint. We first define for each G_i $(i = 1, 2)$, a SS S_i, such that the database view of a word accepted by the grammar G_i is the word itself (e.g. all the actions in these schemas are simple string concatenation). Let T_1, T_2 be the start symbols of G_1, G_2 resp, and let T be a new nonterminal that does not appear in G_1, G_2. The SS $S(G_1, G_2)$ has start symbol T, and is obtained by adding the following rule to the rules of S_1 and S_2:

$$T \quad \rightarrow \quad T_1 \nabla T_2 \quad \{ \quad \$\$:= \pi_{\emptyset}(\{[at_1 : \$1]\} \bowtie \{[at_1 : \$2]\}) \quad \}$$

where ∇ is a new terminal symbol.

Consider a word w accepted by $S(G_1, G_2)$. Clearly, $w = w_1 \nabla w_2$ for some w_i in $L(G_i)$, $i = 1, 2$. For each i, by construction of S_i, we have $w_i \leadsto_{S_i, T_i} w_i$. Two cases arise:

- $w_1 = w_2$. Then the join is nonempty and $w \leadsto_S \{[\,]\}$, i.e., the singleton set containing the tuple $[\,]$.
- $w_1 \neq w_2$. Then the join is empty and $w \leadsto_S \emptyset$.

Therefore, $\{[\,]\}$ can be unparsed iff there is a word $w \in L(G_1) \cap L(G_2)$. $\Box$

Although undecidability is obtained with simple functions, can we still allow some functions in F? The difficulty is to find the matchings. For f in F, we are lead to match a term $f(t_1, ..., t_n)$ with some value v, and so to compute $f^{-1}(v)$. One issue is thus the existence of some inverse for f. For instance, consider the above undecidability theorem. It uses *join* and *projection* for which inverses are not finite. This is a first cause of failure of the technique.

There exists a second origin for the technique to fail. It turns out that the unparsing algorithm modified to handle invertible functions in F may not terminate. (By "invertible" we mean, that for each f in F and each v, $f^{-1}(v)$ is finite and computable.) To illustrate this, consider the structuring schema:

$$\begin{array}{rcll} A & \rightarrow & A\text{"."} & \{\$\$:= \mathit{decrement}(\$1)\} \\ & | & \epsilon & \{\$\$:= 5)\} \end{array}$$

Intuitively, a sequence of i dots yields the database consisting of the single integer $5-i$. Now consider the above unparsing algorithm (with matching extended to handle decrement) on input 6. The unparsing of 6 leads to matching 6 with *decrement*($1) assigning 7 to $1. This leads to the unparsing of 7, then 8, 9, etc.

The loop detection that we use is not sufficient for this function. The technique can be adapted by using more sophisticate loop detection based on the monotony of functions in F.

To conclude this section on updates and unparsing, we consider two important properties of SS's that are essential for unparsing and thus for updates.

4.4 Information Loss and Constraints

We can make the following observations on the mapping defined by a SS:

1. First, this mapping is possibly not one-to-one, i.e., two distinct files may yield the same database value. From a representation viewpoint, this means that there is loss of information when going from the file to the database. From an update viewpoint, this means that the result of a database update is not completely specified.

2. Second, this mapping is possibly not onto, i.e., a database may correspond to no file. From a representation viewpoint, this means that there are constraints on the instances of the database schema. An instance is *valid* only if it corresponds to some file. From an update viewpoint, this means that the result of a database update may be invalid.

Let us consider first the *loss of information*. For some input value, the algorithm returns one particular string among all possible strings that have the same database image. In particular, the operators used in the action part of rules do not guarantee that the mapping is one-to-one. E.g., the decision to represent a sequence of strings from the file by a list, bag or set, has a clear impact on the interpretation of the data, and may lead (depending on the chosen representation) to some information loss. If the file is represented by a list, then it is meant that the order is relevant (the data constructor is not commutative). If it represented by a bag, duplication is relevant (no idempotence). If it is represented by a set, this implies that duplication and order are irrelevant for this particular collection. This is yet another illustration of the fact that structuring schemas bring semantics to the file beyond the grammar that is used to parse it.

In some applications, one might want to insist on the fact that the database is a faithful image of the file. This suggests the following notion. A SS is said to be *lossless* if for every f, f', $f \rightsquigarrow_{T_0} v$ and $f' \rightsquigarrow_{T_0} v$ implies that $f = f'$.

Let us now consider the second property that we call "constraint-freedom". As mentioned above, the unparsing process may fail on some database updates. This happens when the new database is no longer an image of some file (according to the structuring schema). Consider for instance the following structuring schema with one rule:

$S \rightarrow \#string \qquad \{\$\$:= \{\$1\}\}$

Suppose that we have a singleton set containing one string, say "Peter", and we insert "Mary". We then attempt to unparse the value {"Peter","Mary"}. This will fail since the parser constructs singleton sets only. To see another example, consider the SS's in Figure 2. Then S_1 is constraint-free whereas S_2 is not. The value $\langle 1, 2 \rangle$ cannot be derived from T in S_2.

A structuring schema is said to be *constraint-free*, if for each database db over the proper schema, there exists a file f such that $f \rightsquigarrow_{T_0} db$. This second property of structuring schemas may be useful in other applications.

Unfortunately, lossless and constraint-freedom are both undecidable properties.

Theorem 4.3 For a SS S, the following problems are undecidable (even for $F = \emptyset$): (i) testing if S is lossless, (ii) testing if S is constraint-free.

Proof: To prove (i), we use a reduction of the problem of deciding whether the intersection of two context-free languages is empty, known to be undecidable. Consider two languages G_1 and G_2 over the same alphabet. Let a_1, a_2 be two new letters. Consider the language $L = L(G_1)a_1 \cup L(G_2)a_2$. Clearly

S_1	$T \rightarrow int\ int\ T$	$\{\$\$:= cons(\$1, cons(\$2, \$3))\}$
	$T \rightarrow int$	$\{\langle \$1 \rangle\}$
	$T \rightarrow \epsilon$	$\{\langle\rangle\}$
S_2	$T \rightarrow int\ int\ int\ T$	$\{\$\$:= cons(\$1, cons(\$2, cons(\$3, \$4)))\}$
	$T \rightarrow int$	$\{\langle \$1 \rangle\}$
	$T \rightarrow \epsilon$	$\{\langle\rangle\}$
S_3	$T \rightarrow$ "a" $int\ T_1$	$\{\$\$:= cons(\$1, \$2)\}$
	$T \rightarrow$ "b" $int\ int\ T_2$	$\{\$\$:= cons(\$1, cons(\$2, \$3))\}$
	$T_1 \rightarrow int\ int\ int$	$\{\$\$:= \langle \$1, \$2, \$3 \rangle\}$
	$T_2 \rightarrow int$	$\{\$\$:= \langle \$1 \rangle\}$
S_4	$T \rightarrow$ "a" $int\ T_1$	$\{\$\$:= cons(\$1, \$2)\}$
	$T \rightarrow$ "b" $int\ int\ T_2$	$\{\$\$:= cons(\$1, cons(\$2, \$3))\}$
	$T_1 \rightarrow int\ int$	$\{\$\$:= \langle \$1, \$2 \rangle\}$
	$T_2 \rightarrow int$	$\{\$\$:= \langle \$1 \rangle\}$

Figure 2: 3 Structuring Schemas

L is context free. One can easily obtain a SS that transforms each word wa_i $(i = 1, 2)$ in L into $[A : w]$. This SS is lossless iff $L(G_1) \cap L(G_2) = \emptyset$.

