A Database Interface for Files Update*

Serge Abiteboul Sophie Cluet† Tova Milo ‡

1 Introduction

Database systems are concerned with structured data. Unfortunately, data is still often available in an unstructured manner (e.g., in files) even when it does have a strong internal structure (e.g., electronic documents or programs). In a previous paper [2], we focussed on the use of high-level query languages to access such files and developed optimization techniques to do so. In this paper, we consider how structured data stored in files can be updated using database update languages.

The interest of using database languages to manipulate files is twofold. First, it opens database systems to *external* data. This concerns data residing in files (and used by non-database applications) or data transiting on communication channels and possibly coming from other databases [2]. Secondly, it provides high level query/update facilities to systems that usually rely on very primitive linguistic support. (See [6] for recent works in this direction). Similar motivations appear in [4, 5, 7, 8, 11, 12, 13, 14, 15, 17, 19, 20, 21]

In a previous paper, we introduced the notion of structuring schemas as a mean of providing a database view on structured data residing in a file. A structuring schema consists of a grammar together with semantic actions (in a database language). We also showed how queries *on files* expressed in a high-level query language (O_2-SQL [3]) could be evaluated efficiently using variations of standard database optimization techniques. The problem of update was mentioned there but remained largely unexplored. This is the topic of the present paper.

We argue that updates on files can be expressed conveniently using high-level database update languages that work on the database view of the file. The key problem is how to propagate an update specified on the database (here a view) to the file (here the physical storage). As a first step, we propose a *naive* way of update propagation: the database view of the file is materialized; the update is performed on the database; 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 [9, 10, 16, 23].) The technique relies on the existence of an inverse mapping from the database to the file. We show that the existence of such an inverse mapping results from the use of restricted structuring schemas.

---

*Partially supported by Esprit Projects Fide2 and GoodStep.

†I.N.R.I.A., 78153 Le Chesnay Cedex, France, {Serge.Abiteboul,Sophie.Cluet}@inria.fr, Tel: (33)1-39635679, Fax: (33)1-39635330

‡Computer Systems Research Institute, U. of Toronto, Toronto, Canada M5S 1A1, milo@db.toronto.edu, Tel: (1)416-9783820, Fax: (1)416-9781676

The *naive* technique presents two major drawbacks. It is inefficient: it entails intense data construction and unparsing, most of which dealing with data not involved in the update. It may result in information loss: information in the file, that is not recorded in the database, may be lost in the process. The major contribution of this paper is a combination of techniques that allows to minimize both the data construction and the unparsing work. First, we briefly show how optimization techniques from [2] can be used to focus on the relevant portion of the database and to avoid constructing the entire database. Then we show that for a class of structuring schemas satisfying a *locality* condition, it is possible to carefully circumscribe the unparsing.

Some of the results in the paper are negative. They should not come as a surprise since we are dealing with complex theoretical foundations: language theory (for parsing and unparsing), and first-order logic (for database languages). 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 instance, all "real" examples of structuring schemas that we examined are *local*.)

The paper is organized as follows. In the next section, we introduce the update problem and recall the definitions of structuring schemas. Section 3 presents a naive technique for update propagation and the unparsing technique on which it relies. Section 4 introduces a locality condition, and presents a more efficient technique for propagating updates in structuring schemas satisfying this condition. The last section is a conclusion.

2 Getting Started

We are interested in files that have strong inner structure (e.g., electronic documents, programs, SGML files). Our goal is to use the inner structure of files for providing high level and efficient update interface to the data in the file. Bibliography files constitute an example of semi-structured data with which all researchers are well acquainted. Consider for example a bibliography file in the BibTex format [18]. An entry in the file is a string of the following form:

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

Viewing this information as a database provides both modeling and processing benefits. In [2, 7] we studied the use of database query languages to manipulated files. In this paper, we study updates. We assume in the following basic knowledge about object oriented databases. A bibliography database may contain a class of *References*, with reference objects that have attributes like *Key*, *Authors*, *Title*, etc. Now, 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:

Update r1.Notes := concat(r1.Notes,“ A later version of this paper appears in ”, r2.Key)
From r1 in References, r2 in References
Where r1.Title = r2.Title and r1.Authors = r2.Authors and r1.Year < r2.Year

Note that we use here some rather trivial criteria to detect earlier versions. 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?

