SchemaSQL – A Language for Interoperability in Relational Multi-database Systems*

Laks V. S. Lakshmanan† **Fereidoon Sadri**‡ **Iyer N. Subramanian**†

Abstract

We provide a principled extension of SQL, called *SchemaSQL*, that offers the capability of uniform manipulation of data and meta-data in relational multi-database systems. We develop a precise syntax and semantics of *SchemaSQL* in a manner that extends traditional SQL syntax and semantics, and demonstrate the following. (1) *SchemaSQL* retains the flavour of SQL while supporting querying of both data and meta-data. (2) It can be used to represent data in a database in a structure substantially different from original database, in which data and meta-data may be interchanged. (3) It also permits the creation of views whose schema is dynamically dependent on the contents of the input instance. (4) While aggregation in SQL is restricted to values occurring in one column at a time, *SchemaSQL* permits "horizontal" aggregation and even aggregation over more general "blocks" of information. (5) *SchemaSQL* provides a great facility for interoperability and data/meta-data management in relational multi-database systems. We provide many examples to illustrate our claims. We outline an architecture for the implementation of *SchemaSQL* and discuss implementation algorithms based on available database technology that allows for powerful integration of SQL based relational DBMS.

* This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the National Science Foundation (NSF), and The University of North Carolina at Greensboro.

† Dept of Computer Science, Concordia University, Montreal, Canada. {laks,subbu}@cs.concordia.ca

‡ Dept of Mathematical Sciences, University of North Carolina, Greensboro, NC. sadri@uncg.edu

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

1 Introduction

In recent years, there has been a tremendous proliferation of databases in the work place, dominated by relational database systems. An emerging need for sharing data and programs across the different databases has motivated the need for *Multi-database systems* (MDBS), sometimes also referred to as heterogeneous database systems and federated database systems. Systems capable of operating over a distributed network and encompassing a heterogeneous mix of computers, operating systems, communication links, and local database systems have become highly desirable, and commercial products are slowly appearing on the market. For surverys on MDBS, see [ACM90] (in particular, Sheth and Larson, and Litwin, Mark, and Roussopoulos), and Hsiao [Hsi92].

One of the fundamental requirements in a multi-database system is *interoperability*, which is the ability to uniformly *share*, *interpret*, and *manipulate* information in component databases in a MDBS. Almost all factors of heterogeneity in a MDBS pose challenges for interoperability. These factors can be classified into *semantics* issues (*e.g.*, interpreting and cross-relating information in different local databases), *syntactic* issues (*e.g.*, heterogeneity in database schemas, data models, and in query processing, etc.), and *systems* issues (*e.g.*, operating systems, communication protocols, consistency management, security management, etc). We focus on syntactic issues here. We consider the problem of interoperability among a number of component relational databases storing semantically similar information in structurally dissimilar ways. As was pointed out in [KLK91], the requirements for interoperability even in this case fall beyond the capabilities of conventional languages like SQL.

Some of the key features required of a language for interoperability in a (relational) MDBS are the following. (1) The language must have an expressive power that is *independent* the *schema* with which a database is structured. For instance, in most conventional relational languages, *some* queries (*e.g.*, "find all department names") expressible against the database `univ-A` in Figure 2 are no longer expressible when the information is reorganized according to the schema of, say, `univ-B` there. This is undesirable and should be avoided. (2) To promote interoperability, the language must permit the *restructuring* of one database to conform to the schema of another. (3) The language must be easy to use and yet sufficiently expressive. (4) The language must provide the full data manipulation and view definition capabilities, and must be *downward compatible* with SQL, in the sense that it must be compatible with SQL syntax and semantics. We impose this requirement in view of the importance and popularity of SQL in the database world. (5) Finally, the language must admit effective and efficient implementation. In particular, it must be possible to realize a *non-intrusive* implementation that would require *minimal additions* to component RDBMS.