We next prove (ii). This is proved by reduction of the problem of deciding whether a context-free language L over the two-letter alphabet $\Sigma = \{0, 1\}$ coincide with Σ^*. This is also known to be undecidable.

Suppose that there is a decision procedure to test whether a schema is constraint free. Let G be a context-free grammar of alphabet Σ. In structuring schemas, we use the sort "string". A string is a word over some alphabet $\mathcal{C}$ of characters. Observe that Σ is a subset of $\mathcal{C}$. Consider the language:

$$L' = L(G) \cup \{\mathcal{C}^* a \mathcal{C}^* \mid a \in \mathcal{C} - \Sigma\}.$$

Then L' is clearly context-free. Let G' be a grammar for L'. It is easy to take the grammar G' and annotate it to get a SS $S(G')$ such that for each word w, $S(G')$ constructs the value $[A : w]$, where A is an attribute of type strings (of characters in $\mathcal{C}$). One can verify that $L(G) = \Sigma^*$ iff $S(G')$ is constraint-free:

- Suppose $L(G) = \Sigma^*$. For each database value $[A : w]$, two cases occur:
  - w contains some a not in Σ and w is in L' and $parse(w)$ is $[A : w]$.
  - $w \in \Sigma^* = L(G)$ and $parse(w)$ is $[A : w]$.

  Thus $S(G')$ is constraint-free.

- Now suppose that $L(G) \neq \Sigma^*$. Let w be in $\Sigma^* - L(G)$. The value $[A : w]$ is not the image of any string parsed by G', thus $S(G')$ is not constraint-free.

□

Since these two properties are undecidable and are needed by certain applications, it is interesting to show sufficient properties that guarantee losslessness and constraint freedom. To the least, we need to isolate classes of SS's for which these properties are decidable. To do that, we will to impose severe restrictions. (In particular, set and bag constructions unless used in a trivial or redundant manner cannot be allowed.) Note that these restrictions do not apply in general since for most applications, the two properties are not compulsory.

We will define a class of lossless schemas using the following auxiliary notion:

Definition 4.4 A SS is *rule-split* iff for each non terminal T and rules r_1, r_2 defining T, the set of values obtained by derivations starting from T with rules r_1 and r_2 resp., are disjoint.

For instance, consider the SS's in Figure 2. Then S_3 is split-rule whereas S_4 is not. The value $\langle 1, 2, 3 \rangle$ can be derived from T using the two first rules. Observe also that S_3 is lossless whereas S_4 is not: the strings "a 1 2 3", and "b 1 2 3" are mapped to the same value.

Definition 4.5 A SS is in class *LossLess-1* if the following holds:
(1) the actions only use the following constructors: *tupling* (as in $[A : x, B : y, C : z]$), *list* (as in $\langle x, y, z \rangle$), *cons* of lists (as in $cons(x, y)$ where y is a list);[3]
(2) for each rule, each $\$i$ occurs exactly once in the action;
(3) the schema is rule-split.

The restriction posed on schemas in class LossLess-1 assure that

Theorem 4.6 Each SS in class LossLess-1 is lossless.

Proof: Let S be a LossLess-1 SS. Let v be a database value such that for some T and distinct w, w', $w \rightsquigarrow_T v$ and $w' \rightsquigarrow_T v$. Assume also that v is a minimal such v (no such v "smaller" exists for some reasonable definition of smaller). By (3), the two derivations use the same rule r. Let $T_1, ..., T_n$ be the nonterminals and non-constant symbols in r. Let $t(\$1, ..., \$n)$ be the action of r. Let $w_1, ..., w_n$ be the substrings of w parsed by the parsers for $T_1, ..., T_n$ when parsing w and $v_1, ..., v_n$ the values they return. Let $w'_1, ..., w'_n, v'_1, ..., v'_n$ be defined similarly for w'. Since all constructors are one-to-one (by (1)) and since each $\$i$, $1 \leq i \leq n$, appears at least once in the action t (by (2)), $v_i = v'_i$ for each i, $1 \leq i \leq n$. By the minimality of the choice of v, $w_i = w'_i$ for each i, $1 \leq i \leq n$. Then since the two parsings used the same rule r, $w = w'$, a contradiction. Thus S is lossless. □

This class yields a sufficient condition for losslessness. We show next that one can decide whether a SS is in class LossLess-1. The test we will use is quite expensive and is given here mostly as an indication that such classes indeed exist. Other classes with "cheaper" decision procedures should be considered. Indeed, even for the class we present, we believe that simpler and less expensive tests can be obtained although their presentation would be more intricate. Since this is not central to the paper, we prefer to use a simple and brute force test.

Proposition 4.7 One can decide whether a SS is in class LossLess-1.

Proof: Conditions (1) and (2) are syntactic and easy to check. Condition (3) is tested by reduction to a problem on (non-deterministic top-down) tree-automata [30], using the facts that tree automata are closed under boolean operations, and that test for emptiness is decidable for such automata.

[3] So in particular the set of functions F that can be used in actions is empty.

Consider a schema satisfying Properties (1) and (2). One can decide whether it satisfies (3) as follows. Values of such SS can be represented abstractly as trees. A list $\langle v_1, ..., v_n \rangle$ is represented by the tree $(left : v_1, right : (left : v_2, right : (...(left : v_{n-1}, right : v_n)...)))$ (where $left, right$ are respectively the labels of edges leading to left and right children). An n-ary tuple $[v_1, ..., v_n]$ is represented by the tree $(A_1 : v_1, ..., A_n : v_n)$. An integer is represented by a leaf labeled by a particular symbol, say i for integer, and similarly for strings s, reals r and booleans b.

For each nonterminal T and each rule r, let $tree(T, r)$ denotes the set of all trees corresponding to values constructed by parse trees rooted at T, and where the rule used at the root is r. One can construct a non-deterministic top-down tree automata accepting precisely $tree(T, r)$. The automaton simulates the parsing of a word. The states of the automaton are the (non)terminal of the grammar. This is possible because of the restrictions on LossLess-1 SS's.

Now tree automata are closed under boolean operations, so for each T and each pair of rules r_1, r_2 for T, one can construct an automata $A(T, r_1, r_2)$ for $tree(T, r_1) \cap tree(T, r_2)$. It now suffices to test whether the language accepted by $A(T, r_1, r_2)$ is empty.