The above update was specified at the database level. To actually modify the file, the update must be translated to an update on the file.

To understand the relationship between updates at the database level and file modifications, we first have to precisely define the mapping between portions of the file and database elements. *Structuring schema* are a tool for defining such mappings. Structuring schemas where informally introduced in [2]. To give some intuition on the technique we first briefly recall this informal presentation. We next present a formal definition.

2.1 Structuring Schemas

Structuring schemas offer a database view on structured data residing in files. A structuring schema (SS for short) consists of a context-free grammar annotated with one *semantic action* per rule, and one type per non-terminal. The grammar describes the structure of the file (we assume that the grammar is reduced). The annotation 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 subwords derived from $A_1, \ldots, A_n$.

Consider for example the BibTex file discussed above. A portion of the structuring schema for BibTex files, (i.e., some type definitions, and some grammar rules with their associated actions), is presented below:

type $\langle Reference\rangle = \mathbf{tuple}(Key : string, Authors : ...)$
type $\langle Ref\_Set\rangle = \mathbf{set}(\mathbf{tuple}(Key : string, Authors : ...,))$

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

Non-terminals appear between brackets (e.g., $\langle Reference\rangle$), constant tokens appear between quotes, (e.g., “$@Inproceedings\{$”), and base tokens are preceded by the # symbol (e.g., $\#String$). Base tokens correspond to database basic types and their appropriated typed value is returned by the lexical analysis (which we are not considering here). For the action part of the rules we use a Yacc-like notation. 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 non-terminal or base token in the right hand-side of the rule. According to the above structuring schema, the database view of a BibTex file is a set containing one element per bibliographical reference. A reference is represented by a tuple with one attribute per BibTex field.

We now formally describe the types and actions that we consider in this paper. The database types are defined by the following abstract syntax:

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

SSs were first introduced in [2] with a more general notion of types. To simplify the presentation, we mostly ignore here some aspects considered in the original framework: union of types, objects/classes/inheritance. We briefly consider them in section 5.

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 of functions F. 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]. Standard issues such as type checking of actions will not be considered here.

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

1. each $\$i$ is an action, each database constant is an action.
2. with standard 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 collection (set,list,bag) 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 will 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.

As explained in [2], a SS 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 word (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* become undecidable, e.g: given a rule r, is there a successful parsing of a word (taking into account the order of rules) that uses r. [1]

[1] This can be proved by reduction from the undecidability of testing containment of context-free languages.

We therefore concentrate on a relation that ignores the order of rules. Given a SS S, a nonterminal T_0, a file f and a value v, $f \leadsto_{S,T_0} v$, indicates that there exists a parse-tree of word f rooted at T_0 and *yielding* the database value v. When S is understood, $f \leadsto_{S,T} v$ is simply written $f \leadsto_T v$. For T_0 the start symbol, $f \leadsto_{S,T_0} v$ is sometimes written $f \leadsto_S v$. When both S is understood and the start symbol is meant, we sometimes use $f \leadsto v$. The value v s.t. $f \leadsto v$ is called the *database view* of file f. We next explain how updates specified on the database view of the file are propagated to the file.

Remark 2.1 In this paper, we are interested in the effects of updates on the database, and in the propagation of these effects to the file. We are not concerned by the language used to specify updates. (Any SQL-like language could be considered.) To simplify, we first assume that the database instance can be viewed as a tree, and elements in the database as subtrees of this tree. 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. In section 5 we briefly consider objects that turn database trees into graphs. □

3 Naive Propagation and Unparsing

We first propose a naive solution based on a technique (namely *unparsing*) that will be extensively used in the entire paper. The *naive* way of propagating an update from database to file is to perform the following three steps (See Figure 1): (i) materialization of the database (i.e., parsing of the file using the SS, and construction of the entire database); (ii) updating of the database; and (iii) unparsing of the new database to produce an 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.