Contributions: (1) We propose a language called *SchemaSQL* which meets the above criteria, review the syntax and semantics of SQL, and develop *SchemaSQL* as a *principled extension* of SQL. As a result, for a SQL user, adapting to *SchemaSQL* is relatively easy. (2) We illustrate via examples the following powerful features of *SchemaSQL* : (i) *uniform* manipulation of data and meta-data; (ii) creating *restructured views* and the ability to *dynamically create output schemas*; (iii) the ability to express *sophisticated aggregate computations* far beyond those expressible in conventional languages like SQL (Sections 3.1, 4.1). (3) We propose an *implementation architecture* for *SchemaSQL* that is designed to build on *existing* RDBMS technology, and requires *minimal additions* to it, while greatly enhancing its power (Section 5). We provide an implementation algorithm for *SchemaSQL*, and establish its correctness. We also discuss novel query optimization issues that arise in the context of this implementation. (4) Finally, we propose an extension to *SchemaSQL* for systematically resolving the *semantic heterogeneity* problem arising in a MDBS environment (Section 6).

In this paper we illustrate the semantics of *SchemaSQL* mainly via examples. A precise semantics of *SchemaSQL*, together with many more examples illustrating its powerful features can be found in [LSS96b].

2 Syntax

Our goal is to develop *SchemaSQL* as a principled extension of SQL. To this end, we briefly analyze the syntax of SQL, and then develop the syntax of *SchemaSQL* as a natural extension. Our discussion below is itself a novel way of viewing the syntax and semantics of SQL, which, in our opinion, helps a better understanding of SQL subtleties.

In an SQL query, the (tuple) variables are declared in the `from` clause. A variable declaration has the form `<range> <var>`. For example, in the query in Figure 1(a), the expression `employees T` declares `T` as a variable that ranges over the (tuples of the) relation `employees` (in the usual SQL jargon, these variables are called *aliases.*) The `select` and `where` clauses refer to (the extension of) attributes, where an attribute is denoted as `<var>.<attName>`, `var` being a (tuple) variable declared in the `from` clause, and `attName` being the name of an attribute of the relation (extension) over which `var` ranges.

When no ambiguity arises, SQL permits certain abbreviations. Queries of Figure 1(b,c) are equivalent to the first one, and are the most common ways such queries are written in practice. Note that in Figures 1(b) and 1(c), `employees` acts essentially as a tuple variable.

The *SchemaSQL* syntax extends that of SQL in several directions.

1. The federation consists of databases, with each database containing relations. The syntax allows to distinguish between (the components of) different databases.
2. To permit meta-data queries and restructuring views, *SchemaSQL* permits the declaration of other types of variables in addition to the (tuple) variables permitted in SQL.
3. Aggregate operations are generalized in *SchemaSQL* to make horizontal and block aggregations possible, in addition to the usual vertical aggregation in SQL.

In this section we will concentrate on the first two aspects. Restructuring views and aggregation are discussed in Section 4.

Variable Declarations in *SchemaSQL*

SchemaSQL permits the declaration of variables that can range over any of the following five sets: (i) names of databases in a federation; (ii) names of the relations in a database; (iii) names of the attributes in the scheme of a relation; (iv) tuples in a given relation in a database; and (v) values appearing in a column corresponding to a given attribute in a relation. Variable declarations follow the same syntax as `<range> <var>` in SQL, where `var` is any identifier. However, there are two major differences. (1) The only kind of range permitted in SQL is a set of tuples in some relation in the database, whereas in *SchemaSQL* any of the five kinds of ranges above can be used to declare variables. (2) More importantly, the range specification in SQL is made using a constant, i.e. an identifier referring to a specific relation in a database. By contrast, the diversity of ranges possible in *SchemaSQL* permits range specifications to be *nested*, in the sense that it is possible to say, e.g., that `X` is a variable ranging over the relation names in a database `D`, and that `T` is a tuple in the relation denoted by `X`. These ideas are made precise in the following definition.

```
select T.name
from   employees T
where  T.department =
         "Marketing"
```
(a)

```
select employees.name
from   employees
where  employees.department =
         "Marketing"
```
(b)

```
select name
from   employees
where  department =
       "Marketing"
```
(c)

