Querying and Updating the File*

Serge Abiteboul (Serge.Abiteboul@inria.fr)
Sophie Cluet (Sophie.Cluet@inria.fr)
Tova Milo† (milo@db.toronto.edu)

I.N.R.I.A., Rocquencourt, France

Abstract

We show how structured data stored in files can benefit from standard database technology and in particular be queried and updated using database languages. We introduce the notion of *structuring schema* which consists of a grammar annotated with database programs and of a database schema. We study the translation from structured strings to databases, and the converse. We adapt optimization techniques from relational databases to our context.

1 Introduction

Database systems are concerned with structured data. Unfortunately, data is often stored in an unstructured manner (e.g., in files) even when it does have a strong internal structure (e.g., electronic documents or programs). In this paper, we consider how data stored as strings can benefit from standard database technology and in particular be queried and updated using database languages.

In actual systems, data is often stored as string for obvious historical reasons. The problem that we are considering is thus very general. Tools based on the ideas developed here would clearly be useful in a number of classical fields such as software engineering (file = program) or information retrieval (file = SGML) and possibly in more exotic ones such as genetic engineering (file = gene).

*Partially supported by Esprit Projects Fide2 and GoodStep, and by the Chateaubriand scholarship.

†Currently at University of Toronto

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

We are interested here in files that have a strict inner structure. Such files are typically accessed with general editors (e.g., vi, emacs), programs (e.g., awk); or tailored applications. The general tools take little advantage of the inner structure and provide no data abstraction. The application specific tools (e.g., SGML editors) know of the inner structure. However, they do not provide many features currently found in database systems (e.g., high level query facilities). Furthermore, their lack of generality is a severe limitation for their larger use.

Our aim is to propose a framework where database tools can be used to manipulate files based on the inner structure of the information they contain. In particular, we consider the following issues:

- How can the implicit structure of files be described?
- How can the structure be used for defining an abstract interface to the information stored in the file?
- How can this interface be used for (i) accessing (i.e., querying), (ii) updating, and (iii) maintaining the integrity of the data.

To this end, we introduce the notion of *structuring schema* which consists of a grammar annotated with database programs and of a database schema. A structuring schema specifies how the data contained in a string should be interpreted in a database.

We can use a structuring schema to translate data from a file and load it into a database. We can also understand it as the specification of a database view of the file. In this second case, to answer queries, one would like to avoid to load the data of the entire file. This can be achieved using (variations of) some standard optimization techniques from relational databases. We also briefly consider optimizations of a more grammatical nature. For updates, we highlight difficulties and present some solutions.

The second (somewhat inverse) task is to convert data from a database to files. This is typically what is performed for instance in (database) report writers or when transmitting database between distributed databases. Some of the issues are:

- How can the translation be specified?
- What information should be recorded in the target file, in order to maintain the relationship with the original data (for further querying, or updates)?

We present solutions, showing this can be used for the update problem mentioned above.

The problems of extracting data from files (and encoding data onto files to a less extent) have been popular database topics since the early days of the field. One should notably cite the system Express in the 70's for data extraction and restructuring[15]. Our contribution obeys the same motivations since the problem clearly did not disappear over the years. We use now standard database technology tools (e.g., optimization techniques, object-orientation) to provide a modern answer to the old problem.

The database language that we consider here is strongly inspired by O_2SQL [4] and the parsing by Yacc [3]. Indeed, some of the ideas described here were suggested by a first experience gained with the implementation of a prototype system on the O_2 database system [10] called O_2Yacc and with applications developed with O_2Yacc.

2 Structuring Schemas

The problem is to connect structured strings with a database or, more precisely, to get a database representation of a structured string. We present two solutions to this problem.

The first only requires the description of the inner structure of the string. This is done in a very natural way using a grammar. The connection with the database is then obtained in a straightforward manner by linking each non terminal of the grammar to a database type. We call this solution *default structuring* and we will see that it is unsatisfactory.

The second solution requires explicit statements on the links between the non terminals of the grammar and their database representation. This is done again in a very standard manner using an annotated grammar to which we attach a schema definition. We call the resulting couple a *structuring schema.*

The database construction from a text and a grammar that we propose 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) seem applicable to our context. (See [13].)

In the sequel, we assume standard knowledge on object-oriented databases, context-free grammars and parsing. We will use part of the O_2 database type system [10], but (with minor variations), we could have used any reasonable object-oriented database type system.

2.1 Default Structuring

The association between grammars and database definitions arises quite naturally from an analogy between non terminals and typed objects. For instance, we may associate a class A to each non terminal A. Intuitively, this means that an occurrence of A in a parse-tree will be represented by an object in class A. This is illustrated in the following example.

Example 2.1 (Aliases) Consider the following grammatical description of aliases:

$$\begin{array}{lcll} \langle Aliases\rangle & \rightarrow & \langle Aliases\rangle\ \langle Alias\rangle & \mid\ \epsilon \\ \langle Alias\rangle & \rightarrow & string\ \text{“,”}\ string\ \text{“;”} & \end{array}$$

It corresponds to the following class definitions with standard methods (e.g., display, edit).
class Aliases= *tuple(aliases:Aliases, alias:Alias)*
class Alias= *tuple(a_1:string, a_2:string)*

The list of aliases "m , mail ; ll , ls -l ;" is represented by an object o with the following association between objects and values:

class name	*oid*	*value*
Class Aliases :	o	$[aliases : o_1, alias : o_2]$
	o_1	$[aliases : o_3, alias : o_4]$
	o_3	$\perp$
Class Alias	o_2	$[a_1 : \text{“}ll\text{”}, a_2 : \text{“}ls - l\text{”}]$
	o_4	$[a_1 : \text{“}m\text{”}, a_2 : \text{“}mail\text{”}]$

(the empty string is associated to an object with undefined value). $\Box$

It is easy to generalize this *default structuring.* When considering general context-free grammar, *disjunctive types* will naturally arise from non terminals defined disjunctively. Since disjunction of types is not supported in O_2, this would not be directly realizable. There are of course a variety of means of simulating such types (in particular using inheritance).