3.1 Unparsing

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 $f \leadsto_{T_0} db$. We first consider a restricted case where the set of functions F used in the SS is empty, i.e. only basic constructors are used in the action part of rules. The case where F is not empty is considered next. The unparsing process uses an auxiliary notion of *matching*, and a *matching algorithm*. We start by describing the matching algorithm, and later use it in the unparsing algorithm.

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 for the $\$i$ variables occurring in t, such that $\nu(t) = v$. For instance, let $t = \{\text{“}a\text{”}||\text{“}b\text{”}||\$1, \text{“}b\text{”}||\$2\}$ and $v = \{\text{“}bcd\text{”}, \text{“}abc\text{”}\}$. The assignment $\nu(\$1) = \text{“}c\text{”}, \nu(\$2) = \text{“}cd\text{”}$ is a matching for t and v.

The notion of *matching* that we use here is rather standard. The only special thing is that the algorithm computing the matching must take into account the properties of the set, bag and list constructors (e.g., commutativity, idempotence and associativity of the set constructor). Matchings are computed by a recursive function $match(t, v)$ sketched below.

Basis: The basis of the recursion is as follows.
(1) If $t = \$i$ (for some $\$i$ variable), $match(t, v)$ succeeds and returns an assignment ν, such that $\nu(\$i) = v$ and $\nu(\$j)$ is undefined for all the variables $\$j \neq \$i$.
(2) If $t = c$ and $c \neq v$ (where c is a constant), then the matching fails. (i.e. $match(t, v)$ returns $\perp$ and halts).
(3) If $t = c$ and $c = v$ then $match(c, v)$ succeeds and returns an 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 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).

Observe that the above matching algorithm either fails or returns a set of appropriate assignments for the variables in t.

Unparsing for $F = \emptyset$

We next describe the unparsing algorithm for $F = \emptyset$. Let S be a structuring schema. We define below a recursive 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 $s \rightsquigarrow_T v$; 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. The function *unparse_rule*$_S$ uses the following functions:

- *default_string*$_S$*(T)* returns a "default" string that can be parsed with the grammar of S using T as a start symbol. Since we only consider reduced grammars, such a string can be built for every non-terminal T. (This function is used when some $\$i$ does not occur in the action and thus is not constrained by the matching.)

- For each i, *strings*[i] contains the string resulting either from the unparsing for $\$i$ or using the default value. The function *build_strings(strings, r)* uses these strings to construct a string that that can be parsed using rule r to yield the target data value.

The *unparse_rule*$_S$ function is defined as follows:

function $unparse\_rule_S(r, v, history)$ **returns** $[boolean, string]$
let t be the action of 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 ($m(\$i) <> nil$) % \$i is used in the action part
$[success, strings[i]] := unparse_S(A_i, m(i), history)$;
else % $\$i$ is not used in the action part
$strings[i] := default\_string_S(A_i)$;
}
if $success$ return $[true, build\_string(strings, r)]$;
else return $[false, ``"]$;
end

Theorem 3.1 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 depth. The correctness essentially results from the careful case analysis performed by the algorithm. □

3.2 Example

We illustrate the algorithm using the Bibtex SS presented in section 2.1. 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 $\langle Ref\_Set\rangle$:

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

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 reference set.

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 $ref_1 = [Key : TLE90, Authors : \{\text{``}A.A.\ Toto\text{''}, \text{``}H.R.\ Lulu\text{''}, \text{``}A.B.C.D.\ Eux\text{''}\} \ldots,]$. To unparse ref_1 the procedure considers the rules defining $\langle Reference\rangle$. There is only one such rule

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

When ref_1 is matched against the action part of the rule, the matching assigns "TLE90" to $\$1$, $\{\text{``}A.A.\ Toto\text{''}, \text{``}H.R.\ Lulu\text{''}, \text{``}A.B.C.D.\ Eux\text{''}\}$ 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 "$@Inproceedings\{TLE90, author = \text{``}A.A.\ Toto\ and\ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{''}, \ldots\}$"

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 the Section 4.