Figure 1: Syntax of simple SQL queries

Definition 2.1 (Range Specifications) *The concepts of* range specifications, constant, *and* variable *identifiers are simultaneously defined by mutual recursion as follows:*

1. Range specifications *are one of the following five types of expressions, where* db, rel, attr *are any constant or variable identifiers (defined in 2 below).*
   - *(a) The expression* -> *denotes a range corresponding to the set of database names in the federation.*
   - *(b) The expression* db-> *denotes the set of relation names in the database* db.
   - *(c) The expression* db::rel-> *denotes the set of names of attributes in the scheme of the relation* rel *in the database* db[1].
   - *(d)* db::rel *denotes the set of tuples in the relation* rel *in the database* db.
   - *(e)* db::rel.attr *denotes the set of values appearing in the column named* attr *in the relation* rel *in the database* db.
2. *A variable declaration is of the form* <range> <var> *where* <range> *is one of the range specifications above and* <var> *is an identifier*[2]. *An identifier* <var> *is said to be a* variable *if it is declared as a variable by an expression of the form* <range> <var> *in the* from *clause. Variables declared over the ranges (a) to (e) are called* db-name, rel-name, attr-name, tuple, *and* domain variables, *respectively. Any identifier not so declared is a* constant.

As an illustration of the idea of nesting variable declarations, consider the clause from db1-> X, db1::X T. This declares X as variable ranging over the set of relation names in the database db1 and T as a variable ranging over the tuples in each relation X in the database db1.

The following sections provide several examples demonstrating various capabilities of *SchemaSQL*.

3 Fixed Output Schema

In this and the next section, we illustrate via examples the many powerful features of *SchemaSQL* . In this section we discuss *SchemaSQL* queries with a fixed output schema. The topics of dynamic output schema and restructuring views are discussed in the next section.

The following federation of databases is used as our running example. Consider the federation consisting of four databases, univ-A, univ-B, univ-C, and univ-D. *Each database has (one or more) relation(s) that record(s) the salary floors for employees by their categories and their departments, as follows:*

- univ-A has a relation salInfo (category, dept, salFloor).
- univ-B has a relation salInfo (category, dept1, dept2, ...). Note that the domains of dept1, dept2, ... are the same as the domain of salFloor in univ-A::salInfo.
- univ-C has one relation for each department with the scheme dept$_i$ (category, salFloor).
- univ-D has a relation salInfo (dept, cat1, cat2, ...). Note that the domains of cat1, cat2, ... are the same as the domain of salFloor in univ-A::salInfo.

Figure 2 shows some sample data in each of these four databases.

Example 3.1 List the departments in univ-A that pay a higher salary floor to their technicians compared with the same department in univ-B.

```
select A.dept
from   univ-A::salInfo A, univ-B::salInfo B,
       univ-B::salInfo-> AttB
where      AttB       <> "category"   and
           A.dept     =  AttB         and
(Q1)       A.category =  "technician" and
           B.category =  "technician" and
           A.salFloor >  B.AttB
```

Explanation: Variables A and B are (SQL-like) tuple variables ranging over the relations univ-A::salInfo and univ-B::salInfo, respectively. The variable AttB is declared as an attribute name of the relation univ-B::salInfo. It is intended to be a dept$_i$ attribute (hence the condition AttB <> "category" in the where clause). The rest of the query is self-explanatory. ■

Example 3.2 List the departments in univ-C that pay a higher salary floor to their technicians compared with the same department in univ-D.

[1]The intuition for the notation is that we can regard the attributes of a relation as written to the right of the relation name itself!