The default structuring presents two major defaults. First, it clearly lacks flexibility: the resulting structure may be rather inappropriate for querying. Also, it results in the creation of too many (unnecessary) objects: e.g., in the above example o_1, o_3 have no meaningful semantics.

2.2 Structuring Schemas

As we have seen, the grammar does not provide enough semantics for the string. Indeed, a more natural way to see $\langle Aliases\rangle$ is as a list or a set of tuples (each representing one alias). An alternative (since we are in the O_2 context) is to consider $\langle Aliases\rangle$ as a list or a set of objects. Each of these views is related to the default structuring above. However,

1. nothing in the grammar allows us to choose between these possible representations (values or objects, lists or sets); and
2. depending on the application, a particular choice other than the default may be more appropriate, e.g., for querying.

Thus, the grammar does not provide enough information for interpreting the content of a string. A more precise and flexible way of specifying its semantics is provided by a *structuring schema*. A structuring schema consists of a schema and an annotated grammar. 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 (that may be object or value) of a word derived from this rule is constructed using the database representations of the subwords derived from $A_1, \ldots, A_n$. In the sequel, we use a Yacc-like notation [3]. In a rule $A \rightarrow A_1, \ldots, A_n$, \$i denotes the database image of the string corresponding to A_i, and \$\$ the one associated to A.

The schema describes the database types, classes (types + methods) and inheritance relationship. 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. The next example provides a structuring schema for aliases.

Example 2.2 (Aliases continued) The schema is the following:

```
/* Class definition */
class Alias = tuple(Name: string, Definition: string)
              method edit, update_def(string);
              ....
/* Non terminals type definition */
type ⟨Aliases⟩ : set(Alias);
type ⟨Alias⟩ : Alias ;
```

The schema provides type definitions for the non terminals of the grammar. Note the distinction between *Alias*, a class name; and $\langle Alias \rangle$ which denotes a string.

The annotated grammar is given by:

$$
\begin{array}{lll}
grammar & & Alias\_Grammar = \\
\langle Aliases \rangle & \rightarrow & \langle Aliases \rangle \; \langle Alias \rangle \\
 & & \{\$\$:= \$1 \cup set(\$2)\} \\
 & | & \epsilon \\
 & & \{\$\$:= set()\} \\
\langle Alias \rangle & \rightarrow & string \text{ ``,'' } string \text{ ``;''} \\
 & & \{\$\$:= new(Alias, \\
 & & \quad tuple(Name : \$1, Definition : \$2))\}
\end{array}
$$

Observe that an object of class *Alias* will be constructed each time the rule

$$\langle Alias \rangle \rightarrow string \text{ ``,'' } string \text{ ``;''}$$

will be used.

Now suppose that we have a file *myaliases*, we can load in the database the aliases that it contains using:

$$aliases := load\_string(Alias\_Grammar, \text{``}myaliases\text{''})$$

where *aliases* denotes the result of the parsing, i.e., a set of objects of class *Alias*. □

The language we use in the structuring schema is strongly inspired from O_2SQL [4]. We chose a core subset of this language (sufficient to our purpose, e.g., without aggregate functions) and extended it with update features. We will see more of these new features in the sequel. A similar development can be done using other standard OO-database languages as well.

O_2Yacc

Structuring schemas allow to extract data from a string in a fairly convenient way. A class O_2Yacc has been implemented in the O_2 system to provide such a functionality. The actions are written in O_2C. The prototype was easy to implement. Among other applications, it has been used to construct a bibliography database starting from BibTex files. The structuring schema for the BibTex application was developed in a couple of days. (A difficulty of this particular application is the management of duplicate references.)

It is clear when one uses the prototype that the only mode that it provides, loading entirely the data to answer any query, is not acceptable. This led to developing the optimization technique also described in the present paper.

3 Views and Optimization

All we have so far is a sophisticated use of a database programming language (O_2SQL and/or O_2C) with a parser (Yacc) to transfer data from a file to a database. Once there, the data can be manipulated using standard database tools, e.g., query languages [6].

This suffices to transfer data from a file to a database (and forget about the file). On the other hand, suppose that we want the file and the database (with data coming from the file) to logically coexist. Then, one may question the interest of having data physically in two different stores: (i) the storage cost may be prohibitive and (ii) the coherence between the database and the file after updates from one source or the other may be difficult to maintain. Thus, in this case, it is better to consider structuring schemas not as a way of extracting data from a file but as a way of specifying virtual data that is loaded only when needed.

This leads to distinguishing two ways of using a structuring schema:

1. as a building mechanism to load data, and
2. as a view specification mechanism.

We already addressed the first aspect and are now interested in the second. We are facing two problems common to standard view definitions: querying and updating. We will address the first in this section. The latter is considered in Section 7.

Although the notion of view in an object-oriented database context is a rather new topic, lots of work has been already devoted to the issue (e.g., [12, 9, 1]). Many of the problems that will be found are not new to oo-databases. In particular, we will meet the *materialized view problem* and the *view update problem* that are both now textbook material (see [17]).

To answer a query on the database view of a file, one may construct the database image of the file and then evaluate the query. This technique will obviously lead to the construction of many unnecessary objects and complex values. We will show that this unnecessary effort can be avoided using standard query optimization techniques slightly revisited. We start by giving an intuition of the process with a very simple example. We then present a sketch of the optimization algorithm. In the next section, we will see that things are somewhat more complex in general.

