On Implementing *SchemaLog* – A Database Programming Language

Alanoly J. Andrews
Concordia University, Montreal, Canada
alanoly@cs.concordia.ca

Laks V.S. Lakshmanan
Concordia University, Montreal, Canada
laks@cs.concordia.ca

Nematollaah Shiri
Concordia University, Montreal, Canada
shiri@cs.concordia.ca

Iyer N. Subramanian
Concordia University, Montreal, Canada
subbu@cs.concordia.ca

Abstract

Efficient implementation of advanced database programming languages call for investigating novel architectures and algorithms. In this paper, we discuss our implementation of *SchemaLog*, a logic-based database programming language, capable of offering a powerful platform for a variety of database applications involving data/meta-data querying and restructuring. Our architecture for the implementation is based on compiling *SchemaLog* constructs into an extended version of relational algebra called *SchemaLog*. Based on this algebra, we develop a top-down algorithm for evaluating *SchemaLog* programs. We discuss three alternative storage structures for the implementation and study their effect on the efficiency of implementation. For each storage structure, we propose strategies for implementing our algebraic operators. We have implemented all these strategies on top of Microsoft Access DBMS running on Windows 3.1, and have run an extensive set of experiments for evaluating the efficiency of alternative strategies under a varied mix of querying and restructuring operations. We discuss the results of our experiments and conclude with a discussion of a graphic user interface for SchemaLog program development, that has also been implemented.

1 Introduction

This paper reports on our implementation of *SchemaLog*, a powerful language for advanced database logic programming proposed by Lakshmanan et al. [LSS93, LSS96]. It was established there that *SchemaLog* can offer a powerful platform in a variety of settings including multi-database *interoperability*, *database programming* with *schema browsing*, *cooperative query answering*, computing forms of *aggregate queries* which are beyond the scope of conventional database query languages, and *database restructuring*. *SchemaLog* has a higher-order syntax, but unlike some of the previous languages like COL [AG87], LDL [Chi89], etc, and like HiLog [CKW93] and (a fragment of) F-logic [KLW95], *SchemaLog* has a first-order semantics. Indeed, it has a sound and complete proof theory [LSS96]. A prototype platform for interoperability among a number of INGRES databases was completed recently [LSPS95], based on a fragment of *SchemaLog*. The theoretical basis for the use of *SchemaLog* as a powerful database programming language even in the context of a *single database* was established in our earlier papers. In this paper, we establish the *practical basis* for this claim by describing efficient architectures and algorithms for the implementation of *SchemaLog* as a database programming language, in the context of a *single database*.

In the following, we introduce the syntax of *SchemaLog* and illustrate the semantics informally via examples. The syntax we will use is an adaptation of the full *SchemaLog* given in [LSS96] to a single database context. For a formal account, the reader is referred to [LSS96]. *SchemaLog* features three[1] kinds of basic expressions, called *atoms*. Their notation as well as an informal meaning is given below.

- $\langle rel \rangle$ – there is a relation named $\langle rel \rangle$ in the database.
- $\langle rel \rangle[\langle attr \rangle]$ – there is a relation named $\langle rel \rangle$ in the database, and the schema of $\langle rel \rangle$ includes an attribute named $\langle attr \rangle$.
- $\langle rel \rangle[\langle tid \rangle : \langle attr \rangle \rightarrow \langle val \rangle]$ – there is a relation named $\langle rel \rangle$ in the database, and the schema of $\langle rel \rangle$ includes an attribute named $\langle attr \rangle$, and furthermore $\langle rel \rangle$ contains a tuple $\langle tid \rangle$ which has value $\langle val \rangle$ under column $\langle attr \rangle$.

In the above, $\langle rel \rangle$, $\langle tid \rangle$, $\langle attr \rangle$, and $\langle val \rangle$ are arbitrary first-order terms, built up as usual from a vocabulary including constants, function symbols, and variables. In this paper, for simplicity, we will concentrate on function-free *SchemaLog*. *Molecules* are expressions of the form $\langle db \rangle :: \langle rel \rangle[\langle tid \rangle : \langle attr \rangle_1 \rightarrow \langle val \rangle_1, \ldots, \langle attr \rangle_n \rightarrow \langle val \rangle_n]$ and are a syntactic sugar for the conjunction $\langle rel \rangle[\langle tid \rangle : \langle attr \rangle_1] \wedge \cdots \wedge \langle rel \rangle[\langle tid \rangle : \langle attr \rangle_n \rightarrow \langle val \rangle_n]$. The $\langle tid \rangle$ acts as a "glue" to combine the pieces from different atoms together. Existential "don't care" tid variables appearing in rule bodies can be omitted (see Example 1.1). Besides these, *SchemaLog* also uses *programming predicates* of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol and t_i are terms. While programming predicates can be simulated using molecules, they add to programming convenience. We use the generic term *database predicates* to refer to *SchemaLog* atoms and molecules. The main

*This research was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and the Fonds Pour Formation De Chercheurs Et L'Aide À La Recherche of Quebec.

[1] Actually, full *SchemaLog* features four. In a single database context, these reduce to three.

difference between database predicates and programming predicates is that the former are used as an "interface" to relations - either existing in a database (i.e. the so-called EDB relations) or created by database programs written in *SchemaLog* (i.e. the so-called IDB relations) - whose schematic information (i.e. names of relations and their attributes) is regarded just as important as data, and needs to be queried and manipulated. On the other hand, programming predicates are used as a convenient device for storing intermediate results in computations, or sometimes for storing results of queries, where no particular attention is paid to the schema under which such results are stored. The term *database relations* refers to relations corresponding to database predicates, while *programming relations* refer to relations corresponding to programming predicates. When no confusion arises, we refer to database relations, simply as relations.