[2]Abbreviations similar in spirit to those allowed for SQL are also allowed in *SchemaSQL* .

univ-A

salInfo

category	dept	salFloor
Prof	CS	65,000
Assoc Prof	CS	50,000
Prof	Math	60,000
Assoc Prof	Math	55,000

univ-B

salInfo

category	CS	Math
Prof	55,000	65,000
Assoc Prof	50,000	55,000

univ-D

salInfo

dept	Prof	Assoc Prof
CS	75,000	60,000
Math	60,000	45,000

univ-C

CS

category	salFloor
Prof	60,000
Assoc Prof	55,000

Math

category	salFloor
Prof	70,000
Assoc Prof	60,000

Figure 2: Representing Similar Information Using Different Schemas in Multiple Databases univ-A, univ-B, univ-C, and univ-D

```
        select RelC
        from   univ-C-> RelC, univ-C::RelC C,
               univ-D::salInfo D
(Q2)    where  RelC        = D.dept       and
               C.category = "technician" and
               C.salFloor > D.technician
```

3.1 Aggregation with Fixed Output Schema

In SQL, we are restricted to "vertical" (or column-wise) aggregation on a *predetermined* set of columns, while *SchemaSQL* allows "horizontal" (or row-wise) aggregation, and also aggregation over more general "blocks" of information. We illustrate these points with examples. The formal development of semantics can be found in [LSS96b].

Example 3.3 The query

```
        select   T.category, avg(T.D)
(Q3)    from     univ-B::salInfo-> D,
                 univ-B::salInfo T
        where    D <> "category"
        group by T.category
```

computes the average salary floor of each category of employees over *all* departments in univ-B. This captures horizontal aggregation. The condition D <> "category" enforces the variable D to range over department names. Hence a knowledge of department names (and even the number of departments) is not required to express this query. Alternatively, we could enumerate the departments, e.g., use the condition (D = "Math" or D = "CS" or ...)[3]. By contrast, the query

```
        select T.category, avg(T.salFloor)
(Q4)    from univ-C-> D, univ-C::D T
        group by T.category
```

computes a similar information from univ-C. Notice that the aggregation is computed over a multiset of values obtained from *several relations* in univ-C. In a similar way, aggregations over values collected from more than one database can also be expressed. Block aggregations of a more sophisticated form are illustrated in Example 4.3. ■

[3] An elegant solution would be to specify some kind of "type hierarchy" for the attributes which can then be used for saying "D is an attribute of the following *kind*", rather than "D is one of the following attributes". Our proposed extension to *SchemaSQL* discussed in Section 6, addresses this issue.

4 Dynamic Output Schema and Restructuring Views

The result of an SQL query (or view definition) is a single relation. Our discussion in the previous section was limited to the fragment of *SchemaSQL* queries that produce one relation, with a fixed schema, as output. In this section, we provide examples to demonstrate the following capabilities of *SchemaSQL* . (i) *declaration of dynamic output schema*, (ii) *restructuring views*, and (iii) *interaction between dynamic output schema creation and aggregation.*

We illustrate the capabilities of *SchemaSQL* for the generation of an output schema which can dynamically depend on the input instance (*i.e.* the databases in the federation). While aggregation in SQL is restricted to vertical aggregation on a predetermined set of columns, we have so far seen that *SchemaSQL* can express horizontal aggregation and aggregation over more general "blocks" (see Example 3.3). In this section, we shall see that the combination of dynamic output schema and meta-data variables allows us to express vertical aggregation on a *variable number of columns* as well. The examples in this and later sections are based on the database schema of Figure 2.

Example 4.1 Consider the relation salInfo in the database univ-B. The following *SchemaSQL* view definition restructures this information into the format of the schema univ-A::salInfo.

```
create view
BtoA::salInfo(category, dept, salFloor) as
        select      T.category, D, T.D
(Q5)    from        univ-B::salInfo-> D,
                    univ-B::salInfo T
        where       D <> 'category'
```

Explanation: Two variables are declared in the `from` clause: `T` is a tuple variable ranging over the tuples of relation `univ-B::salInfo`, and `D` is an attribute-name variable ranging over the attributes of `univ-B::salInfo`. The condition in the `where` clause forces `D` to be a department name. Finally, each output tuple `(T.category,D,T.D)` lists the category, department name, and the corresponding salary floor (which is in the format of `univ-A::salInfo`).