Observe that we ignore here the actual values of atomic elements (e.g., integers) from the string. This is because LossLess-1 SS's essentially do not constrain or interpret such elements in any way. $\Box$

It is also possible to test for constraint-freedom of schemas in class of LossLess-1.

Theorem 4.8 One can decide if a structuring schema in class LossLess-1 is constraint-free.

Proof: (sketch) The proof is again by reduction to the emptiness test for tree-automata.

Let $A(S)$ be the tree automata "accepting" the values generated by some LossLess-1 SS S (built as explained above). Suppose that the type of values generated by S is τ. Let $A(\tau)$ be an automata accepting the set of all values of type τ. Consider the set of trees $L = L(A(\tau)) - L(A(S))$ is empty. Since tree automata are closed under boolean operations, one can construct an automata $A'(S)$ for L. It now suffices to test whether $L(A'(S))$ is empty. $\Box$

5 Update Optimization

We now study the optimization of the propagation of updates (specified on the database) to the file. We saw that unparsing is expensive (when possible). We will therefore mostly concentrate our efforts on minimizing unparsing. For that, we introduce a notion of *correspondence* between databases and parse-trees, a *locality* property (that is in general undecidable), and a large class of structuring schemas that are local.

The naive technique for update propagation: (i) compute the database (if this has not been done yet), (ii) perform the update, and (iii) unparse the database. This brute force solution presents two serious drawbacks: (1) we may have to construct the entire database although the update may involve only a small part of it; and (2) we may have to unparse the entire database although the update may change only a small part of it. Figure 3 illustrates the *optimized* update propagation technique. Observe in the figure that we minimize the database construction by building only a relevant portion of the database (a solution to (1)); and we minimize unparsing by focusing on the updated part of the database (a solution to (2)).

We distinguish two phases in an update. In the first phase, the query language is used to select the database portions thats is relevant to the update and this portion is constructed. The update then

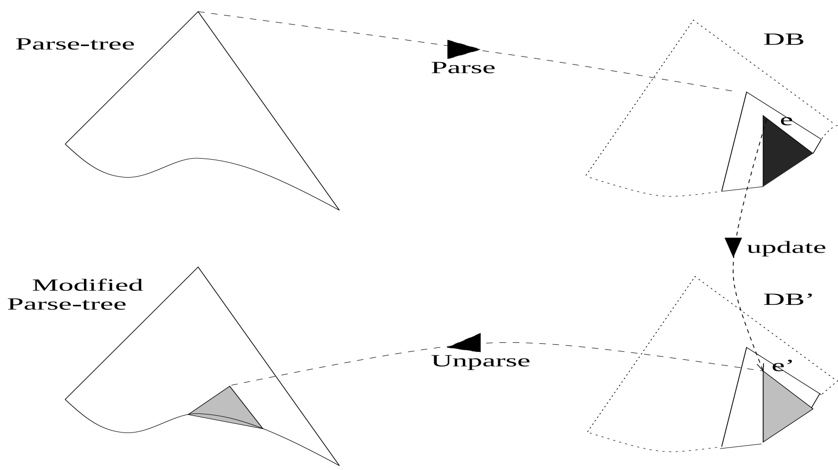


Figure 3: The Propagation Technique

consists of the replacement of some subtree e by some new subtree e'. In a second phase, we replace e by e'. In the query phase, we can apply the optimization technique of Section 3 to avoid constructing the whole database. The minimization of the unparsing is discussed next.

At the database level, an update is a replacement of one subtree e of the database-tree by a new tree e'. In order to propagate such an update to the file, one is tempted to try to find the subparse-tree(s) corresponding to e and replace it (them) by the result of the unparsing of e'. Unfortunately, there are several difficulties in pursuing this approach. First, it is not simple to define what "a subtree that corresponds to a database element e" means. Also, we have to make sure that the parse-tree obtained by the subtree replacement is valid and that it indeed yields the correct updated database.

To pursue the development along these lines, we have to make more precise the correspondence between vertexes in the parse-tree and in the database-tree. Assume for example that we parse the string "a,a,b,c" using the structuring schema:

$$
\begin{array}{lcll}
A & \rightarrow & B\text{“,”}C\text{“,”}D\text{“,”}E & \{\ \ \$\$:= [A_1 : [A_{1,1} : \text{“}abc\text{”}, A_{1,2} : \{\$1\} \cup \{\$2\}], A_2 : \$3, A_3 : \$3] \ \ \} \\
B & \rightarrow & \#String & \{\ \ \$\$:= \$1\} \\
C & \rightarrow & \#String & \{\ \ \$\$:= \$1\} \\
D & \rightarrow & \#String & \{\ \ \$\$:= \$1\} \\
E & \rightarrow & \#String & \{\ \ \$\$:= \$1\}
\end{array}
$$

Figure 4 shows the parse-tree and database-tree constructed using these rules. The curved lines describe the correspondence between the database elements and the nodes in the parse-tree. The root of the database tree (the constructed triple) corresponds to the A-vertex in the parse-tree.

The correspondence between database and parse-tree vertexes is complex:

many-1: One database element may correspond to several vertexes in the parse-tree (e.g., when \$1 and \$2 return the same value a, the value of attribute $A_{1,2}$ is a singleton set $\{a\}$ and its member corresponds to both the $B-$ and $C-$vertexes).

0-1: On the other hand, there may be database elements that do not have any corresponding vertex in the parse-tree (e.g., the value of attribute $A_{1,1}$).

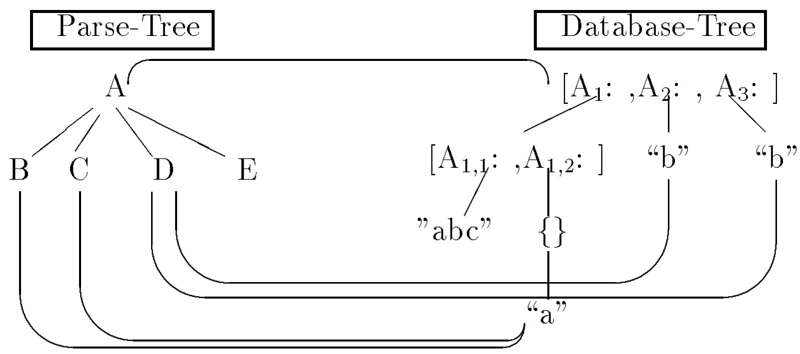


Figure 4: Parse-tree vs. Database-tree

1-many: The D-vertex has two corresponding database vertexes (attributes A_2 and A_3). While in the database level, the two attributes are in principle independent (and can be modified separately), the action requires that their values be identical.

1-0: Finally, observe that the E-vertex in the parse-tree has no corresponding vertex in the database-tree since \$4 does not occur in the action.

The idea underlying the notion of correspondence is very procedural. We start from the leaves of the two trees, and follow the construction of actions, remembering as much as possible the correspondences between vertexes in lower levels of the tree and using this information to determine correspondence between nodes at higher levels. A precise description of these correspondences is given in Appendix A for F empty. When F is not empty, one also has to specify for each $f \in F$ how the correspondence is affected.