Example of Optimization

Suppose that we have a view on a file through the *aliases* structuring schema (we use the same structuring schema, but now we see it as *virtual*) and suppose also that we want to evaluate the following query:

$$\begin{array}{ll} select & x.Definition \\ from & x \ in \ aliases \\ where & x.Name \ = \ \text{“}lm\text{”}. \end{array}$$

The corresponding algebraic query is the following:

$$\varphi(X) \equiv \Pi_{Definition}(\sigma_{Name=\text{“}lm\text{”}}(X)).$$

To reduce the construction performed for evaluating the query, we push the query down inside the grammar specification. We leave the grammatical part of the structuring schema unchanged (the parser must still recognize the same file) while modifying the data construction part in an appropriate manner. The rule used to compute the *aliases* is:

$$\begin{array}{lcl} \langle Aliases \rangle & \rightarrow & \langle Aliases \rangle \ \langle Alias \rangle \\ & & \{\$\$:= \$1 \cup set(\$2)\} \\ & | & \epsilon \\ & & \{\$\$:= set()\} \end{array}$$

The desired result is thus obtained using the query φ above:

$$\textbf{(1)} \quad \begin{array}{lcl} \langle \varphi(Aliases) \rangle & \rightarrow & \langle Aliases \rangle \ \langle Alias \rangle \\ & & \{\$\$:= \varphi(\$1 \cup set(\$2))\} \\ & | & \epsilon \\ & & \{\$\$:= \varphi(set())\} \end{array}$$

Now we are back to (almost standard) algebraic optimization:

$$\textbf{(2)} \quad \begin{array}{c} \varphi(set()) \\ \rightsquigarrow \\ \Pi_{Definition}(\sigma_{Name=\text{“}lm\text{”}}(set())) \\ \rightsquigarrow \\ set() \end{array}$$

and

$$\textbf{(3)} \quad \begin{array}{c} \varphi(\$1 \cup set(\$2)) \\ \rightsquigarrow \\ \varphi(\$1) \cup \Pi_{Definition}(\sigma_{Name=\text{“}lm\text{”}}(set(\$2))) \\ \rightsquigarrow \\ \varphi(\$1) \cup set(\Pi_{Definition}(\sigma_{Name=\text{“}lm\text{”}}(\$2))). \end{array}$$

All this is rather standard (e.g., distributivity of projection w.r.t. union) except for the pushing of the query onto a single element ($2 is not a set but an object). For this, we need to extend the algebra to have our operations also operate on single elements. This is rather straightforward if we view such an element as a singleton set. We therefore extend the algebra to single elements. 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).

Now, we go on pushing the query down the grammar. The query on $1 and on $2 is pushed down on the corresponding non terminals:

$$\textbf{(4)} \quad \begin{array}{lcl} \langle \varphi(Aliases) \rangle & \rightarrow & \langle \varphi(Aliases) \rangle \ \langle \varphi(Alias) \rangle \\ & & \{\$\$:= \$1 \cup set(\$2)\} \\ & | & \epsilon \\ & & \{\$\$:= set()\} \end{array}$$

We have a definition for the nonterminal $\langle \varphi(Aliases) \rangle$. We obtain a definition for $\langle \varphi(Alias) \rangle$ by pushing φ in the grammar rule defining $\langle Alias \rangle$ in the following way:

$$\textbf{(5)} \quad \begin{array}{l} \langle \varphi(Alias) \rangle \ \rightarrow \ string \ \text{“,”} \ string \ \text{“;”} \\ \{\$\$:= \varphi(new(Alias, \\ \qquad tuple(Name : \$1, Definition : \$2)))\} \end{array}$$

We can use again a rewriting technique:

$$\textbf{(6)} \quad \begin{array}{l} \Pi_{Definition}(\sigma_{Name=\text{“}lm\text{”}}(\\ new(Alias, tuple(Name : \$1, Definition : \$2)))) \\ \qquad \rightsquigarrow \\ \Pi_{Definition}(new(Alias, \sigma_{Name=\text{“}lm\text{”}}(\\ \qquad tuple(Name : \$1, Definition : \$2)))) \\ \qquad \rightsquigarrow \\ \Pi_{Definition}(\\ tuple(Name : \sigma_{\lambda.x(x=\text{“}lm\text{”})}(\$1), Definition : \$2))) \end{array}$$

This uses a slightly unusual rewrite rule: the absorption of object creation (*new*) by projection. 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 *lm*, 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). The advantage of the above optimized construction is (i) it does not construct *any* object, and (ii) it discards immediately irrelevant tuples.

To conclude with the above example, suppose that we want to obtain the objects and not only the *Definition* attribute (e.g., to edit them). Then we use:

$$\begin{array}{ll} select & x \\ from & x \ in \ aliases \\ where & x.Name = \text{"}lm\text{"}. \end{array}$$

The same optimization (except for the absorption of *new*) would yield a structuring scheme which creates objects only for *lm* aliases.

The Optimization Algorithm

Let us consider more precisely the algorithm we used for this query optimization. It consists in applying alternatively two kinds of rewriting rules that we call *grammar rewriting rules* and *query rewriting rules*. For space limitations, the presentation of the rewrite rules is rather informal.

The *grammar rewriting rules* push queries down the grammar specification. In the previous example, they are the ones used in Steps (1,4,5). Once a query is pushed into a rule, query rewriting rules are applied.

The *query rewriting rules* are more traditional and rely on algebraic equivalences [5, 8, 14, 16]. The algebraic expression assigned to \$\$ (which is the algebraic translation of an O_2SQL expression) is rewritten in order to cut it into subexpressions on the \$i variables. In the previous example, these rules were applied in Steps (2,3,6). Once these rules have been suitably applied we go back to the grammar rewriting rules.

The algorithm stops when no rewriting rules can be applied. In the best case, the only objects/values constructed are the ones returned by the query (e.g., the previous example). In the worst case, the query remains at the root of the grammar and the whole data is constructed.

4 Closer Look at the Problems

To give an intuition of the optimization algorithm, we had to omit important aspects that are considered now. These are summarized in the following questions:

1. The *grammar rewriting rules* generate new non terminals (e.g., $\langle\varphi(Aliases)\rangle$) and new rules. What happens to the old ones?