Note that each tuple in the `univ-B::salInfo` format generates several tuples in the `univ-A::salInfo` scheme. The mapping, in this respect, is one-to-many. But each *instantiation* of the variables in the query, actually contributes to one output tuple. ■

The following example illustrates restructuring involving dynamic creation of output schema.

Example 4.2 This view definition restructures data in `univ-A::salInfo` into the format of the schema `univ-B::salInfo`.

```
      create view AtoB::salInfo(category, D) as
       select   A.category, A.salFloor
(Q6)   from     univ-A::salInfo A, A.dept D
```

Explanation: Each tuple of `univ-A::salInfo` contains the salary floor for one category in a single department, while each tuple of `univ-B::salInfo` contains the salary floors for one category in every department. Intuitively, all tuples in `univ-A::salInfo` corresponding to the same category are grouped together and "merged" to produce one output tuple.

Another aspect of this restructuring view is the use of variables in the `create view` clause. The variable `D` in `create view AtoB::salInfo(category, D)` is declared as a domain variable ranging over the values of the `dept` attribute in the relation `univ-A::salInfo`. Hence, the schema of the view `AtoB::salInfo` is "dynamically" declared as `AtoB::salInfo(category, dept1, ..., deptn)`, where `dept1, ..., deptn` are the values occurring in the `dept` column in the relation `univ-A::salInfo`.

The restructuring in this example corresponds to a many-to-one mapping from instantiations to output tuples. ■

In the full paper [LSS96b], we present additional examples to illustrate restructuring views that distribute values from one tuple into many relations, and vice-versa. Examples of many-to-many mappings (*e.g.* mappings between schemas of `univ-B` and `univ-D`) are also given there.

4.1 Aggregation with Dynamic View Definition

In Section 3, we illustrated the capability of *SchemaSQL* for computing (i) horizontal aggregation and (ii) aggregation over blocks of information collected from several relations, or even databases. In this section, we shall see that when *SchemaSQL* aggregation is combined with its view definition facility, it is possible to express vertical aggregation over a *variable* number of columns, determined dynamically by the input instance.

Example 4.3 Suppose that in the database `univ-D` in Figure 2, there is an additional relation `faculty(dname, fname)` relating each department to its faculty. Consider the query

```
       select U.fname, avg(T.C)
       from univ-D::salInfo-> C,
(Q7)   univ-D::salInfo T, univ-D::faculty U
       where C <> "dept" and T.dept = U.dname
       group by U.fname
```

Q7 computes, for each faculty, the faculty-wide average floor salary of *all* employees (over all departments) in the faculty. Notice that the aggregation is performed over 'rectangular blocks' of information. Consider now the following view definition Q8, which is essentially defined using the query Q7.

```
   create view averages::salInfo(faculty, C) as
      select U.fname, avg(T.C)
      from univ-D::salInfo-> C,
(Q8)  univ-D::salInfo T, univ-D::faculty U
      where C <> "dept" and T.dept = U.dname
      group by U.fname
```

The view defined by Q8 actually computes, for each faculty, the average floor salary in *each category* of employees (over all departments) in the faculty. This is achieved by using the variable `C`, ranging over categories, in the dynamic output schema declaration through the `create view` statement. ■

5 Implementation Issues

In this section we describe the architecture of a system for implementing a multidatabase querying and restructuring facility based on *SchemaSQL*. A highlight of our architecture is that it builds on existing architecture in a *non-intrusive way*, requiring minimal extensions to prevailing database technology. This makes it possible to build a *SchemaSQL* system on top of (already available) SQL systems. We also identify novel query optimization opportunities that arise in a multidatabase setting.