3.3 Unparsing for $F \neq \emptyset$

The above unparsing algorithm applies to SS where the semantic actions of rules use only basic constructors (i.e. F is empty). We next consider the case where actions can use other operations as well. It turns out that this makes the unparsing much more difficult.

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 uparsing v iff such s exists). The difficulty of the problem for non empty F is demonstrated by:

Theorem 3.1 *The unparsing problem is undecidable for structuring schemas where F contains the operations $join$ and $projection$.*

The proof is by reduction from the problem of testing if the intersection of two context-free languages is empty.

Can the unparsing algorithm be accommodated to handle some cases where F is not empty ? Consider some function f be in F. The core of the technique is to find the matchings. We are lead to match a term $f(t_1, ..., t_n)$ with some value v, and so to compute $f^{-1}(v)$. The issue is thus the existence of some inverse for f.

For instance, consider the above undecidabilty theorem. It uses $join$ and $projection$ for which inverses are not finite. This is a first cause of failure of the technique. It turns out that the unparsing algorithm (modified to handle functions in F) may not terminate even when the functions in F have recursive inverses (i.e. for each f in F and each v, $f^{-1}(v)$ is finite and computable). We illustrate this second cause of failure with an example. Consider the structuring schema:

$$\begin{array}{lll} A & \rightarrow & A\text{``.''} \quad \{\$\$:= increment(\$1)\} \\ & | & \epsilon \quad\quad \{\$\$:= 5)\} \end{array}$$

If we try to unparse value $v = 4$, we loop forever, trying first to match 4 against $increment(\$1)$ (assigning 3 to \$1), then matching 3 against $increment(\$1)$ (assigning 2 to \$1), then matching 2, 1, 0, -1, etc.

In the above example, it seems that for such functions, we have to add some condition of "bounded monotonicity" for the functions in F to avoid entering infinite loops. We do not pursue this direction in the present paper.

3.4 Information Loss and Constraints

We can make the following observations on the unparsing algorithm:

- The algorithm returns one solution although there may be other strings that would generate the same value. This is because different files may have the same database image. For example, the reference set $v = \{ref_1, \ldots, ref_n\}$ may represent a file where ref_1 is the first reference entry in the file, or where it is the k^{th} entry. Also, a file may have several occurrences of ref_1, and still be represented by v. This is a consequence of the fact that the operators used in the action part do not guarantee one-to-one mapping. The decision if a sequence of strings from the file is represented in the database 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 collection. Indeed, in the above example the information loss is caused by the commutativity and idempotance of set construction. 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.

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 $f = f'$. This is a useful property of SSs. It states that the database captures essentially **all** the information contained in the file, and thus two different files never have the same database representation.

- The unparsing process may fail after a modification accepted by the database. This happens when the update satisfies the typing restriction posed by the database schema, but the modified database is no longer a possible image of some file (according to the structuring schema). Consider the following structuring schema with one rule:

  $S \rightarrow string\{\$\$:= set(\$1)\}$

  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.

  A structuring schema is said to be *constraint-free*, if for each database *db* over the domain of the database types associated to it grammar, there exists a file f such that $f \rightsquigarrow_{T_0} db$. This is also a useful property of structuring schemas. It states that each database of the proper type is "legal", i.e, is the image of some file. In particular, this implies that every valid update on the database can be propagated to the file.

Lossless and constraint freedom are useful properties. Unfortunately, they are undecidable.

Theorem 3.2 For a SS s, the following problems are undecidable (even if $F = \emptyset$): (i) testing if S is lossless, (ii) testing if S is constraint-free.