2. In the previous examples, the result of the view in a single set (\$\$ for the start symbol of the grammar). Is it possible to specify several virtual anchors with a unique grammar?

3. In object oriented databases, cyclic data is common. Does the presence of cyclic data have impact on our technique?

4. In [2], duplicate elimination was shown to be an important issue for languages allowing the creation of objects. Can we eliminate duplicates? And does this have impact with our technique?

5. We showed the optimization of a simple query. What happens when considering complex queries?

Each of these questions highlights some limitations of the technique or subtle issues which force us to complicate this technique. We next address these different questions. The end of this section is rather technical; and it is possible to skip it and continue directly with Section 5. We use a slightly more complicated example.

Example 4.1 We are interested in a database view of a BibTex file[1]. For those unfamiliar with BibTex, below is one BibTex reference:

$$\begin{array}{rl} @Inproceedings\{ & ChandraLM81, \\ & author = \text{"}A.K.Chandra\ and \\ & R.Lewis\ and\ J.A.Makowsky\text{"}, \\ \% & source = \text{"}Kanellakis91\text{"} \\ & title = \text{"}Embedded\ Implicational \\ & \ldots their\ Inference\ Problem\text{"}, \\ & booktitle = stoc, \\ & year = \text{"}1981\text{"}, \\ & pages = \text{"}342-354\text{"}\} \end{array}$$

We now give a partial structuring schema.

```
/* Types and Classes */
```

$$\begin{array}{lll} Class\ Reference & = & \\ & tuple(& Keyword : string, \\ & & Authors : Author\_Set \\ & & Source : string, \ldots) \end{array}$$

```
/* Non terminals type definition */
```

$$\begin{array}{lll} Type\ \langle Ref\_set\rangle & = & set(Reference) \\ Type\ \langle Reference\rangle & = & Reference \\ Type\ \langle String\rangle & = & string \end{array}$$

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

[1] BibTex is a program which, given a file containing bibliographical data (in BibTex format) produces a LaTex bibliography. The syntax of the bibliographical data should be self explanatory. We have slightly modified the syntax to simplify the presentation.

grammar $Alias\_Grammar$ =

$$
\begin{array}{lcl}
\langle Ref\_Set\rangle & \rightarrow & \langle Reference\rangle\ \langle Ref\_Set\rangle \\
 & & \{\$\$:= set(\$1) \cup \$2\} \\
 & | & \epsilon \\
 & & \{\$\$:= set()\} \\
\langle Reference\rangle & \rightarrow & \ldots \\
 & & \text{“}@Inproceedings\{\text{”}\ \langle String\rangle \\
 & & \text{“}author = \text{”}\ \langle Author\_Set\rangle \\
 & & \text{“}\%source = \text{”}\ \langle String\rangle \\
 & & \ldots \\
 & & \{\$\$:= new(Reference, \\
 & & \quad tuple(Keyword : \$1, \\
 & & \quad\quad Authors : \$2 \\
 & & \quad\quad Source : \$3, \\
 & & \quad\quad \ldots)\} \\
\langle String\rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\}
\end{array}
$$

□

4.1 Multiplication of Rules

The optimization algorithm applies *grammar rewriting rules* whose effect is to modify a rule to obtain a new rule (e.g., $\langle Aliases\rangle$ into $\langle\varphi(Aliases)\rangle$). What happens to the old rules? The old rules are kept in the grammar. The reason for this is that one rule may have different uses in different places of the grammar. In some cases, this may yield unreachable nonterminals that are then just discarded. The nonterminals that are defined in the annotated grammar for *alias* after the rewriting process are the following:

$$\langle aliases\rangle,\ \langle\varphi(aliases)\rangle,\ \langle alias\rangle,\ \langle\varphi(alias)\rangle.$$

Considering the rules defining these non terminals, it must be noted that $\langle aliases\rangle$ or $\langle alias\rangle$ is not derivable from the new start symbol $\langle\varphi(aliases)\rangle$. Thus, it may be discarded. However, such nonterminals cannot be discarded in general:

Example 4.2 Let us now consider the following query: $\varphi(X) =$
$\sigma_{Keyword=\text{“}ChandraLM81\text{”}\wedge source=\text{“}Kanellakis91\text{”}}(X)$
Suppose that we apply this query on the view

$$references := view\_string(Bib\_Grammar, \text{“}my\_bib\text{”}).$$

In a manner similar to that of the *aliases* example, the rewriting process will derive the following rule:

$$
\begin{array}{lcl}
\langle\varphi(Reference)\rangle & \rightarrow & \ldots \\
 & & \{\$\$:= new(Reference, \\
 & & \quad tuple(Keyword : \varphi_1(\$1), \\
 & & \quad\quad Authors : \$2, \\
 & & \quad\quad Source : \varphi_2(\$3), \\
 & & \quad\quad \ldots)\}
\end{array}
$$

with $\varphi_1(X) = \sigma_{\lambda x.(x=\text{“}ChandraLM81\text{”})}(X)$ and $\varphi_2(X) = \sigma_{\lambda x.(x=\text{“}Kanellakis91\text{”})}(X)$
Now, if we apply grammar rewriting rules to push the queries further down, we have:

$$
\begin{array}{lcl}
\langle\varphi(Reference)\rangle & \rightarrow & \ldots \\
 & & \text{“}@Inproceedings\{\text{”} \\
 & & \quad \langle\varphi_1(String)\rangle \\
 & & \text{“}author = \text{”}\ \langle Author\_Set\rangle \\
 & & \text{“}\%source = \text{”}\ \langle\varphi_2(String)\rangle \\
 & & \ldots \\
 & & \{\$\$:= new(Reference, \\
 & & \quad tuple(Keyword : \$1, \\
 & & \quad\quad Authors : \$2, \\
 & & \quad\quad Source : \$3, \\
 & & \quad\quad \ldots))\} \\
\langle\varphi_1(String)\rangle & \rightarrow & string \\
 & & \{\$\$:= \varphi_1(\$1)\} \\
\langle\varphi_2(String)\rangle & \rightarrow & string \\
 & & \{\$\$:= \varphi_2(\$1)\} \\
\langle String\rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\}
\end{array}
$$

□

This example shows that several nonterminals may be derived from the same original one while still needing that original one. Thus no rule should be discarded unless one shows (which is rather straightforward to check) that the nonterminal in the head of the rule is not reachable.