[LSS96] provides a number of examples to illustrate the semantics of *SchemaLog* programs, and the various terminologies and conventions introduced above. For lack of space, we provide just one example in this section.

Example 1.1 Throughout the paper, we shall use the database pertaining to the New York Stock Exchange, shown in Figure 1, as a running example. The data is based on the *actual* data maintained at the URL http://www.ai.mit.edu/stocks.html.

The following is a *SchemaLog* program that restructures the above database such that the restructured database contains, corresponding to each index such as *high*, *low* etc, a relation with scheme $\langle date, name, price\rangle$ that stores the value of the index for all dates for various companies.

$A[date \rightarrow D, name \rightarrow N, price \rightarrow P] \longleftarrow$
$stocks[name \rightarrow N, ticker \rightarrow T], T[date \rightarrow D, A \rightarrow P], A \neq \text{'}date\text{'}$

Contributions of this paper: Implementation of *SchemaLog* as a DBPL in the context of a single database is the major goal of this research. In this paper, we develop a top-down approach for implementing *SchemaLog*. For database applications, it is important that queries and restructuring be implemented in a set-oriented manner, as opposed to the tuple-at-a-time paradigm of Prolog. Our contributions include the following. (1) We develop an approach based on compiling constructs in *SchemaLog* into operations in an extended relational algebra (Section 2). (2) We discuss algorithms for top-down implementation of *SchemaLog* and establish their correctness (Section 3). (3) Since the schema components of relations are given first-class status, this calls for efficient storage structures. We discuss three alternative storage structures and associated implementation strategies for the algebraic operators (Section 4). (4) These strategies have been implemented on top of Microsoft Access DBMS. In order to assess the performance of alternative strategies, we ran a number of experiments. The experiments as well as their outcome are discussed in Section 5. (5) We have developed a graphic user interface for our implementation. This is described in Section 6. Conclusions and a comparison with some related implementations are given in Section 7.

For lack of space, details of some of our algorithms as well as many additional examples illustrating features of *SchemaLog* and the extended algebra are not included in this paper. Readers may refer to [ALSS96] for the details.

2 Algebra

Our approach to the implementation of *SchemaLog* is based on compiling constructs in *SchemaLog* into corresponding operations in an extended version of relational algebra, called *Schema Algebra*, or $\mathcal{SA}$ for short. Our algebra includes conventional relational algebra as a proper subset and features extensions that facilitate meta-data querying and restructuring. Specifically, $\mathcal{SA}$ consists of the following kinds of operations.

1. Classical RA operators: these are capable of querying programming relations.
2. Operators which query the data and schema of (database) relations and present the output as programming relations. Thus, they map relations in a database to programming relations.
3. Operators which take as input programming relations (and some parameters) and structure the information in them in specified ways. Thus, they map programming relations into (database) relations.

Operators of type (2) and (3) are new and are unique to our algebra. In principle, our algebra can well be defined in the context of a federation of databases. Indeed, operations of type 1 and 2 were defined in such a context in [LSS96] and it was proved that they have an expressive power equivalent to that of *SchemaLog* programs containing only programming predicates in rule heads. Before presenting the definitions of the operators, we remark that in classical RA, one can refer to attributes (which are schema components) whether by position or by name. However, this does not mean that their schema is given a first class status. The point is that schema information cannot be retrieved nor restructured using classical RA.

2.1 Definitions of Operators

The definitions of classical RA operators are as usual: no modifications are necessary. We next define type (2) operators.

Definition 2.1 (Fetching relation names) *The first operator is a 0-ary operator. It returns as output the* names *of all relations in the database. Formally,* $\rho = \{r \mid r$ *is the name of a relation in the database*$\}$.

E.g., $\rho()$ would return the set $\{stocks,\ ibm,\ msft,\ xon\}$, w.r.t. the stock market database of Figure 1. Notice that the output of $\rho()$ includes both base and derived (database) relations, in general.

Definition 2.2 (Fetching relations and their schemas) *The second operator takes a programming relation as input and a column number as parameter. It then interprets the values appearing in that column as possible relation names in the database, and retrieves the names of their attributes. More formally, let* s *be a programming relation of arity* k *and* $i \leq k$ *a positive number. Then* $\alpha_i(s) = \{(r, a) \mid r \in \pi_i(s)$, r *is the name of a relation in the database, and* r *has an attribute with name* $a\}$. *Note that* $\pi_i(s)$ *denotes the classical projection.*

Stocks

Name	Ticker	Type
Exxon Corp	xon	oil
IBM Corp	ibm	computer
Microsoft Corp	msft	computer

ibm

Date	High	Low	Close	Volume
95/06/16	93.62	92.62	92.62	2696.8
95/06/19	94.62	93.25	94.62	2078.7
95/06/20	98.00	93.87	97.75	3568.9
95/06/21	98.62	97.12	97.12	3580.5

xon

Date	High	Low	Close	Volume
95/06/16	72.25	71.37	72.12	3420.6
95/06/19	72.25	71.50	71.50	826.1
95/06/20	71.37	70.00	70.12	1267.3
95/06/21	70.00	68.75	69.25	1792.2

msft

Date	High	Low	Close	Volume
95/06/16	87.50	84.87	87.00	5767.5
95/06/19	89.87	86.87	89.62	4410.1
95/06/20	91.37	89.75	91.37	3541.9
95/06/21	92.37	90.00	90.50	3583.8

Figure 1: The NYSE Database

As an example, suppose $s = \{(msft,\ micsrosoft,\ 100), (hp, hewlett\text{-}packard, 200)\}$. Then $\alpha_1(s) = \{(msft,\ date), (msft,\ high), (msft,\ low), (msft, close), (msft,\ volume)\}$. Notice that no output corresponding to hp is produced since hp does not correspond to any relation name in the database.