Proof: (Sketch) To prove (i) we use reduction to the problem of deciding whether the intersection of two context-free languages is empty. (ii) is proved by reduction of the problem $L = \Sigma^*$ for context-free languages. □

Because of this undecidability, we are interested in properties that guarantee lossless and constraint freedom, or at least make the test for them decidable. We start by considering lossless. We present below an important class of lossless schemas. Note that we will have to impose severe restrictions to avoid information loss. The following observation should somewhat motivates part of the restrictions:

> Set and bag construction (used in nontrivial manner) has to be ruled out because of mathematical properties of these constructors (e.g commutativity). For instance, if a list of items in a string is represented in the database by a set, we loose the order.

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

Definition: 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.

Definition: 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) ;[2]
(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 3.3 Each SS in class LossLess-1 is lossless.

It is important to note that it is possible to check whether a schema is in this class.

Proposition 3.4 One can decide whether a SS is in class LossLess-1.

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

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

Theorem 3.5 One can decide if a structuring schema in class LossLess-1 is constraint-free.

Proof: (Sketch) The proof is by reduction to the emptiness test for (non-deterministic bottom-up) tree-automata, using the facts that tree automata are closed under boolean operations, and that emptiness test is decidable. □

4 Update Propagation

In this section, we study the optimization of the propagation of updates (specified on the database) to the file. We mostly concentrate 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 previous section provided a 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 2 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)).

A technique for minimizing the database construction when querying a file was presented in [2]. When updates are considered, things work similarly. We therefore only explain the process briefly, and will mainly concentrate on the minimization of unparsing. The key idea is that an update on the database can be performed in two steps. First, a query is invoked. The query selects the database elements that need to be updated, and computes the new

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

values for these elements. The result of the query consists of sets of pairs (e, e'), where e is a vertex in the database-tree, and e' is the new value for the vertex. Once this is computed, the updates are performed, and the old values are replaced by the new ones.

In the query phase, we apply the optimization technique of [2] to avoid constructing the whole database. (This is achieved by using variants of standard optimization techniques such as "pushing selections").

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 would be tempted to try to find the subparse-tree(s) corresponding to e and replace it(them) by the result of the unparsing of e'. There are several difficulties in following this approach. First we have to define what "a subtree that corresponds to a database element e" means. Second, 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.

In particular, 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. Consider for example the following rule of a structuring schema:

$$A \quad \rightarrow \quad BCDE \quad \{ \quad \$\$:= [a1 : [a1_1 : \text{“}abc\text{”}, a1_2 : \{\$1, \$2\}], a2 : \$3, a3 : \$3] \quad \}$$

Figure 3 shows a possible occurrence of a parse-tree and a database-tree constructed using this rule. 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., if \$1 and \$2 return the same value v_1, then the value of attribute $a1_2$ is a singleton set and its member corresponds to both $B-$ 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 $a1$).

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. In particular, it is possible to modify the parsing algorithm to have it compute such correspondences. This is rather straightforward with $F = \emptyset$. When F is not empty, one also has to specify for each $f \in F$ how the correspondence is affected. Due to space limitations, we do not present here the full algorithm and the formal definition. (It is a rather straightforward but tedious exercise to derive the formal definition from the procedure computing the correspondence).

Now, let us consider again the idea of propagating an update by replacing the subparse-tree(s) corresponding to the updated element [3] e, by the new subparse-tree constructed by unpasing 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

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

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 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 4.1 *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, db' is legal[4] iff $unparse_S(T, e')$ succeeds.

Observe that if a structuring schema is local, one can propagate an update from the database to the file as follows:

Algorithm Local-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 4.1 Locality is undecidable for structuring schemas (even for $F = \emptyset$).

[3] 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.

[4] Where *legal* means that db' is the image of some file f' (using the schema S).

Proof: By reduction from the problem of testing if the intersection of two context free languages is empty. □

To conclude this section, we describe a class of structuring schemas that have the locality property. Observe that this class is general enough to capture the class of electronic document applications that motivated this research.