4.2 Using Several Anchors

If we consider the *references* view that we defined above, we may notice that the only access the user has to the data is through the set containing all the references. This does not seem satisfactory. For example, one may want to be able to access directly the set of all authors as well. Of course, one can construct first the set of references, and *then* specify a set of authors (virtual as well) defined using the references. This is a possibility which may be unpractical for many applications. In this brief section, we consider an alternative that would allow to construct (again virtually) in the specification the set of authors and the information concerning them.

Remark: In the O_2 OODBMS, the user accesses the data through names that are explicit in the schema and that may be considered as global variables. It is then the user responsibility to assign and update these variables. In other OODBMS, class extents are automatically maintained by the system. It is not our goal here to advocate one or the other choices. However, observe that the maintenance of these extents from the point of view of the optimizer is as complicated as the maintenance of global variables (i.e., names, such as set of authors) that are constructed in the specification. □

As we saw, the optimization process starts by considering the rules defining the start symbol of the grammar. This is due to the fact that, using our view definitions, the corresponding $$ variables denote the set of relevant object/values. Then the optimization process smoothly pushes (if possible) the projection-selection down the grammar. Allowing global variables (or extents) updated at different levels of the grammar requires a preliminary analysis in order to know where to push the query. The "pushing" must sometimes be blocked: elements that may be relevant for other parts of the query should not disappear. This is the difficulty of performing optimization in presence of side-effects.

We propose the following solution. Suppose that we want to maintain two anchors (say *references* and *authors*). We define these anchors as fields of the $$ corresponding to the start symbol. The elements contained in these anchors are pushed up the parsing tree to the top. To illustrate this idea, we modify the BibTex structuring schema as follows:

`/* Types and Classes */`

$Class \quad Author \quad = \quad string$

...

`/* Non terminals type definition */`

$$
\begin{array}{ll}
Type\langle BibTex\rangle = & tuple(\\
 & \quad references : set(Reference) \\
 & \quad authors : set(Author)) \\
\ldots & \\
Type\langle Author\_set\rangle = & set(Author) \\
Type\langle Author\rangle = & Author
\end{array}
$$

`/* Annotated Grammar */`

$$
\begin{array}{ll}
\langle BibTex\rangle & \rightarrow \langle Reference\rangle\ \langle BibTex\rangle \\
 & \{\$\$:= tuple(\\
 & \qquad references : \$2.references \\
 & \qquad\qquad \cup\ set(\$1), \\
 & \qquad authors : \$2.authors \\
 & \qquad\qquad \cup\ \$1.authors)\} \\
 & \mid \epsilon \\
 & \{\$\$:= tuple(\\
 & \qquad references : set(), \\
 & \qquad authors : set())\} \\
\langle Reference\rangle & \rightarrow \ldots \\
 & Author\text{“} = \text{”}\ \langle Author\_Set\rangle \\
 & \ldots \\
 & \{\$\$:= new(Reference, \\
 & \qquad tuple(\ldots, \\
 & \qquad\qquad Authors : \$2, \\
 & \qquad\qquad \ldots.)\} \\
\langle author\_set\rangle & \rightarrow \langle author\rangle\text{“,”}\langle author\_set\rangle \\
 & \{\$\$:= set(\$1) \cup \$2\} \\
 & \mid \langle author\rangle \\
 & \{\$\$:= set(\$1)\} \\
\langle author\rangle & \rightarrow string \\
 & \{\$\$:= new(Author, \$1)\}
\end{array}
$$

As we may see, the set containing the authors is accessible from the rules defining the start symbol $\langle BibTex\rangle$. Thus, a query defined on the set of authors can be pushed down the grammar from these rules.

Let us reconsider the view definition:

BibTex=view(Bib_Grammar, "my_bib")

The parsing returns a tuple with two attributes denoting a set of references and a set of authors. In order to formulate the query given in Example 4.2, we now need an extra projection operation in order to access the set of references ($\varphi(X) = \sigma_{source=\ldots}(\Pi_{references}(X))$)

4.3 Dealing with Cyclic Data

To simplify the exposition of the previous issues, we deliberately simplified the BibTex example. We did not consider cyclic data. In an OO context, cyclic data is common. For instance, a more natural way to view the class author will be the following:

$$
Class \quad Author \quad = \quad tuple(\quad name : string, \quad refs : set(Reference))
$$

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 rule and update their values in another. Before we illustrate that, it is important to note that this issue is closely related to the issue of duplicate elimination which is postponed to the next subsection.

The annotated grammar corresponding to the new definition of Class *Author* is as follows:

$$
\begin{array}{ll}
\langle Reference\rangle & \rightarrow \ldots \\
 & \text{“}author = \text{”}\langle Author\_Set\rangle \\
 & \ldots \\
 & \{\$\$:= new(Reference, \\
 & \qquad tuple(\ldots, Authors : \$2, \\
 & \qquad\qquad \ldots); \\
 & MAP_{\lambda x.(x.refs := x.refs\ \cup\ set(\$\$))}(\$2)\} \\
\langle author\rangle & \rightarrow string \\
 & \{\$\$:= new(Author, \\
 & \qquad tuple(name : \$1, \\
 & \qquad\qquad refs : set())\}
\end{array}
$$

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 now allow several assignments in the same rule specification forces us to reconsider our rewriting rules. In order to simplify the rewriting process, we decided to restrict the specification language in the following way:

Restriction: The construction part of a rule starts with an assignment on $$, possibly followed by updating assignments on the $i's and $$. It must be noted, that this restriction does not prohibit any significant updates.