The architecture consists of a *SchemaSQL* server that communicates with the local databases in the federation. We assume that the meta-information comprising of component database names, names of the relations in each database, names of the attributes in each relation, and possibly other useful information (such as statistical information on the component databases useful for query optimization) are stored in the *SchemaSQL* server in the form of a relation called

Federation System Table (FST). Due to the varying degrees of autonomy component databases enjoy in a multidatabase system, some or all of this information may not available. In [LSS96b] we describe a flexible architecture that makes use of as much of the available information as possible. In discussions here, we assume that the component database names as well as their schema information is available in the *SchemaSQL* server.

In our architecture, global *SchemaSQL* queries are submitted to the *SchemaSQL* server, which determines a series of local SQL queries and submits them to the local databases. The *SchemaSQL* server then collects the answers from local databases, and, using its own *resident* SQL engine, executes a final series of SQL queries to produce the answer to the global query. Intuitively, the task of the *SchemaSQL* server is to compile the *instantiations* for the variables declared in the query, and enforce the conditions, groupings, aggregations, and mergings to produce the output. Many query optimization opportunities at different stages, and at different levels of abstraction, are possible, and should be employed for efficiency (see discussions at the end of this section). Figure 3 depicts our architecture for implementing *SchemaSQL*. Algorithm 5.1, gives a more detailed account of our query processing strategy.

Query processing in a *SchemaSQL* environment consists of two major phases. In the first phase, tables called VIT's (Variable Instantiation Table) corresponding to the variable declaration in the from clause of a *SchemaSQL* statement are generated. The schema of a VIT consists of all the variables in one or more variable declarations in the from clause and its contents correspond to instantiations of these variables. *VIT's are materialized by executing appropriate SQL queries on the FST and/or component databases.* In the second phase, the *SchemaSQL* query is rewritten into an equivalent SQL query on the VIT's and the generated answer is appropriately presented to the user. Our algorithm below considers *SchemaSQL* queries with a fixed output schema possibly with aggregation. A complete algorithm for the implementation of the full language, as well as novel query optimization strategies are discussed in [LSS96b].

In the following, we assume that the FST has the scheme `FST (db-name, rel-name, attr-name)`. Also, we refer to the db-name, rel-name, and attr-name variables (defined in Definition 2.1) collectively as *meta-variables.*

Algorithm 5.1 *SchemaSQL Query Processing*
INPUT: A *SchemaSQL* query with a fixed output schema and aggregation.
OUTPUT: Bindings for the variables appearing in the select clause of the *SchemaSQL* statement.
METHOD: The algorithm consists of two phases.
(1) Corresponding to a set of variable declarations in the from clause, create VITs using one or more SQL queries against some local databases and/or the FST.
(2) Rewrite the original *SchemaSQL* query against the federation into an equivalent query against the set of VIT relations and run it using the resident SQL server.

Phase I

(0) The input *SchemaSQL* statement is rewritten into the following form such that the conditions in the where clause are in conjunctive normal form.

select $S_1, \ldots, S_n$
from $\langle range_1 \rangle\ V_1,\ \ldots,\ \langle range_k \rangle\ V_k$
where $\langle cond_1 \rangle$ and ... and $\langle cond_m \rangle$
groupby groupList
having haveConditions

(1) Consider the variable declaration for variable V_i.
(a) If V_i is a meta-variable: In this case, all variables in the declaration $\langle range_i \rangle\ V_i$ range over meta-data. Create VIT_i with a schema consisting of V_i and any variables appearing in $\langle range_i \rangle$, and contents obtained using an appropriate SQL query against the FST. For example, let '`D::rel->` V_i' be the declaration and one of the conditions in the where clause be 'V_i *.op.* c' where *.op.* is a (in)equality operator and c is a constant. Obtain VIT_i corresponding to VIT_i as:

select db-name as D, attr-name as V_i
from FST
where rel-name = 'rel' and attr-name *.op.* c

Meta-variable declarations of other forms are handled in a similar way.
(b) If V_i is a domain variable: Group together domain variable declarations that are declared using the same tuple variable as for V_i. Create VIT_i with schema consisting of the domain variables in the group. Obtain the (tuple of) bindings for the attr-name variables (in the range declarations) in the group, using their corresponding VIT's. Using this, generate a set of SQL queries against local databases. The contents of VIT_i will be the union of answers to these queries. For example, let db::rel T be a tuple variable declaration, 'T.A V_i' be the declaration for the domain variable and 'V_i *.op.* c' be a condition in the where clause, where *.op.* is a (in)equality operator and c is a constant. Let 'T.attr V_j' be another domain variable declaration in the from clause.