Definition 4.2 *A schema is in class* Local-1 *if:*
(A-1) at most one occurrence of one constructor 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.*

Due to space limitation, the proof of the following result is omitted.

Theorem 4.2 Each *Local-1* structuring schema is local.

The *Local-Update* algorithm can be used to propagate updates in Local-1 SS. 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 the set of corresponding vertexes in the parse-tree is constructed while parsing the file.

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

Remark 4.3 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 (one sequence with the symbol "," separating 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 non terminals 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 are used in a rule. This may result in database values not corresponding to any vertex in the parse-tree. As a consequence, we may have to unparse more than necessary and be forced to reconsider (A-2) in non-trivial ways. It is somewhat more tricky to relax the condition $F = \emptyset$ as has been illustrated in the previous section on unparsing.

Relaxing (A-3) may result in having parse-tree vertexes matching several database elements, (as shown in Figure 3), and thus the modified file may yield a value that differs from the modified database. 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 4.4 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 4. 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. Note that 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:

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

It constructs parse-trees having the same form as the database tree on Figure 4. Thus, deleting an element is just deleting its corresponding subtree, and similarly for insertion. □

5 Conclusion

We studied in the paper a general framework for propagating updates specified logically on a database, to a file that actually stores the data (in a structured manner). We provided general techniques for unparsing database values, and studied optimization techniques. The most important, from a practical viewpoint, is the presentation 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 without objects and union types. Objects can be easily introduced without modifying substantially the framework. On the other hand, the introduction of union types is more challenging.

To conclude, we examine the possibility of using structuring schemas to provide a view mechanism in an heterogeneous context. 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 dump (e.g., [22]). Furthermore, it is now becoming customary that the input or output files follow some structuring standard such as SGML (e.g., [22]).

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. 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.

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

References

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

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

[3] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems: the O2 proposal. In *Proc. DBPL, Salishan Lodge, Oregon*, June 1989.

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

[5] 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.

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

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

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

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

[10] 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.

[11] J. de Lima and H. Galy. The integration of structured documents into a dbms. In *Proc. Int. Conf. Electronic Publishing*, 1990.

[12] P. English et. al. An extensible object-oriented system for active documents. In *Proc. Int. Conf. Electronic Publishing*, 1990.

[13] A. FEng and T. Wakayama. Simon: A grammar based transformation system of structured documents. In *Proc. Int. Conf. Electronic Publishing*, 1994.

[14] R. Furuta and P. Stott. Specifying structured documents transformations. In *Proc. Int. Conf. Electronic Publishing*, 1988.

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

[16] 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.

[17] E. Kuikka and M. Penthonen. Transformation of documents with the use of grammars. In *Proc. Int. Conf. Electronic Publishing*, 1994.