Before presenting the definition of our next operator, we need the notion of a *pattern*, introduced in [LSS96] in a different context. A pattern is of one of the following forms: '$a{\rightarrow}v$', '$a{\rightarrow}$', '${\rightarrow}v$', or '$\rightarrow$'. Intuitively, a pattern may be viewed as an attribute value pair, where either the attribute or the value component (or both) of the pair could be missing. Our next operator uses a pattern to query data and meta-data in a database. This is achieved via a notion of *satisfaction*. Let r be a relation, t a tuple in r, and p a pattern. Then we say t satisfies p provided one of the following conditions holds:

- p is of the form $a{\rightarrow}v$, and r has a as one of its attributes and $t[a] = v$. In this case, the attribute value pair (a, v) is said to be a *witness pair*.
- p is of the form $a{\rightarrow}$, and r has a as one of its attributes. In this case, the attribute value pair (a, v) where $t[a] = v$, is said to be a *witness pair*.
- p is of the form ${\rightarrow}v$, and there is some attribute b in the schema of r, such that $t[b] = v$. In this case, for every attribute b for which $t[b] = v$, (b, v) is said to be a *witness pair*.
- p is of the form $\rightarrow$. In this case, for every attribute a of r, and every value v such that $t[a] = v$, (a, v) is a *witness pair*.

Definition 2.3 (Querying data and meta-data) *This operator allows us to relate data to meta-data. It takes as input a programming relation, a column number and a pattern as parameters, and returns as output the details of all tuples in the database which satisfy the pattern. More precisely, let s be any programming relation of arity k, $i \leq k$ any number, and let p be any pattern. Then $\gamma_{i;p}(s) = \{(r, t, a, v) \mid r \in \pi_i(s), t$ is the id of a tuple that satisfies p, and (a, v) is an associated witness pair$\}$.*

E.g., let $s = \{xon, ibm\}$. Then $\gamma_{1;high\rightarrow}(s) = \{(xon, t4, high, 72.25), \ldots, (xon, t7, high, 70.00), (ibm, t8, high, 93.62), \ldots, (ibm, t11, high, 98.62)\}$.

We next introduce an operator, which can be derived from the previous operators. The main motivation for this operator is query processing efficiency. This operator behaves essentially the same as γ except that it does not explicitly extract tuple ids, and it deals with a conjunction of patterns in one go. We denote a conjunction of patterns as $\langle p_1, \ldots, p_n\rangle$, where each p_i is a pattern. Satisfaction of conjunctions of patterns is defined in the obvious manner: a tuple satisfies a conjunctive pattern if it satisfies all patterns in the conjunction.

Definition 2.4 (Querying conjunctive patterns) *This operator takes a programming relation of arity k as input, a column number $i \leq k$ and a conjunctive pattern as parameters, and returns as output the details of all parts of the database queried about using the conjunctive pattern. More formally, let s be a programming relation, i a column number, and $\langle p_1, \ldots, p_n\rangle$ a conjunctive pattern, where each p_j is of one of the the forms – '$a{\rightarrow}v$', '$a{\rightarrow}$', '${\rightarrow}v$', or '$\rightarrow$'. Then $\gamma^{\wedge}_{i;\langle p_1,\ldots,p_n\rangle}(s) = \{(r, a_1, v_1, \ldots, a_n, v_n) \mid r \in \pi_i(s), r$ is the name of a relation in the database, a_j's are attributes in the schema of r, and there is a tuple $t \in r$ such that $t[a_1, \ldots, a_n] = v_1, \ldots, v_n$, and t satisfies $\langle p_1, \ldots, p_n\rangle\}$.*

E.g., let $s = \{stock\}$. Then for the stock market database, $\gamma^{\wedge}_{1;\langle ticker\rightarrow, \rightarrow computer\rangle}(s) = \{(stock, ticker, ibm, type, computer), (stock, ticker, msft, type, computer)\}$.

This completes the definition of operators that extract the information in relations and convert it into programming relations.

We next turn to the type (3) (restructuring) operators.

Definition 2.5 (Creating relations) *The first operator takes a programming relation of arity k as input and a column number $i \leq k$ as a parameter and creates relations with names corresponding to the entries appearing in column i of the input relation. More formally, $\kappa_i(s)$, for a programming relation s and a column number i creates a* *relation named r for each $r \in \pi_i(s)$, if such a relation does not already exist.*

E.g., $\kappa_1(\{(close), (high), (low), (volume)\}$ creates the relations *close*, *high*, *low*, *volume* (whose schema is not yet defined).