Now, let us consider again the idea of propagating an update by replacing the subparse-tree(s) corresponding to the updated element[4] e, by the new subparse-tree constructed by unparsing the new value e', (assuming now that we know what are the vertexes corresponding to e in the parse-tree). Unfortunately it is not, in general, sufficient to view updates in a local manner. This is illustrated in the following example. Consider the schema

$$\begin{array}{lcll} S & \rightarrow & S_1\ T & \{\$\$:= [A : \$1, B : \$2]\} \\ & | & S_2\ T & \{\$\$:= [A : \$1, B : \$2]\} \\ S_1 & \rightarrow & T & \{\$\$:= \{\$1\} \\ S_2 & \rightarrow & T\ T & \{\$\$:= \{\$1, \$2\} \\ T & \rightarrow & string & \{\$\$:= \$1\} \end{array}$$

Suppose that we have a file f corresponding to a database value $[A : \{a\}, B : b]$. The attribute A contains a singleton set, and its corresponding vertex in the parse-tree is labeled by S_1. Now assume that the attribute A is updated and a new element is inserted into the set. Note that the new set can not be unparsed w.r.t S_1 (since S_1 only builds singleton sets). Thus the update cannot be propagated by replacing the subtree rooted at S_1 by a new subtree. The updated propagation requires a more involved modification. The parse-tree above S_1 has to be modified since we need to replace the use

[4]Note that even if e does not have a corresponding vertex, we may still try to minimize the unparsing by considering the first ancestor of e that does have a corresponding parse-tree vertex.

of the first rule of S by the use of the second. For most practical purposes, this kind of structuring schemas may be avoided. It is indeed natural to assume that an update only has local effects. This is captured by the locality property described below.

Definition 5.1 A structuring schema S is *local* iff for every file f, every parse tree p of f (using the schema S), its corresponding database-tree db, and every vertex e in db, the following hold:

(L-1) e has at least one corresponding vertex in the parse-tree p, and all its corresponding vertexes are labeled with the same nonterminal T.

(L-2) for each database db' obtained by replacing in db the subtree e by some e' of the same type, if db' is legal then $unparse(T, e')$ succeeds. (Where *legal* means that db' is the image of some file f' using the schema S);

(L-3) and if $unparse(T, e')$ succeeds, then the parse-tree p' obtained by replacing in p all the subtrees corresponding to e by the subtree $unparse(T, e')$, constructs the database db' obtained by replacing e in db by e'.

Observe that from the above definition it follows that for local structuring schemas, the database obtained by replacing e by e' is legal iff $unparse(T, e')$ succeeds. If a structuring schema is local, one can propagate an update from the database to the file as follows:

> **Algorithm Local-Update:**
> Let $replace(e, e')$ be an update. Computes $unparse(T, e')$. If the unparsing fails, rejects the update. Else, replace in the parse-tree p all the subtrees corresponding to e by new subtrees corresponding to e'.

Unfortunately, it turns out that:

Theorem 5.2 Locality is undecidable for SS's even for $F = \emptyset$.

Proof: This is proven by reduction from the problem of testing if the intersection of two context free languages is empty. Given two context free grammars G_1 and G_2, $L(G_1) \cap L(G_2) = \emptyset$ iff the following schema $S(G_1, G_2)$ has the locality property.

We assume w.l.o.g. that the nonterminals of the two grammars are disjoint. Let S_1, S_2 be the start symbols of G_1, G_2 respectively. The SS $S(G_1, G_2)$ contains the following rules:
For each $i = 1, 2$,

$$\begin{array}{lll} S & \rightarrow & S_1 S_1' \quad \$\$:= [A_1 : \$1, A_2 : \$2] \\ S_1' & \rightarrow & \epsilon \qquad\;\; \$\$:= true \\ S & \rightarrow & S_2 S_2' \quad \$\$:= [A_1 : \$1, A_2 : \$2] \\ S_2' & \rightarrow & \epsilon \qquad\;\; \$\$:= false. \end{array}$$

where S returns a tuple with a string in the first field and a boolean in the second. For each of the rules in G_1 and G_2, $S(G_1, G_2)$ contains a corresponding rule with an action that simply builds the string derived by that rule. For instance, if $V \rightarrow X\ a\ Y$ is in one of the G_i's then $S(G_1, G_2)$ contains a corresponding rule

$$V \quad \rightarrow \quad X\ a\ Y \quad \$\$:= \$1 || \text{“}a\text{”} || \$3$$

Observe that $S(G_1, G_2)$ maps a word w in $L(G_1)$ to $[A_1 : w, A_2 : true]$ and a word w in $L(G_2)$ to $[A_1 : w, A_2 : false]$. Consider the parse-tree for a word w in $L(G_1)$. Then the corresponding data value is $[A_1 : w, A_2 : true]$. The root of the parse-tree corresponds to the root of the data value, and the left and right children correspond to the first and second attributes of the tuple. There is no other correspondence.

Let us see now whether $S(G_1, G_2)$ is local. Clearly, (L-1) is always satisfied. First, suppose that the two languages are not disjoint and let w be in $L(G_1) \cap L(G_2)$. Suppose the parse-tree is using G_1. Then the database value is $[A_1 : w, A_2 : true]$. Consider the update to the second attribute that modifies $[A_1 : w, A_2 : true]$ to $[A_1 : w, A_2 : false]$. Then this update is legal since w is in $L(G_2)$ but $unparse(S'_1, false)$ fails since in $S(G_1, G_2)$, S'_1 only builds $true$. Hence, (L-2) is violated. Thus, $S(G_1, G_2)$ is not local.

Now, suppose that the two languages are disjoint. We already saw that (L-1) is satisfied. Now consider (L-2). Suppose the file is $w \in L(G_1)$. Thus the database value is $[A_1 : w, A_2 : true]$. Let u be a legal update. Three cases occur:

1. u modifies only w to w'. Since the update is legal, w' is in $L(G_1)$ and $unparse(S_1, w')$ succeeds.

2. u modifies $true$ to $false$. Since w is in $L(G_1)$, w is not in $L(G_2)$, a contradiction. This case cannot occur.

3. u modifies both w to w' and $true$ to $false$. Since the update is legal, w is in $L(G_2)$. Thus, $unparse(S, [w', false])$ succeeds.

Hence (L-2) is satisfied. One can show similarly that (L-3) is also satisfied. Thus $S(G_1, G_2)$ is local. □

To conclude this section, we describe a class of structuring schemas that have the locality property. Although this seems a rather limited class, most examples of structuring schemas that we considered could be transformed very easily into an equivalent local structuring schema. (The transformation essentially involved introducing new nonterminals to guarantee (A-1) below.)

Definition 5.3 A schema is in class *Local-1* if:

(A-1) at most one occurrence of one constructor or one constant is used in each action, and $F = \emptyset$;

(A-2) for each two distinct nonterminals, T_1, T_2, their associated types are distinct;

(A-3) for each rule, each $\$i$ occurs at most once in the action.