(i) Obtain the bindings for attr-name variable A from its VIT, and name it relation T.

(ii) For $a \in T$, generate an SQL query against database db:
select a as V_i, $attr$ as V_j
from rel
where a *.op.* c

(iii) Obtain VIT_i as the union of answers to all the SQL queries generated in (ii), against db.

Domain-variable declarations of other forms (*e.g.* when db, rel, attr are also variables) are handled in a similar way.

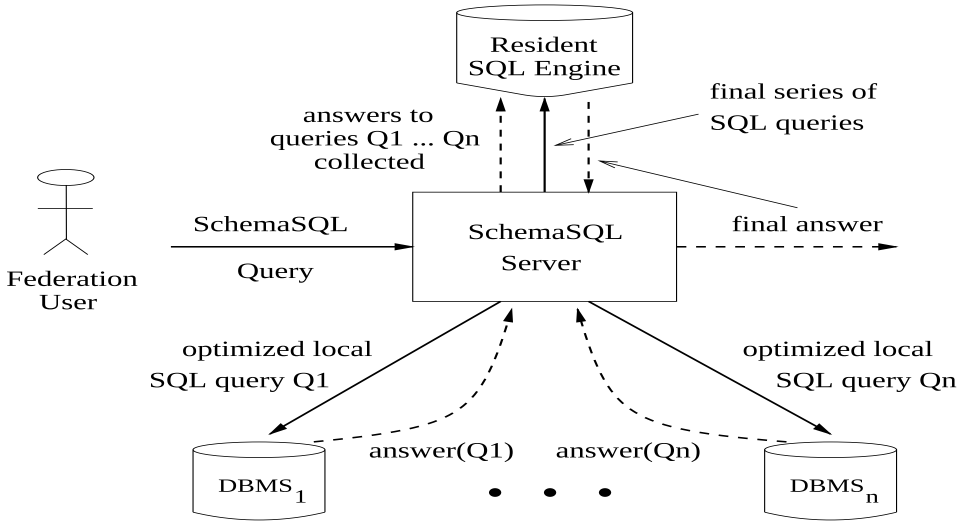

Figure 3: SchemaSQL – Implementation Architecture

(c) If V_i is a tuple variable: Generate bindings for the meta-variables in $\langle range_i \rangle$ as in case (a). The attributes of the VIT corresponding to V_i are obtained by analyzing the *select*, *where*, *group by*, and *having* clauses. We consider a variable V as *relevant* in the context of tuple variable V_i, if (i) V is of the form $V_i.C$ or $V_i.c$ (C, c are a variable and constant respectively) and occurs in the *select*, *groupby*, or *having* clause, or (ii) V occurs in the declaration of V_i and either is compared with a variable in the *where* clause, or occurs in the *select* clause, or (iii) V occurs in a relevant variable of the form $V_i.V$ and V is compared with a variable in the *where* clause. The schema of the VIT is the set consisting of all relevant variables in the context of V_i. Finally, the contents of VIT are obtained by generating appropriate SQL queries against local databases. In general, if there are occurrences of the form $V_i.C$ in the *select* or the *where* clause, the VIT would be obtained as a union of several SQL queries.

For example, let the select clause contain an aggregation of the form avg($V_i.C$), the variable declaration be 'db::R V_i' and two of the conditions in the where clause be '$V_i.a_1$ *.op.* $V_j.a_2$' and '$V_i.a_3$ *.op.* c', where a_1, a_2, a_3, c are constants.

(i) Obtain a VIT corresponding to `db-> R` (as in (a) above) and name it T.

(ii) The schema of VIT_i is $\{V_i.C, V_i.a_1\}$.