Definition 2.6 (Creating relations with schemas) *The second operator takes a programming relation of arity k as input and two column numbers $i, j \leq k$ as parameters. It creates relations with names corresponding to the entries in column i whose schemas are determined by interpreting the entries appearing column j as the attributes associated with the relation names in column i. More formally, for a programming relation s and column numbers i, j, $\varsigma_{i,j}(s)$ creates a relation r with attributes $a_1, \ldots, a_n$ exactly when $\sigma_{\$1='r'}(\pi_{i,j}(s)) = \{(r, a_1), \ldots, (r, a_n)\}$, whenever such a relation does not already exist. Note that $\sigma_{\$1='r'}$ denotes classical selection [Ull89].*

E.g., let $s = \{(close, date), (close, ibm), (close, msft), (close, xon), (volume, date), (volume, ibm), (volume, msft), (volume, xon)\}$. Then $\varsigma_{1,2}(r)$ will create two relations *close* and *volume* both with the schema $\{date, ibm, msoft, xon\}$, and with no data.

Definition 2.7 (Creating and populating relations with schemas) *The last operator takes a programming relation of arity k as input, three column numbers i, j, k and another list of column numbers $g_1, \ldots, g_m$ as parameters and returns as output several relations structured according to the interpretation of column i entries as relation names, column j entries as attribute names, and column k entries as values. Facts so generated form pieces of larger tuples. The grouping for forming output tuples is determined based on equality on the columns $g_1, \ldots, g_m$. More formally, $\varrho_{i,j,k;g_1,\ldots,g_m}(s)$ creates a relation r with attributes $a_1, \ldots, a_n$ exactly when $\sigma_{\$1='r'}(\pi_{i,j}(s)) = \{(r, a_1), \ldots, (r, a_n)\}$, whenever such a relation does not already exist. Furthermore, it populates the relation r with a tuple t such that $t[a_1, \ldots, a_n] = \langle v_1, \ldots, v_n \rangle$ exactly when $\exists t_1, \ldots, t_n \in r$ such that $t_i[i, j, k] = (s, a_i, v_i)$, $\pi_{i,j,k}(\{t_1, \ldots, t_n\}) = \{(s, a_1, v_1), \ldots, (s, a_n, v_n)\}$, and finally, $t_1[g_1, \ldots, g_m] = \cdots = t_n[g_1, \ldots, g_m]$. When the relation s already exists, the above mentioned tuples are appended to this relation.*

E.g., let $r = \{(close, ibm, 95/06/21, 97.12), (close, msft, 95/06/21, 90.50), (close, xon, 95/06/21, 69.25), (close, ibm, 95/06/20, 97.75), (close, msft, 95/06/20, 91.37), (close, xon, 95/06/20, 69.25), (close, date, 95/06/21, 95/06/21), (close, date, 95/06/20, 95/06/20)\}$. Then $\varrho_{1,2,4;3}(r)$ will create the relation shown in Figure 2.

Close

date	ibm	msft	xon
95/06/20	97.75	91.37	69.25
95/06/21	97.12	90.50	69.25

Figure 2: Example of Restructuring

We remark that our restructuring operations also have limited update capabilities in the sense that whenever the relation corresponding to the restructured form already exists, newly generated data is appended to such a relation. E.g., suppose that the relation *close* shown in Figure 2 already exists (say, because of an invocation of operation ϱ. Let $s' = \{(close, ibm, 95/06/19, 96.12), (close, msft, 95/06/19, 90.50), (close, xon, 95/06/19, 70.25), (close, ibm, 95/06/19, 99.75), (close, date, 95/06/19, 95/06/19)\}$. Then $\varrho_{1,2,4;3}(s')$ will have the effect of *appending* the tuple (5/06/19, 99.75, 90.50, 70.25) to the *existing* relation *close*.

3 Top-down Processing of *SchemaLog* Programs

In this section we discuss the top-down processing of *SchemaLog* programs. In particular, we investigate how the set oriented Rule/Goal Tree (RGT) evaluation method [Ull89] proposed for classical logic can be extended to the *SchemaLog* setting. Our choice of this methodology is due to the fact that set-oriented query processing techniques are more suitable for database applications as opposed to the tuple-at-a-time paradigm of Prolog. At the guts of the algorithm we discuss here, lie the $\mathcal{SA}$ operators defined in the previous section.

The notion of unification plays an important role in the construction of RGTs. Unification in *SchemaLog* is different from its classical counterpart. [LSS96] discusses this issue at depth and presents an algorithm for computing the most general unifier (MGU) of two *SchemaLog* atoms. Based on the *SchemaLog* notion of unification, the conventional algorithm for constructing the RGT of a program can be easily adapted to our setting. An important consequence of the fact that *SchemaLog* unification is performed on atoms is that we need to 'atomize' a *SchemaLog* program (which in general might contain molecules) before applying the top-down algorithm.We refer the reader to [ALSS96] for the details of the algorithm for atomization.

The major strength of *SchemaLog* lies in its ability to express novel querying as well as powerful restructuring operations. Our adaptation of the classical RGT evaluation algorithm accounts for these unique features of *SchemaLog* . In the following, we sketch the major issues that arise in the development of such an algorithm. In [ALSS96], we present a comprehensive algorithm for the RGT evaluation of a *SchemaLog* program and establish its correctness.