Theorem 5.4 Each *Local-1* structuring schema is local.

Proof: Let S be a SS satisfying (A-1),(A-2), and (A-3). We show that it is local.

First consider (L-1). In the database view by S, every vertex e has at least one corresponding node in the parse-tree. This comes from (A-1) and by inspection of the algorithm for constructing correspondences. In the first phase, all vertices in the database tree have corresponding vertices in the parse-tree. The other two phases do not introduce new vertices and the correspondences of the old vertices remain.

Now consider a vertex v in the database tree that corresponds to two two vertices in the parse tree labeled by nonterminals T, T' respectively. Note that in the algorithm computing the correspondence nodes are merged only when the associated data values are identical, hence obviously have the same type. Thus the types of the values associated to T and T' must the same, i.e., the type of v. By (A-2), T and T' coincide. Thus (L-1) holds.

We next consider (L-2). Let db' be some legal database obtained from some database db by replacing some subtree e by some e' of the same type. Since db' is legal, there is a parse-tree p' that builds db'. By (A-1), the vertex e' corresponds to some nonterminal in the parse-tree of db'. By (A-2), this nonterminal can only be the nonterminal T corresponding to e. Thus $unparse(T, e')$ succeeds.

To conclude the proof we show that (L-3) holds as well. For that we use the correspondence algorithm. Let $n_1 \ldots, n_l$ be the nodes of the parse tree p corresponding to the updated element e, and let p' be the updated parse tree. Consider the first phases of the correspondence algorithm, when applied on the parse trees p and p'. The database trees db_1, db'_1 built in this phase represent the data terms of db and db' resp. They are the same everywhere except for the subtrees rooted in the nodes $m_1 \ldots, m_k$ to which $n_1 \ldots, n_l$ correspond (at this phase). In the case of db_1 they describe the term of e, and in the case of db'_1 that of e'.

To turn these data terms into database values db and db' one can apply standard term rewriting to get rid of the $cons$, $\cup$, and $||$ operators, and of duplicates in sets. In particular, this can be done by applying the rewritings in the second and third phases of the correspondence alg.

Consider first the set of rewritings performed by the algorithm when applied to db_1. One can show that all these rewritings, except for those performed on subtrees rooted at $m_1, \ldots, m_k$ can be performed on db'_1. The proof is based on the fact that each $\$i$ appears in an action at most once. This implies that each node n_i corresponds to exactly one node m_i. For e to correspond to $n_1 \ldots, n_l$, all these m_i nodes must be merged into one node (the node of e), and no other node is merged with them. Since db_1 and db'_1 are the same everywhere except for the subtrees rooted at $m_1, \ldots, m_l$, all the merges performed on db_1 can be performed on db'_1.

The tree obtained by applying those rewritings on db'_1 is the same as the tree of db except that subtree representing e is replaced by a subtree representing e'. This proves the claim. □

The *Local-Update* algorithm can be used to propagate updates in Local-1 structuring schemas. From an implementation viewpoint, we can assume that the actions in the structuring schema are modified so that an auxiliary data structure giving for each vertex in the database-tree, its corresponding vertexes in the parse-tree, is constructed while parsing the file. It is important to observe that the data structure used to described the correspondences can be maintained, i.e., modified to reflect database updates.

We conclude this section with one remark on the relaxation of the constraints on Local-1 schemas and one on some standard updates.

Remark 5.5 The most critical limitation seems to be (A-2). For instance, one may find it useful to use in different parts of a document two sequences of strings with different syntax (e.g., one sequence with the symbol "," between the strings, and another with the word "and" separating them). To support this, we may need two distinct nonterminals constructing lists of strings, a violation of (A-2). In fact, if the two nonterminals appear in strictly "separated" parts of the document (and their corresponding lists belong to "separate" parts of the database) the restriction can be relaxed, while still preserving locality. It is easy to come up with a formal definition of *separated*.

Now consider relaxing (A-1). First suppose that more than one constructors is used in a rule. This may result in database values not corresponding to any parse-tree vertex. As a consequence, we may

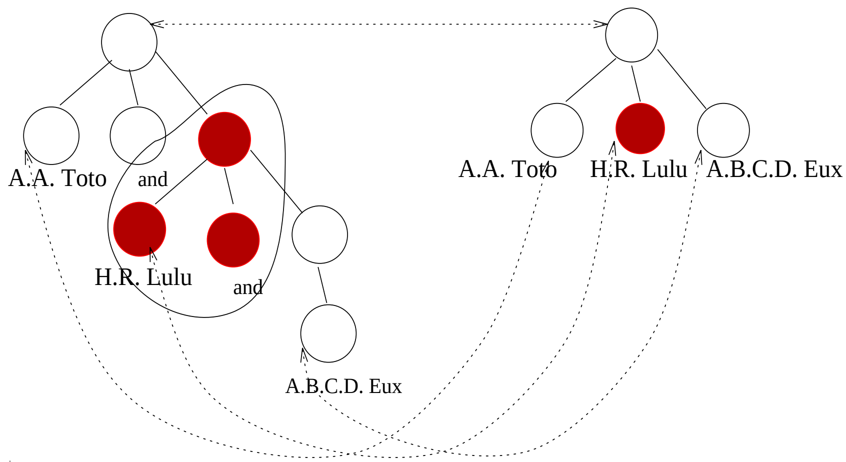


Figure 5: A list of Authors

have to unparse more than necessary and be forced to reconsider (A-2) in non-trivial ways. Relaxing the condition $F = \emptyset$ is also somewhat tricky, as has been illustrated in the previous section on unparsing.

Relaxing (A-3) may result in having parse-tree vertexes that match more than one database elements, and thus the modified file may yield a value that differs from the modified database (e.g., if we change the value of A_2 independently of A_3 in the example illustrated by Figure 4). To prevent that we have to parse the updated file, and make sure that the resulting database is indeed the updated one (if not, the update is rejected). □

Remark 5.6 Consider standard update operations, such as modifying an attribute of a tuple, or adding/deleting members of a set. A modification of a tuple attribute leads, in our framework, to unparsing the new value of the attribute, and "plugging" the resulting subparse-tree in the appropriate location. Clearly, this is optimal.

Now consider a deletion from a set. For instance, consider the parse-tree and database-tree in Figure 5. Assume that we want to delete the author H.R. Lulu from the collection. Observe that deleting Lulu from the database entails more than just deleting the corresponding vertex in the parse-tree. The deletion can be viewed as an update to the whole set, and the update can be propagated by unparsing the updated set. This, however, seems to require too much work – the whole set is unparsed, although the update involves only a single element. This potentially may also cause unnecessary loss of information. (A solution to the loss of information is mentioned in conclusion: guided unparsing). A similar situation occurs when considering insertion in a collection.