Now, having this restriction, pushing a query into a rule is done as before (i.e. the query is applied on the right part of the first assignment on $$). The main difference relies in the decision of pushing a query or not, and whether the updates after the assignment to $$ can be removed. Let us illustrate this point with an example. Suppose we have the following query on the BibTex view.

$$\varphi(X) = \Pi_{Keyword}(\Pi_{references}(X))$$

Since the rule defining $\langle BibTex \rangle$ contains only one assignment to $$, and no additional updates, the query can be pushed into this rule as before. After further (query) rewriting, we would like to push $\varphi'(X) = \Pi_{Keyword}(X)$ into the rule defining $\langle Reference \rangle$. Now, having an added update after the assignment on $$, we have to make sure that this update does not modify the *Keyword*. In this case, the update on the set of authors has no influence on the attribute *Keyword*. Since the update is irrelevant to the query result, it can be removed from the rule, and the query can be pushed as before.

As another example consider the query of Example 4.2. At some point of the optimization process, one wants to push the query

$$\varphi(X) = \sigma_{Keyword=\text{“}ChandraLM81\text{”} \wedge source=\text{“}Kanellakis91\text{”}}(X)$$

into the rule defining $\langle Reference \rangle$. This query implicitly involves attributes other than *Keyword* and *Source*; the query returns the whole reference, including the *Authors* attribute. Thus clearly we can not remove the update to the set of authors. Note however that the *Keyword* and *Source* attributes are not modified or used in the update to the set of authors. Thus we can push the selection into the assignment on $$, leave the update on the set of authors unchanged, and get the desired result.

When objects are involved, updates can be very tricky. Our goal is not to present the numerous dangerous situations that may arise (and that are very unnatural). We just present strong conditions that that guaranties pushing a query down the grammar will not lead to a wrong result. For lack of space we only present below the rewrite rules used in the above examples. Other cases are handled similarly.

Given a query Q, and a literal A, assume that we want to push Q into a rule where A appears in the head. We assume that the construction associated with the rule starts with an assignment on $$ of the form $\$\$:= new(C, tuple(a_1 : \$i_1, \ldots, a_n : \$i_n))$, followed by a sequence of updates $\vec{U}$.

Strong Condition A

1. Let Q be a query of the form $Q = \varphi(\Pi_{a_{j_1},\ldots,a_{j_m}}(\$\$))$. If non of the projected attributes of $$ and non of the $\$i$'s corresponding to those attributes participate in $\vec{U}$, Then, Q can be pushed into the rule by applying it on right part of the assignment on $$, and removing all the other updates.

2. Let Q be a query of the form $Q = \varphi(\sigma_c(\$\$))$. If non of the attributes of $$ used in the selection criteria and none of the $\$i$'s corresponding to those attributes participate in $\vec{U}$, Then, the selection σ_c can be pushed into the rule by applying it on the right part of the assignment on $$. Note that in this case, the other updates in $\vec{U}$ remain unchanged.

It must be noted that this strong condition may, in some cases, forbid *correct* pushing of queries.

4.4 Avoiding Duplicates of Objects

As shown in [2], duplicate elimination is a crucial problem for languages allowing the creation of objects. We will see that it is indeed true in our context. This problem is also closely related to cyclic data. Let us consider the BibTex view and the new definition of class *Author*. If we leave the annotated grammar as it is, there is no way that we can group all the references corresponding to one author in the same object. Indeed, an author figuring twice in the *my_bib* file will be represented by two objects, each of which will contain the reference in which it appears in the file.

This is not what we would expect and, accordingly, we have to give the user the means to avoid duplicates. We do that by introducing a new primitive that we call *conditional new*. Let us consider the modifications we have to perform on the BibTex annotated grammar in order to avoid duplicates.

$$\langle author \rangle \quad \rightarrow \quad string$$
$$\{\$\$:= new(Author, \\ \sigma_{name=\$1}, \\ tuple(name : \$1, refs : set()))\}$$

The *conditional new* primitive requires three parameters: the name of a class, a query and a value. The query is applied on the extent of the class whose name is denoted by the first parameter. If it returns the empty set, an object is created and added to the class extent. The value (state) of the object is given by the third parameter. If the query returns a nonempty set, the "first" object of this set will be assigned to $$ (if the program is correct, there will only be one object in the returned set).

It must be noted that this primitive requires that the system maintains the extent of the class. Recall that we addressed the problem of querying such extents in Section 4.2. If one want to optimize queries on class extents, their members should be pushed up the grammar specification and be also maintained at the root of the grammar.

Now that we have seen how duplicates could be eliminated, let us consider the consequence of having *conditional new* in the rewriting process. The problem is that objects created in one particular subtree T_1 may be used in different subtrees $T_2, T_3, .., T_n$. Thus, even if the query we are considering is not concerned with the information generated by T_1, the construction must still be performed in order to preserve the structure generated in $T_2, T_3, .., T_n$.

The solution to this problem consists in distinguishing a special kind of literals called **global**. We say that a literal whose associated construction specification contains a *conditional new* is a **global** literal. In order to treat global literals we introduce a new strong condition that is added to the one defined in Section 4.3.

Strong Condition B

1. A query Q can be pushed on a literal A if A does not derive a global literal.
2. If Q is a query of the form $Q = \varphi(\Pi_{a_{j_1},\dots,a_{j_m}}(X))$, Then the projection $\Pi_{a_{j_1},\dots,a_{j_m}}$ can be pushed into the rule. The additional updates can be removed only if all the variables appearing in them correspond to literals that do not derive global literals.
3. If Q is of the form $Q = \varphi(\sigma_c(X))$, and none of the literals corresponding to the variables ($\$i$'s) used in the selection criteria derives a global literal. Then the selection σ_c can be pushed into the rule.

Once again, this condition will block some "correct" rewriting. However, it can be suitably weakened in some particular cases that we will not consider here for lack of space.