Two operations are often invoked during the top-down evaluation of *SchemaLog* programs — one for converting the database relations corresponding to (programming) subgoals to relations over variables mentioned in that subgoal, and the other for converting a (programming) relation for the body to a relation for the head by translating from the viewpoint of variables to the viewpoint of arguments. For datalog, this switching between argument and variable viewpoints is accomplished by means of procedures called A2V() and V2A() [Ull89]. As *SchemaLog* atoms are syntactically different from their classical counterpart, the A2V() and V2A() procedures are somewhat different for our setting. Our approach efficiently realizes these operations by using the technique of first reducing the database predicate argument of the operation to a template that corresponds to a conventional predicate, and then applying the classical version of the operations. For instance, in our A2V() procedure, in order to convert a relation M into a relation whose attributes correspond to variables appearing in a *SchemaLog* atom $\mathcal{A}$ of the form $\alpha_1[\alpha_2 : \alpha_3 \rightarrow \alpha_4]$, $\mathcal{A}$ is reduced to a template $temp(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ and the conventional A2V() algorithm with the template and M as arguments is applied. The adaptation of the V2A() algorithm is similar in nature.

At the heart of the RGT evaluation algorithm lies two mutually recursive procedures - EXPAND_GOAL() and EXPAND_RULE(). Given a *SchemaLog* goal G and a relation M that provides bindings for variables in G, EXPAND_GOAL() returns a relation R that is the set of tuples (bound by M) that match G and can be inferred from the database using the program. EXPAND_RULE() on the other hand, takes a rule r and initial binding for variables in this rule and generates a relation R that is the set of tuples inferred from the database and the rule. Further, this procedure performs restructuring dictated by R and the head predicate of rule r. Thus the querying and restructuring facets of *SchemaLog* query processing are neatly decoupled in procedures EXPAND_GOAL() and EXPAND_RULE() respectively. EXPAND_GOAL() invokes the querying operations in $\mathcal{SA}$ via a procedure called QUERY_GOAL(). Corresponding to each type of *SchemaLog* atom, this procedure invokes an appropriate $\mathcal{SA}$ expression involving type (2) operations. For instance, the call QUERY_GOAL($X[T : a \rightarrow V]$) invokes the operation $\sigma_{\$3='a'}\gamma_{\$1;\rightarrow}(\rho)$. Procedure RESTRUCTURE_HEAD(), called from EXPAND_RULE(), invokes an $\mathcal{SA}$ expression that includes a type (3) operation corresponding to the head predicate of the *SchemaLog* rule under expansion. These two procedures are presented in Algorithm 3.1.

```
Algorithm 3.1
procedure QUERY_GOAL($A$)
Input: A SchemaLog atom of the form $\beta_1[\beta_2 : \beta_3 \rightarrow \beta_4]$,
or an atom of a lesser depth.
  begin
    case A is of depth:
    1 : if $\beta_1$ is a constant, return $\sigma_{\$1=\beta_1}(\rho)$
          else return $\rho$;
    2: return $\sigma_{\wedge_i \$_i=\beta_i}\alpha_{\$1}(\rho)$, $\beta_i$ is a constant;
    3: return $\sigma_{\wedge_i \$_i=\beta_i}\gamma_{\$1;\rightarrow}(\rho)$, $\beta_i$ is a constant
end

procedure RESTRUCTURE_HEAD($A, P$)
Input: A SchemaLog atom of the form $\beta_1[\beta_2 : \beta_3 \rightarrow \beta_4]$, or
an atom of a lesser depth, and a relation P whose attributes
correspond to variables in A.
  Note: The unary function $\nu$ used below, takes as
argument a variable appearing in A and returns the position
of its corresponding attribute in P.
  begin
    case A is of depth
    1: if $\beta_1$ is a constant, return $\kappa_1\{\beta_1\}$
          else return $\kappa_{\nu(\beta_1)}(P)$;
    2: form a tuple $t =< \beta_i, \ldots, \beta_k >, 1 \leq i \leq k \leq 2$,
              $\beta_i, \ldots, \beta_k$ are all the constants in A;
          compute $Q = P \times \{t\}$;
          return $\varsigma_{\nu(\beta_1),\nu(\beta_2)}(Q)$;
    3: form a tuple $t =< \beta_i, \ldots, \beta_k >, 1 \leq i \leq k \leq 4$,
              $\beta_i, \ldots, \beta_k$ are all the constants in A;
          compute $Q = P \times \{t\}$;
          return $\varrho_{\nu(\beta_1),\nu(\beta_3),\nu(\beta_4);\nu(\beta_2)}(Q)$
  end
```

As in the classical case, a queue-based version of this algorithm based on a breadth-first search of the RGT can be realized by queuing the calls to EXPAND_GOAL() and EXPAND_RULE() rather than stacking them.

4 Physical Storage Architectures

As illustrated in the preceding sections, *SchemaLog* possesses powerful capabilities for querying data and meta-data of relations as well as for restructuring them. Supporting these features in an implementation requires efficient storage structures. Recall that *SchemaLog* is implemented by compiling its constructs into appropriate operations in Schema Algebra. Clearly, depending on the chosen storage structures, the underlying implementation strategy for the algebraic operations would differ. In this section, we outline three alternative storage structures at the level of physical schemas. We also discuss the implementation of $\mathcal{SA}$ operators corresponding to each of them. For lack of space, we discuss only some of the implementation algorithms. The rest can be found in [ALSS96].