It turns out that if we work with an SGML-like grammars, we can avoid unparsing the whole collection. For instance, consider the following SGML rule:

$$\langle Authors \rangle \;\; \rightarrow \;\; \langle Author \rangle^{*} \quad \{\$\$:= list(\$i)\}$$

It constructs parse-trees having the same form as the database tree on Figure 5. Thus, deleting an element is just deleting its corresponding subtree, and similarly for insertion. □

6 Conclusion

We studied a general framework for queries and updates specified logically on a database, to a file that actually stores the data (in a structured manner). For queries, we presented an optimization technique. For updates, we provided general techniques for unparsing database values, and studied optimization techniques. Most importantly, from a practical viewpoint, we presented of a large class of schemas where unparsing can be performed locally.

Even with locality, update propagation may introduce a number of unnecessary changes to the file. For instance, when a set is modified (by insertion/deletion), we may need to unparse the entire set and possibly modify elements that were not explicitly involved in the update. It is possible to use a "guided unparsing" technique to reduce the difference between the original parse-tree and the updated one. The idea is to use the original parse-tree to guide the unparsing of the updated database portion thereby minimizing the changes.

To simplify the presentation we considered here a rather simple type system, e.g., without union types. (The impact of objects has been considered.) It would be interesting to study the impact of other modeling features (and in particular, union of types) on the technique.

To conclude, we examine two issues:

Who came first: the hen or the egg

An issue is who came first: the database or the file. In many cases, one knows in advance that data will be accessed both from a database and from a file system. The mappings from files to databases (via structuring schemas) and from databases to files (as in Appendix B) are designed at an early stage, indeed, at the same time the database and the file structures are designed. More research is therefore necessary to understand that process. We believe that much more can be achieved if they are both developed concurrently instead of first designing one, then designing the mapping to the other, and *finally only* trying to invert that mapping.

Views in an heterogeneous environment

Structuring schemas may form the basis of a view mechanism in an heterogeneous context. This is illustrated by Figure 6. It is reasonable to assume that a database model is delivered together with some grammar describing a possible representation of the database in a file. Indeed, database systems often provide gateways to file systems under the form of data loaders and sophisticated report writer (e.g., [29]). Furthermore, it is now becoming customary that the input or output files follow some structuring standard such as SGML (e.g., [29]).

Now suppose that we have a database db in some first model (e.g., relational) and want to provide access to this data through a view $View$ in another model (e.g., object-oriented). For this, we need the structuring schemas for the two models. Let us assume first that the two schemas use the same grammar and differ only in their semantic actions. Now, suppose that we want to propagate an update from the view to the database. The real database (e.g. the relational) is unparsed entirely using its structuring schema. Now we are in the situation of the paper, we have a file (in fact we already have the parse-tree) and a database view of it. The update on the view is propagated to the parse-tree using the structuring schema of the view. It remains to propagate the update to the real database. This direction of update propagation (from file to database) is rather straightforward and was not considered here; it can be performed using standard techniques from incremental parsing.

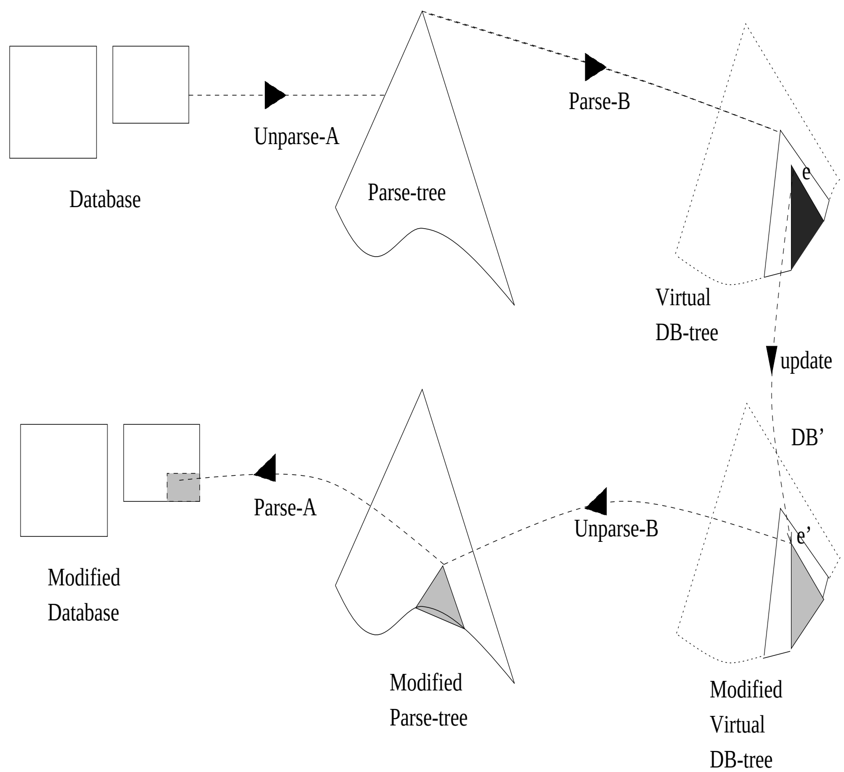


Figure 6: Views for Heterogeneous Databases

Finally, assume that the grammars for the two schemas are different. In such cases, a translation phase must be added.

Acknowledgment: We thank Eric Angel, Vassilis Christophides, Peter Buneman, Sophie Gamerman, Laurent Herr, Bernard Lang, Jean-Claude Mamou, Michel Scholl for discussions on the topic.

Appendix A - Computing Correspondences (for $F = \emptyset$)

We want to relate vertices in the database tree to vertices in the parse tree. This will be achieved by constructing a graph G consisting of (1) two trees - a parse tree and a database tree, and (2) correspondence edges connecting nodes in the two trees.

The graph is constructed in 3 phases. We start with a very rough description of the database (and the correspondence to the parse tree nodes) and then refine it. At the first phase, we take the parse tree and build a preliminary database tree describing the data term constructed by the actions of the structuring schema. To turn this data term tree into a tree representing the actual value of the database, we rewrite the tree to get rid of operators like *cons*, $\cup$, and $||$, and of duplicates in sets. The removal of the *cons*, $\cup$ and $||$ operators is done in the second phase. At the end of this phase we are left with a tree where the only difference with a "concrete" database tree is that set vertices may have two children that are identical. The last third phase merges identical sub-trees of set vertices.

At each step we update the correspondence relationship between the nodes in the database tree and the nodes in the parse tree to reflect the rewriting being performed.

Phase 1: Initializing the correspondence

Consider a parse-tree with root T, and children $T_1, ..., T_n$. T_i, $i = 1 \ldots n$ is either a non-constant tokens or a nonterminal. Assume that the the parse tree of T_i build the database value v_i. The (preliminary) database tree and correspondence edges are built inductively.

The basis of the induction: For each non-constant token T_i with associated atomic value v_i, G_i consists in the graph with: a parse tree containing one node labeled with T_i, a database tree containing one node labeled with v_i, and correspondence edge between these two vertices.