4.5 Complex Queries

Complex queries are the ones involving several view definitions (e.g. the BibTex authors that are cited in a LaTex file) or several occurrences of the same view (e.g. nested queries). We will not address in details the problems they raise because they are similar to that of complex queries in any rewriting system. Indeed, as in any rewriting system, the problem consists in rewriting the query in order to obtain a join involving several subqueries on distinct elements. In the OO context, algebraic equivalences for performing this rewriting can be found in [14, 16]. Of course, this is not always feasible and, in some cases, we will have to construct more data than needed (in the worst case, the whole data corresponding to a view will have to be loaded).

5 Optimizing the grammar

In the previous section, we assumed that the grammar is not modified which may be an acceptable restriction for most practical purposes. In this section, we briefly show that it may be possible to achieve more by also modifying the grammar. The topic is interesting technically and is new to our knowledge. Its practical interest is yet unclear.

We next briefly present a possible optimization based on modifying the grammar to illustrate this potential direction of research.

Suppose that we have a rule:

$$\langle A \rangle \leftarrow LeftDelimiter\ \langle A_1 \rangle \dots \langle A_n \rangle\ RightDelimiter$$

such that the delimiters are tokens not occurring in $\langle A_1 \rangle \dots \langle A_n \rangle$ and the information in $\langle A_1 \rangle \dots \langle A_n \rangle$ is not used in the query. Then we may replace the parsing of $\langle A_1 \rangle \dots \langle A_n \rangle$ by a scan for *RightDelimiter*.

Furthermore, between two delimiters, it may be possible to switch from an analysis mode to a scanning mode when some conditions are met. For instance, suppose that we are looking for a reference with a given keyword in the bibliography database. When a keyword has been tested and the test failed, the parser can switch to a scanning mode and start searching for "}" (the right parenthesis closing this particular reference).

Such technique can clearly be automatized, i.e., one may analyze the grammar graph to detect such situations and apply the relevant parsing optimization. Although this may yield speed-up in many cases, we believe that the gain will be marginal compared to that provided by the technique presented in the previous section.

6 From Databases to Files

So far, we have been concerned with extracting structures from files. In this section, we study the converse. This converse is of practical importance and furthermore, it will prove to be useful when considering updates in Section 7.

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. This may be necessary in many contexts such as report generation or transfer of data between databases. Not surprisingly, one of the most intricate issues will be the naming of database objects.

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:

Integers, Reals, Strings, etc
We need some standard encodings of these.

Tupling

The application programmer may define the encoding of tuple $\langle A_1 : t_1, \ldots, A_n : t_n\rangle$ for instance as:

$$enc(\langle A_1 : t_1, \ldots, A_n : t_n\rangle) = \text{“}A_1 : \text{”} \ . \ enc(t_1) \ . \ \text{“;”} \ldots \text{“;”} \ . \ \text{“}A_n\text{”} \ . \ enc(t_n)$$

where “.” denotes concatenation.

Bulk Types

This is also rather simple using iteration, and for instance, the structural induction operator of [7]. 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.)

In O_2C, structural induction can be realized using *for each* statements. The following programs encodes a set X of tuples:

$$\begin{array}{l} enc := \text{“”}; \\ for\ each\ x\ in\ X \\ \quad \{enc := concat(enc, \text{“}A_1 : \text{”}, enc(x.A_1), \text{“;”}, \\ \qquad\qquad \ldots, \text{“;”}, \text{“}A_n : \text{”}, enc(x.A_n)); \}; \\ enc := concat(enc, \text{“.”}); \end{array}$$

Objects

When an object is transformed into a string, one may want to record both the oid, and the state. This leads to the following issues:

> **Question 1:** How do we encode oid's?
> **Question 2:** How do we handle data cycles?

To encode oid's, there are many possibilities. We may use some surrogate, or some internal representation. Note that some systems consider oid's as data values that can even be printed. In all cases, the problem is to guarantee a meaningful semantics to updates. Suppose for instance that we use a conversion table that translates oid's to say, integers. We have then to be very precise on the scope of this translation: is it valid for this specific output string only? Is this translation persistent? Is it general to the system or is it particular to an application? Such questions have to be answered[2].

The second issue comes from the transcription of object values when data is cyclic. We need to flag the objects that have already been considered. Suppose that we want to output in a string a genealogy database. We want to avoid entering in an infinite loop:

$$\begin{array}{l} [\ o1 : (Name : Adam, \\ \qquad Spouse : o2 : (Name : Eva, \\ \qquad\qquad Spouse : o1 : (Name : Adam \ldots] \end{array}$$

This is easily handled by maintaining an occur check.

The above discussion leads to the following general framework. To extract data from a database to a file, we must specify how each occurrence of a data constructor is handled, i.e., we annotate the database schema with indications on the encoding of these occurrences. (Of course, a default encoding is provided by the system in particular, for oid's and the base types such as *string*.)

Observe that the previous process only yields a string, while we insisted so far that grammars are attached to the strings that we consider. It is not difficult to obtain automatically a grammar (just by following the generation of the string). But, as mentioned above a grammar alone is a very limited way of representing a database structure. We would like is to obtain automatically the inverse structuring schema. This is not possible in general. However, if we restrict our attention to a few system-supplied (eventually customizable) kinds of encodings, this becomes relatively straightforward.

What do we mean by “system-supplied encodings”? First, there are standards for documents such as SGML that the application should preferably adopt. Also, the system may offer a few choices for encodings for each constructor. For instance, consider a set, say a set of integers. For most practical purposes, it suffices to specify the encoding of an integer; the delimiters between the integers (e.g., a “,”), a string to put at the end (“:end of list”); and one to put at the beginning (“list of integers separated with commas:”).

This leads to considering the parsing/encoding problem in a more global manner. We will come back to this fundamental issue in Section 7.

7 Updates

Returning to structuring schemas, suppose that we have a file containing data and a database view of that file.