Before we present the alternative storage structures, we remark that for existing (i.e. base) database relations, it is unrealistic to suppose that they can be converted into any form other than their existing form. For one thing, such a conversion would incur a massive overhead. For another, this would disrupt applications running on the existing database. [LSS96] discusses these issues in detail and argues that from a practical perspective, the base relations should be preserved in their existing form. Thus, we are really considering alternative storage structures for database relations which are *created* or *derived* by *SchemaLog* programs.

1. Conventional Storage:

The idea is to use the same schema as for conventional database relations. In other words, a relation r with attributes A, B, C would be implemented as a file of records with those fields. Thus, derived database relations would be stored and accessed the same way as base relations are. Algorithm 4.1 implements the ϱ operator in $\mathcal{SA}$ in this scenario.

```
Algorithm 4.1 $\varrho_{i,j,k;l}(r)$
begin
    while there are no rows left (in the given schemaless
            relation)
        read a row
        store the relation name, attribute name, value
            and grouping attribute value in an
            appropriate data structure
    endwhile
    for each relation name rel in the stored relation
        create a table called temp_rel with the
            appropriate (stored) attributes
        append tuples to the table (using the
                stored values)
    endfor
    for each temp_rel created
        if $\exists$ a relation named rel in the database
            rel = rel $\bowtie$ temp_rel
        else create a relation rel = temp_rel
    endfor
end
```

Since classical RA operations are essentially tailored for conventional storage, we expect that these operations will have the most efficient implementation under conventional storage. By contrast, conventional storage is not particularly suited for meta-data querying and restructuring. Thus, we expect the new strategies proposed (see below) to perform better for such operations.

2. Reduced Storage:

The term *reduced* refers to the fact that *SchemaLog* admits a faithful first-order reduction, as established in [LSS96]. The idea is that each relation in the database can be "flattened" and all the information in the database can be compiled into three relations - $call_4(R, T, A, V)$ (corresponding to $R[T : A \rightarrow V]$), $call_2(R, A)$ (corresponding to $R[A]$), and $call_1(R)$ (corresponding to $R[]$). As pointed out earlier, in a practical setting, this flattening can only be applied to derived database relations. E.g., for the derived relation *aeps* of Example 1.1, the corresponding reduced storage would be $call_4 = \{(aeps, t1, quarter, I), (aeps, t1, ticker, xon), \ \ldots\}$, $call_2 = \{(aeps, quarter), (aeps, ticker), \ \ldots\}$, $call_1 = \{aeps, asp\}$. Under reduced storage, meta-data querying is essentially reduced to conventional data querying, and restructuring is reduced to updating the $call_i$ relations. In other words, the extended operations of type (2) and (3) in $\mathcal{SA}$ are reduced to classical RA operations, under this storage. In particular, piecemeal computation under conventional storage reduces to normal computation (where entire tuples are computed at a time, rather than in parts). By contrast, simple classical operations like selection and join translate into complex operations against this storage. Algorithm 4.2 illustrates the implementation of the ϱ operator.

```
Algorithm 4.2 $\varrho_{i,j,k;\ell}(r)$
begin
    while tuples remain in r
        read tuple r from r.
        form a tuple $< r, t, a, v >$ such that r, t, a, and
            v are the $i^{th}$, $\ell^{th}$, $j^{th}$, and $k^{th}$
            components respectively of r.
        write the new tuple.
    endwhile
end
```

3. Reduced, Atomized Storage:

A derived database relation of the form $r(a_1, \ldots, a_n)$ is physically stored in relations $physrel(r, a_1)(tid, val), \ldots,$ $physrel(r, a_n)(tid, val)$. Note that in this storage scheme, $physrel(r, a_j)$ is the *name* of a relation used for physical storage while $\{tid, val\}$ is its schema. The first column in a physical relation corresponds to the tuple-ids of tuples in the database relation r, and the second column contains the values. The tuple $< i, v >$ in a physical relation $physrel(r, a_j)$ represents the fact that a tuple i in relation r has value v on attribute a_j.

Thus, in this strategy, a derived database relation is stored using as many physical relations as there are attributes in it - each such relation storing one column of the database relation. The relation name and the attribute name corresponding to the column are 'encoded' in the name of the physical relation. The $\mathcal{SA}$ operations are interpreted against such a representation; for instance, operations that add attributes to an existing relation translate in this strategy to operations that add new relations. Many of the comments made for reduced storage also apply to reduced atomized storage. Thus, we expect that this scheme will suit meta-data querying and restructuring better than conventional data querying. Algorithm 4.3 explains how the ϱ operator is implemented in this strategy.

```
Algorithm 4.3 $\varrho_{i,j,k;\ell}(r)$
begin
    for each tuple in the input relation r
        read tuple t.
        add tuple $< t[\ell], t[k] >$ to the relation
            $physrel(t[i], t[j])$
        (create the relation if it does not exist).
    endfor
end
```

5 Experiments and Analysis

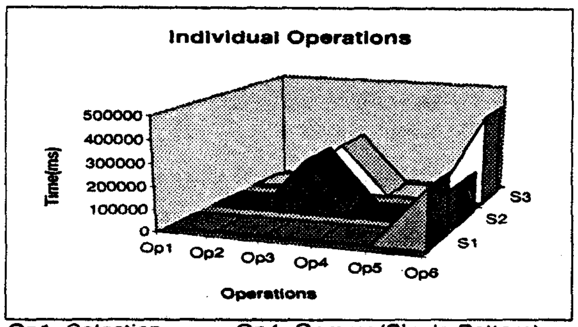
In this section we present the conclusions from the many experiments that we have conducted on implementing $\mathcal{SA}$ querying and restructuring operations for the various strategies. The experiments were run on MS Access DBMS on PC/Windows platform. Our experiments cover a broad range of scenarios, with number of tuples ranging from 500 to 10,000 and the join density varying from 0.25 to 0.5. For lack of space we do not include here all the charts illustrating the results of our experiments. Figures 3 and 4 may be considered as representatives. A more comprehensive set of charts is included in [ALSS96].