The induction step: let $G_1, ..., G_n$ be the correspondence graphs for the parse-trees $T_1, ..., T_n$ and the database values $v_1, ..., v_n$ resp. Let r be the rule that is used at the root of the parse-tree T, and let t be the term assigned to \$\$ in the action part of r. We construct a graph G such that

1. G contains the parse-tree rooted at T;

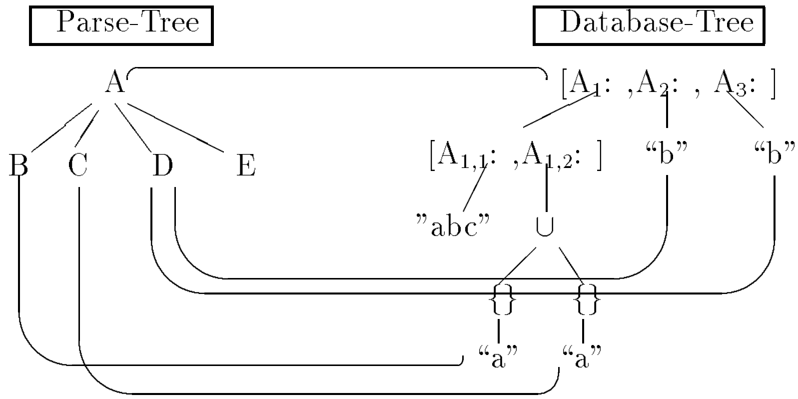
2. the database tree is obtained by taking a tree representing the term t and replacing the leaves $\$1, ..., \$i$ by the database trees from $G_1, ..., G_n$;

3. the correspondence edges are those from $G_1, ..., G_n$; and an additional edge between the root of the parse-tree and the root of the database-tree.

For example, consider the structuring schema

$$\begin{array}{lcll}
A & \rightarrow & B``,"C``,"D``,"E & \{ \quad \$\$ := [A_1 : [A_{1,1} : ``abc", A_{1,2} : \{\$1\} \cup \{\$2\}], A_2 : \$3, A_3 : \$3] \quad \} \\
B & \rightarrow & \#String & \{ \quad \$\$:= \$1\} \\
C & \rightarrow & \#String & \{ \quad \$\$:= \$1\} \\
D & \rightarrow & \#String & \{ \quad \$\$:= \$1\} \\
E & \rightarrow & \#String & \{ \quad \$\$:= \$1\}
\end{array}$$

Assume that the parsers of B, C, and D return the strings "a", "a", and "b" resp. The graph constructed by the first phase is the following.



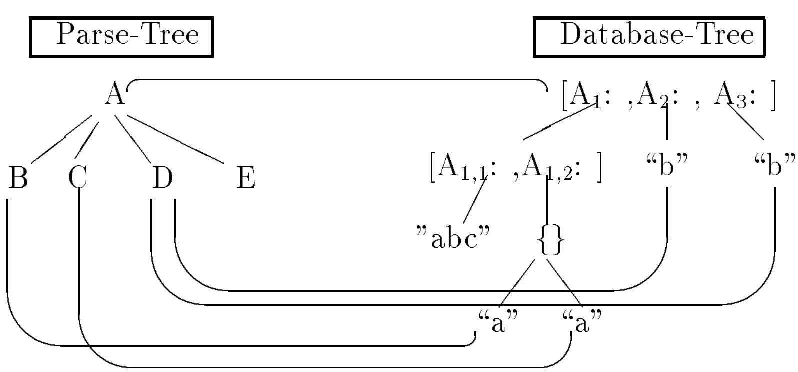
Phase 2: Eliminating the operations *cons*, $\cup$, and $||$

We first show the elimination process for $||$ (string concatenation). We process the tree bottom up. At each step we pick a database vertex v labeled by $||$, and where all the children are strings (e.g., leaves). The children vertices of v are removed from the tree (possibly resulting in eliminating correspondence edges for these vertices). Now v is a leaf. The label ($||$) of v is replaced by a string which is the concatenation of the strings labeling the deleted children. By induction, the graph resulting at the end of this process has no $||$ vertex.

Now consider the *cons* vertices. Again, we process the tree bottom up. At each step we pick a database vertex v labeled by *cons*, but where where none of v's children is labeled with *cons*. Let v_1, v_1' be the left and right children of v, respectively. v_1' must be labeled with a collection constructor, say it is a set vertex with children $v_2, ..., v_m$. (The other constructors are treated similarly.) We remove v_1' from the tree (again this may result in deletion of correspondence edges), make $v_1, ..., v_m$ the children of v, and replace v's label by the collection constructor. Again, by induction, the resulting graph has no *cons* vertex.

Finally, the $\cup$ vertices are treated similarly. (In this case both v's children are removed and their children become v's children.)

For example, after this phase the graph of the above structuring schema is the following.



The nodes being deleted from the database tree represent intermediate computations. We are interested only in the final database. The nodes and their correspondence edges are thus irrelevant.

Phase 3: Dealing with set idempotence.

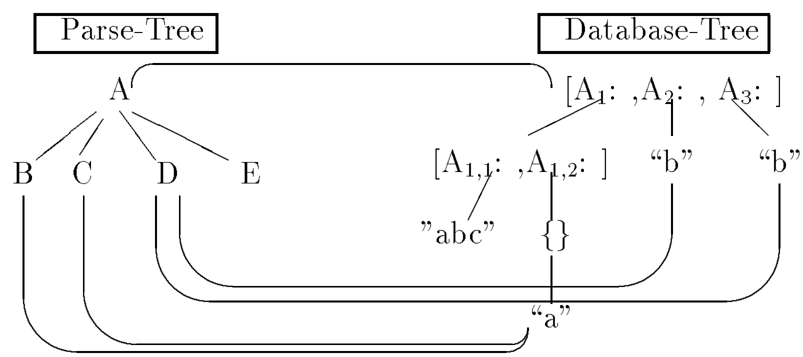
In the "database tree" resulting from Phase 2, a set vertex may have two vertices denoting the same database value. To get a "concrete" database value we eliminate the duplicates. We treat those bottom up as follows.

At each step we pick a node v having two children v_1, v_2 representing the same database value, but where none of v's descendant has this property (i.e. no descendant of v is a set vertex with two children representing the same value). Then, we merge vertex v_2 into v_1 as follows:

1. if v_1, v_2 are atomic values, we remove v_2.
2. if v_1, v_2 are tuples, lists or sets, v_2 is removed and each component of v_2 is merged into the corresponding component of v_1. (For sets, this is possible since we chose a "minimal" v.)
3. if v_1, v_2 are bags, we first choose one of the possibly many matches between elements in v_1 and elements in v_2. Then we remove v_2 and merge each component of v_2 into the corresponding component of v_1.

In each case, when a vertex v_2' is merged into a vertex v_1', v_1' inherits all the correspondences of v_2' (besides its own), i.e., if the correspondence edge (T, v_2') was in the graph (and disappears since v_2' is removed), then (T, v_1') is inserted to replace (T, v_2').