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

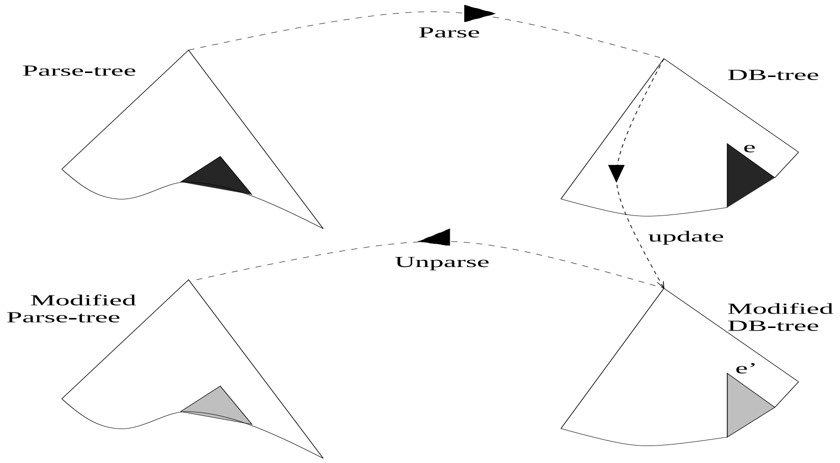

Figure 1: The Naive Propagation

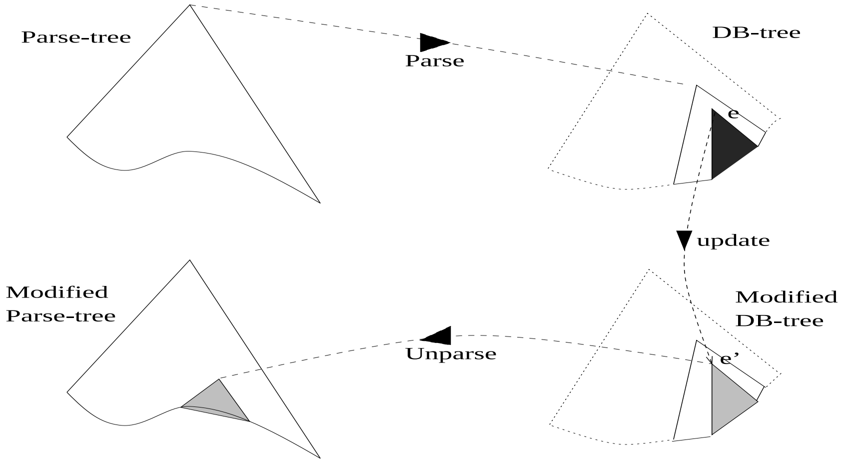

Figure 2: The Propagation Technique

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

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

[21] 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.

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

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

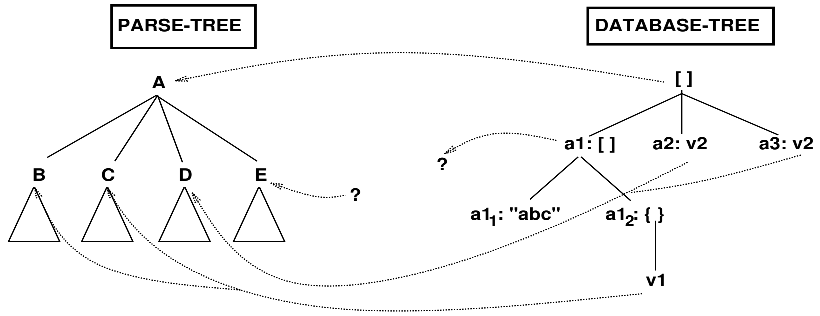

Figure 3: Parse-tree vs. Database-tree

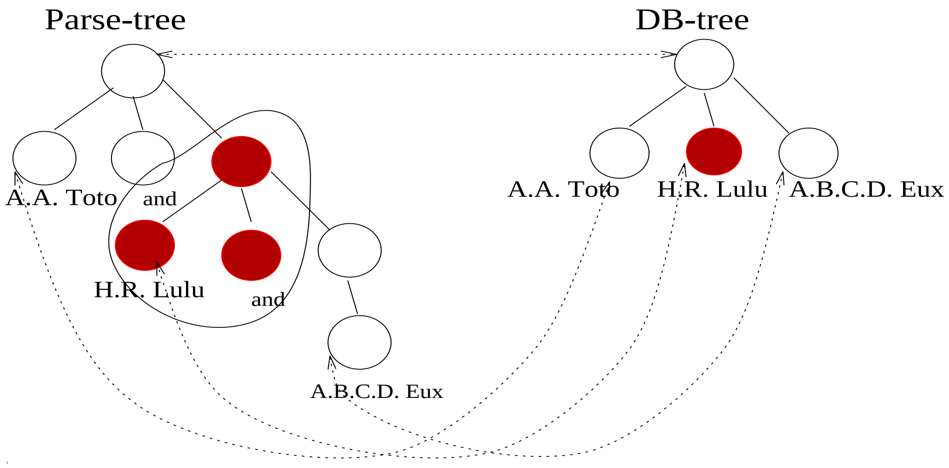

Figure 4: A list of Authors