5.1 Experimental Results: Individual Operations

In this experiment, we studied the cost of implementing each of the $\mathcal{SA}$ operations. Figure 3 is a sample graphical representation of one of the experiments (done on two tables of 10,000 tuples each, with a Selection Density of 3 and a Join Density of 0.25).

As can be seen from the figure, the conventional storage strategy (S1 in the figure) seems to be superior for the purely querying operations. The peak in the middle of the plot for the reduced strategy (S2) can be attributed to the extra cost involved in the join operation for this strategy. This is because the join is made only on a fragment of the original "conventional" tuple and the full tuples have to be reassembled once the results of the join are known.

A similar explanation can be given for the peak in the join plot for the "reduced and atomized" (S3) storage. It



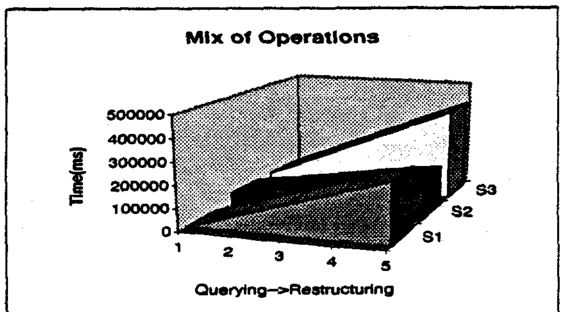
Op1=Selection	Op4=Gamma(Single Pattern)
Op2=Projection	Op5=Gamma(Multiple Pattern)
Op3=Join	Op6=Var-Rho

Figure 3: Individual Operations

must be noted that restructuring costs for S2 are low. The peak for restructuring in S3 (the plot for reduced and atomized storage) can be explained by the fact that several data sets corresponding to the various tables being created/modified have to be successively opened and closed as tuples from the input programming relation are being processed. Opening/closing of data sets is an operation that does incur some overhead.

5.2 Experimental Results: Mix of Operations

Processing a *SchemaLog* program typically consists of a mixture of various operations from $\mathcal{SA}$. To simulate this scenario, we have conducted experiments on combining the operations in varying proportions. The mixes we have investigated are: (1) 100% querying operations (2) 75% querying and 25%restructuring operations (3) 50% querying and 50% restructuring operations (4) 25% querying and 75%restructuring operations, and (5) 100% restructuring operations. In each of the above combinations, the load for querying is distributed equally among the querying operations in $\mathcal{SA}$ and the load for restructuring distributed likewise. It is our belief that such a study would better reveal the performances of the various strategies than specifically chosen programs would.



1=100% query	4=25%query,75%res
2=75% query, 25% restruct	5=100% restruct
3=50% query, 50% restruct	

Figure 4: Mix of Operations

Figure 4 illustrates the results obtained from one such experiment (on the same database used for figure 3). We see that in an application used predominantly for querying operations, the conventional storage strategy is superior. Querying costs for the reduced storage strategy are somewhat higher. But as we move towards a greater mix of restructuring operations, we see that *the cost for conventional storage strategy keeps on increasing whereas the cost for the reduced strategy remains fairly constant.* In a *SchemaLog* application that has more restructuring operations than querying operations, the preferred strategy should be the reduced strategy. A mixture of the two strategies, in which the conventional storage is used for base relations and the reduced strategy for derived relations would appear to be appropriate. In this context it is worth noting that many emerging database applications such as online analytical processing (OLAP) technology ([CCS95]) are concerned with considerable amount of restructuring operations.

In summary, we conclude that the conventional storage strategy is superior for applications that have a higher ratio of querying to restructuring operations, the reduced strategy for applications in which restructuring predominates. In general, for *SchemaLog* applications, our experimental results recommend a mixture of the two strategies where the base relations are stored in the conventional form and derived relations in the reduced form.

6 User Interface

In this section we discuss a typical UI system that provides an environment for developing *SchemaLog* applications. Our discussion is based on a system we have developed in the context of a prototype platform for interoperability based on a fragment of *SchemaLog* [LSPS95]. We have designed and implemented the UI using the UIM/X interface development toolkit [Vis93].

Aside from the usual text editor, file selection box, and related facilities for preparing *SchemaLog* programs, the other features of the UI are as follows. (1) *Schema browsing*: Our system provides a natural navigation of the schema hierarchy. For instance, the invocation of `schema browse` returns the list of relations in the database; clicking on one of the relations lists the attributes in that relation and so on. Interestingly, the schema browsing capability in the UI is realized using an underlying *SchemaLog* program. (2) *Ad hoc querying*: The implementation of the *SchemaLog* system uses CORAL [RSS92] as a back-end. Our UI exploits the ad hoc querying features of CORAL by providing *querying window widgets* in the interface. The use of CORAL, however, is hidden from the user. The UI also has a *recall* feature with which the user can scan the queries already posed and reuse them to formulate new queries. (3) The many features that the UI provides, lets the user initiate several concurrent processes. We have used the process handling facilities of UIM/X to synchronize among these processes in a clean way.