[2] The situation may be even more complicated in the context of several databases. It is necessary to distinguish between objects from different databases. This is of course without mentioning the problem of naming of objects migrating from one database to another.

We now consider updates. A difficulty is that data is represented in two forms: files and databases.

7.1 Updating the Files

There is a priori no problem in allowing an update to a file when a (database) view has been defined on it. If the view is materialized, we have to maintain it. We run into the classical problem of materialized view update. One has to provide incremental algorithms to avoid having to reparse the entire file. A solution is to keep the parse tree and maintain a correspondence between nonterminals in the parse tree and indices of characters in the file. When updating the file, we try to limit the impact of the update to a nonterminal of the parse tree corresponding to a minimal substring. This update will therefore have minimal effect on the materialized database.

Observe that structuring schemas can serve as the basis for providing concurrency control on files with a finer level of granularity than the whole file. Suppose that the database provides object-level concurrency control. Then by specifying a structuring schema, we specify the level of concurrency that we allow on a given file: it is necessary to "lock" only the minimal substring corresponding to a pair nonterminal/database-object which is affected by the update. This lock can be achieved (at least virtually) by locking the corresponding database object.

To illustrate this technique, consider C programs. Suppose that the structuring schema associates an object to each C function. This will allow two users to modify concurrently two distinct functions in the same file.

7.2 Updating the Database

We now run into a much more challenging problem, *the view update problem.* There is a-priori no requirement on the function from the file to the database; and in particular it does not have to be a one-to-one mapping. Thus, an update to the database may have zero, one or many translations on the file. Clearly, one should not expect miracles and, in most cases, such updates will be impossible.

We consider two modest ways of allowing such updates, a very limited one for arbitrary structuring schemas, and an extremely powerful one for very constrained structuring schemas:

Arbitrary structuring schema
: Consider first a lexical token of the parser, say a string. Given an occurrence w of this token in the text, one can find where this particular string is kept in the database (it may be in zero, one or more than one places). An update to one of these database strings may be allowed and would lead to the update to the corresponding string in the file. (An example of such an update would be the modification of the definition of some alias.) This is very modest in the sense that it would be limited to specific lexical units.

  It seems difficult to offer more automatically. It is even difficult to allow application programmers to explicitly specify the meaning of database updates when the database is the view of a file. For instance, suppose that we would like to write in the database environment a method that would insert a new alias in the list of aliases. This would involve the specification of the modification of the file. There are few available tools for that and these tools (e.g., pushdown translation [11]) are very heavy compared to database update tools. A possibility is use the transformations from databases to files of Section 6: the new state of the file is entirely obtained (at least logically) from the database. (Optimization is clearly necessary to avoid reconstructing entirely the file.)

Constrained structuring schema.
: We restrict our attention to structuring schemas such that (i) there are one-to-one correspondences between substrings and database portions, (ii) these correspondences are easily extracted from the specifications, and (iii) inverse mapping can easily be obtained as well.

  This leads to reconsidering entirely the problem which we do next.

7.3 Reconsidering the Issue

In this section, we argue that some of the issues raised by updates disappear in an (open to debate) different way of addressing the problem.

A major limitation of our approach so far is that we started from the string and from there went to data. The grammar tells us something about the data structure and an analysis of the actions of the structuring schema tells us a little more. But still, our knowledge of the data is limited.

We will argue that we should not start with the grammatical specification:

- Firstly, if we handle a large quantity of data, it seems more appropriate to do it directly in a database rather than in a file. Clearly, nowadays large quantities of data (e.g., large programs or documents) are still stored in file systems. However, this should be considered more as a temporary aberration than as a desirable state.

- Secondly, in other motivating examples for studying "structured strings" such as data transfer between databases, the data is originally structured before being on a string medium.

So, we will now adopt the following law:

> **Law 1:** the specification of the database schema precedes that of the grammar.

When we considered the issue of translating from a database to a file, we saw that it is not immediate to obtain the "inverse structuring schema", i.e., the structuring schema which would reconstruct the database from the file. However, as mentioned above, this is possible if we restrict our attention to a few system supplied (eventually customizable) encoding schemes.

So, we will also adopt the following law:

> **Law 2:** the specification of the database schema should include choices of encodings of the database constructs occurring in it.

Observe that by these two laws, the structuring schema is now automatically generated. In particular,

> **Consequence:** the grammar is a by-product of the specification of the database.

Note that this does not prevent from considering applications such as BibTex where we have to load data from a file. However, we do not start by specifying a grammar. Instead, we specify the intended database schema and encodings which should yield the underlying grammar. It is now necessary to experiment these ideas to see whether this would entail unacceptable limitations compared to the direct specification of the grammar.

The optimization techniques described in the paper are of course still relevant. Furthermore, with this approach, we can support some nontrivial updates since the system is now aware of one-to-one transformations between subdatabases and substrings. Indeed, we believe that this global *database-driven* approach to the couple (database $\leftrightarrow$ structured string) is the only serious answer to the problems raised by updates. Clearly, this requires further work and, it is important, in particular, to elaborate on the work of Section 6.

Acknowledgment: We thank Sophie Gamerman, Bernard Lang, and Laurent Herr for discussions on the topic. Laurent is also thanked for implementing O_2-Yacc.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM Sigmod Conference*, 1991.

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

[3] A. V. Aho and S. C. Johnson. Programming utilities and libraries lr parsing. *Computing Surveys*, June 1974.

[4] 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.

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

[6] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. 11th Symp. on Principles of Database Systems - PODS, San-Diego*, 1992.

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

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

[9] U. Dayal. Queries and views in an object-oriented data model. In *Proc. Internat. Workshop on DBPL*, 1989.

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

[11] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory Languages and Computation.* Addison-Wesley, 1979.

[12] W. Kim. A model of queries for object-oriented databases. In *Proc. VLDB*, 1989.

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

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

[15] 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.

[16] 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.

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