For example, at the end of this phase the above correspondence graph becomes



The result from Phase 3 is a graph describing (1) the parse-tree of T, (2) a database tree representing the value returned by the parser, and (3) the correspondence relation between the two.

The following property is easily verified:

> Let G be a graph relating a parse-tree to some data value, constructed by the above algorithm. Let (T, v) be a correspondence edge in G. If the parse-tree rooted at T is the result of parsing a string w, then $w \leadsto_T v$.

Appendix B - From Databases to Files

Structuring schemas define a mapping from a file to a database. As shown in the previous section, in some cases this mapping can be inverted, and can be used to unparse a database and generate a file. Note that in this case, the mapping from a database to files is not specified directly but is rather a derivative of the mapping from files to databases. The possibility of specifying the mapping directly is interesting. Indeed, the mapping from databases to files is at the core of database report generation, and that together with the mapping from files to databases, it is often nowadays the main support for inter-operating a database and other applications (data being exchanged via files).

We do not consider here the direct mapping from databases to files in detail. But for the sake of completeness, we briefly address the issue.

We assume that we have structured data in a database and want to produce a string (with an associated grammar) containing some of the database information. To encode data on a string, we need (i) to specify the object or value u to encode; (ii) the encodings of its components (if any); and (iii) the encoding of u (eventually based on that of its components). This yields the following cases:

- For bases types (**int, real, string**, etc.), we only need to specify the format of the encoding.
- For tuples, we have to specify the encoding of components and that of the tuple in terms of the components.
- For **collection types**, we need to use some iteration, and for instance, the structural induction operator of [8]. The structural induction φ on a set X with parameters f, g, h is defined by:

  $$\begin{array}{lcl} \varphi(f,g,h)(\emptyset) & = & f \\ \varphi(f,g,h)(\{x\} \cup X) & = & h(g(x), \varphi(X)). \end{array}$$

  For instance, to encode a set X with members of type T, we can use: (a) a function g encoding T elements, (b) the function f returning the empty string; and (c) h defined by: for each u, v,

  $$h(u,v) = concat(u, \text{ ``,'' }, v).$$

  Observe that, in this case, the resulting string depends on the order of the elements that is chosen. In that sense, the operation is non-deterministic. (This is not surprising since, in general, structural induction is non-deterministic.)
- The most interesting issue is that of encoding of objects. There are many possibilities to encode an object identifier, depending of the database model: logical oid's (if supported by the system), external oid (generated for the "externalization" of the object), or using a query that somewhat qualifies the object. One is also lead to encode the state of the object together with some surrogate for this object. In presence of cyclic data, it is then necessary to avoid entering into loops. These choices have dramatic impact on the performances of the system and indeed even on its semantics.

Some important issues also arise:

1. The choice of the specific encoding, and consequently the structure of the resulting file, depends on the application. In some cases the resulting file should obey a fixed grammar (e.g., HTML). In other cases, the grammar may be determined by the particular database being processed. The grammar of the resulting file may also need to have a specific format (Yacc, SGML, etc.).

2. Another issue is that for large and/or complex data, one may not want to generate one large file, but rather to build several files - a *directory* with possibly references between the various files. The granularity of the files is thus an important issue.

3. Another issue is the selection of the portion of the database that is represented in the file and its consistency (no dangling references).

4. The last issue is architectural. Typically, the task of generating a file may be split between three partners: (i) the database, via a view definition module that may restructure and prepare data, (ii) the report writer that generates the file, and (iii) eventually a tool that does some post processing on the file. This may be some simple stream editor (e.g., sed) or a more sophisticated tool that may involve re-parsing of the file (e.g. SGML editor). The relationship and cooperation mode of these partners needs to be investigated.

References

[1] S. Abiteboul and C. Beeri. On the manipulation of complex objects. Technical report, INRIA and Hebrew Univ., 1988. To appear in VLDB Journal.

[2] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. VLDB*, 1993.

[3] S. Abiteboul, S. Cluet, and T. Milo. A database interface for files update. In *Proc. SIGMOD*, 1995.

[4] S. Abiteboul and P. Kanellakis. Identity as a query language primitive. In *Proc. SIGMOD, Portland, Oregon*, 1989.

[5] D. Barbara, H. Garcia-Molia, and S. Mehrota. The gold mailer. In *IEEE Data Eng.*, pages 92–99, 1993.

[6] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. ICDT, Paris, France*, 1990.

[7] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The Datacycle architecture. *cacm*, 35(12):71–81, dec 1992.

[8] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Conf. on Database Programming Languages, DBPL*, 1991.

[9] R. Cattell, editor. *The Object Database Standard: ODMG 93.* Morgan Kaufmann, 1993.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM Sigmod, Minneapolis*, 1994.

[11] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. Sigmod, San Diego, USA*, 1992.

[12] M. Consens and T. Milo. Optimizing queries on files. In *Proc. ACM Sigmod, Minneapolis*, 1994.

[13] Open Text Corporation. *PAT Reference Manual and Tutorial*, 1993.

[14] S.S. Cosmadakis and C.H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[15] U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(3):381–416, 1982.

[16] O. Deux et al. The story of o2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1989.

[17] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *CACM*, 35(12), December 1992.

[18] A. Keller. Algorithms for translating view updates to database updates for views involving selections, projections and joins. In *Proc. ACM PODS*, pages 154–163, 1985.

[19] L. Lamport. *LaTex Manual.* 1985.

[20] A. Paepcke. An object oriented view onto public hetrogeneous text databases. In *IEEE Data Eng.*, page 484, 1993.

[21] T.W. Reps and T. Teitelboum. *The Synthesizer Generator, A system for Constructing language based editors.* Springer-Verlag, 1989.

[22] M. F. Schwartz. Internet resource discovery at the university of colorado. *IEEE Computer Networking*, 26(9), September 1993.

[23] G. Shaw and S. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proc. DOOD, Kyoto, Japan*, 1989.

[24] G. Shaw and S. Zdonik. An object-oriented query algebra. In *Proc. DBPL, Salishan Lodge, Oregon*, 1989.

[25] K. Shoens, A. Luniewski, P. Schwartz, J. Stamos, and J . Thomas. The rofus system: Information organization for semi-structureddata. In *Proc. of the 19th Int. conf. on Very Large Databases, VLDB93*, pages 97–107, 1993.

[26] N.C. Shu, B.C. Housel, R.W. Taylor, S.P. Ghosh, and V.Y. Lum. Express: a data extraction, processing, and restructuring system. *ACM Transactions on Database Systems*, 2(2), June 1977.

[27] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT, Cambridge*, 1994.

[28] D. Straube and T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. Technical report, Department of computing science, university of Alberta, Edmonton, Alberta, Canada, 1990.

[29] O2 Technology. *The O2 User's Manual Version 3.3*, March 1992.

[30] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science, Vol B*, page 167, 1990.

[31] J.D. Ullman. *Principles of Database and Knowledge Base Systems: Volume I and II.* Computer Science Press, 1988.