7 Comparison and Summary

In this section, we compare our work with similar work related to the implementation of other higher-order logic database languages. We specifically consider three such implementations: (i) implementation of F-logic [KLW95], one of the most comprehensive logical accounts for the object-oriented model; [Law93]'s implementation is useful for running small F-logic programs, but it is not clear how this can be used in a real database context. (ii) implementation of Gulog, an object-oriented logic developed at Griffith University, Australia [Dob95]; [Lef93]'s implementation bootstraps on the implementation of the F-logic interpreter above and inherits its limitations. (iii) implementation of HiLog, a higher-order database logic programming language developed at SUNY, Stony Brook [CKW89]. [SW95] describes an efficient implementation of HiLog within the WAM(Warren Abstract Machine) and is based on using a first-order translation of HiLog. For a comparison of the languages themselves, interested readers are referred to [LSS96].

In contrast with all the above implementations, our implementation of *SchemaLog* has the following unique features. (1) It is not based on translation into any other language like Prolog. Rather, our implementation is a direct one. (2) Schema Algebra is at the core of our implementation. This is especially suited for set-oriented processing which is more appropriate for a database context as opposed to a logic programming context. (3) To our knowledge, issues like meta-data querying and piecemeal computation have not been dealt with in previous work. (4) We proposed alternative strategies for dealing with these challenging issues, and evaluated their effectiveness with a series of experiments.

In conclusion, we investigated the major issues in the implementation of a database programming language based on *SchemaLog* . We proposed an architecture for the implementation, based on compiling *SchemaLog* constructs into an extended version of RA called *SA* . We addressed challenging issues unique to *SchemaLog* implementation and proposed three alternative storage structures for dealing with them. We also proposed algorithms for top-down implementation of *SchemaLog* , including alternative strategies for the implementation of the algebraic operators. We evaluated the effectiveness of the alternative strategies with a series of experiments on top of MS Access. From practical considerations and from the experiments, a viable approach seems to be to use conventional storage for existing database relations and reduced storage for derived database relations. An implementation of *SchemaLog* for multi-database interoperability among several INGRES databases is described in [LSPS95] and a standalone implementation of *SchemaLog* is in progress.

References

[AG87] Abiteboul, S. and Grumbach, S. Col: A logic-based language for complex objects. In *Proc. of Workshop on Database Programming Languages*

[ALSS96] Andrews, Alanoly J., Lakshmanan, Laks V.S., Shiri, Nematollaah, and Subramanian, Iyer N. On implementing advanced database programming languages. Technical Report TR-DB-96-04, Concordia University, Montreal, Quebec, April 1996.

[CCS95] Codd, E.F., Codd, S.B., and Salley C.T. Providing olap (on-line analytical processing) to user-analysts, 1995. White paper - URL: http://www.arborsoft.com/papers/coddTOC.html.

[Chi89] Chimenti, D. *et al.* The ldl system prototype. *IEEE TKDE*, 2(1):76–90, 1989.

[CKW89] Chen, W., Kifer, M., and Warren, D.S. Hilog as a platform for database language. In *2nd Intl. Workshop on Database Programming Languages*, June 1989.

[CKW93] Chen, W., Kifer, M., and Warren, D.S. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3), 1993.

[Dob95] Dobbie, Gillian. Foundations of deductive object-oriented database systems. Phd dissertation, Univ. of Melbourne, Parkville, Australia, March 1995.

[GBLP96] Gray, J., Bosworth, A., Layman, A., and Pirahesh H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th ICDE*,1996.

[KLW95] Kifer, M., Lausen, G, and Wu, J. Logical foundations for object-oriented and frame-based languages. *Journal of ACM*, May 1995.

[Law93] Lawley, M. J. A prolog interpreter for F-logic. Technical report, Griffith University, 1993.

[Lef93] Lefebvre, Alexandre. Implementing an object-oriented database system using a deductive database system. Technical report, Griffith University, April 1993.

[LSPS95] Lakshmanan, L.V.S., Subramanian, I. N., Papoulis, Despina, and Shiri, Nematollaah. A declarative system for multi-database interoperability. In *Proc. of the 4th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST)*, Montreal, July 1995. Springer-Verlag. Tools Demo.

[LSS93] Lakshmanan, L.V.S., Sadri, F., and Subramanian, I. N. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proc. 3rd International Conference on Deductive and Object-Oriented Databases (DOOD '93)*. Springer-Verlag, LNCS-760, December 1993.

[LSS96] Lakshmanan, L.V.S., Sadri, F., and Subramanian, I. N. Logic and algebraic languages for interoperability in multidatabase systems. Technical report, Concordia University, Montreal, Feb 1996. Accepted to the Journal of Logic Programming.

[RSS92] Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Coral: Control, relations, and logic. In *Proc. Int. Conf. on Very Large Databases*, 1992.

[SW95] Sagonas, Konstantinos and Warren, David S. Efficient execution of hilog in wam-based prolog implementations. Technical report, Depertment of Comp. Sc., SUNY, Stony Brook, NY 11794-4400, 1995.

[Ull89] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Maryland, 1989.

[Vis93] Visual Edge Software Ltd., St-Laurent, Quebec, Canada. *The UIM/X Developer's Guide*, 1993.