(ii) For each $r \in T$, obtain the attribute names in relation r (using an SQL query on the FST) and generate the following SQL statement.
Let $c_1, \ldots, c_k$ be the instantiations of C, corresponding to r.

select c_1 *as* $V_i.C$, a_1 *as* $V_i.a_1$
from r
where $V_i.a_3$ *.op.* c
UNION
$\cdots$
UNION
select c_k *as* $V_i.C$, a_1 *as* $V_i.a_1$
from r
where $V_i.a_3$ *.op.* c

(iii) Obtain VIT_i as the union of all the SQL statements generated in (ii).

Tuple variable declarations of other forms are handled in a similar way.

Phase II

Execution of this phase happens in the *SchemaSQL* server. The *SchemaSQL* query is rewritten into an equivalent conventional SQL statement on the VIT's generated in Phase I, in the following way. *(a)* The select, group by, and having clauses of the rewritten query are obtained by copying the corresponding clauses in the *SchemaSQL* query after disambiguating the attribute names that appear in more than one VIT; *(b)* the from clause consists of the subset of VIT's relevant to the final result, and *(c)* the where clause is obtained by retaining the conditions involving tuple variables and by adding a condition '$VIT_i.X = VIT_j.X$' for tables VIT_i and VIT_j having a common attribute.

It is interesting to note that using our algorithm, the novel horizontal aggregation (Section 3.1, Example 3.3) which cannot be performed in a conventional SQL system, can be easily realized in our framework. More general kind of 'block' aggregations can also be handled in a similar way – details can be found in [LSS96b], which also contains the proof of the following theorem.

Theorem 5.1 *Algorithm 5.1 correctly computes answers to* SchemaSQL *queries.*

Example 5.1 In this example, we illustrate our algorithm using a variant of the query Q2 of Example 3.2.
'List the departments in `univ-C` that pay a higher salary floor to their technicians compared with the same department in `univ-D`. List also the (higher) pay.'

VIT_1
RelC
cs
math

VIT_2	
RelC	C.salFloor
cs	50,000
math	40,000

VIT_3	
D.dept	D.technician
cs	55,000
math	40,000

Figure 4: Example – Query Processing

```
select   RelC, C.salFloor
from     univ-C-> RelC, univ-C::RelC C,
         univ-D::salInfo D
where    RelC =  D.dept        and
         C.category  = "technician"  and
         C.salFloor  >  D.technician
```

Phase I

VIT_1 corresponding to the variable declaration `univ-C-> RelC` is created using:

```
select rel-name as RelC
from   FST
where  db-name = 'univ-C'
```

Figure 4 shows VIT_1. To generate the SQL statement that creates VIT_2, the following SQL queries are issued against the FST.

```
select attr-name           select attr-name
from FST                   from FST
where                      where
db-name = 'univ-C'         db-name = 'univ-C'
and rel-name = 'cs'        and rel-name = 'math'
```

Let the answer to both the queries be {*category, salFloor*}. VIT_2, corresponding to `univ-C::RelC C` is obtained by querying the database univ-C using:

```
select 'cs' as RelC, cs.salFloor as C.salFloor
from    cs
where   cs.category = 'technician'
        UNION
select 'math' as RelC,
        math.salFloor as C.salFloor
from    math
where   math.category = 'technician'
```

To obtain VIT_3 corresponding to `univ-D::salInfo D`, querying is first done on the FST to obtain the names of the attributes in relation *salInfo* of database univ-D:

```
select attr-name
from   FST
where db-name = 'univ-D' & rel-name = 'salInfo'
```

Let the answer to this query be {*dept, prof, technician*}. VIT_3, shown in Figure 4 is obtained by querying the database univ-D:

```
select   dept as D.dept,
         technician as D.technician
from     salInfo
```

Phase II

Having obtained all the VIT's corresponding to the variable declarations, Phase II now consists of rewriting the *SchemaSQL* statement into the following SQL statement to obtain the final answer.

```
select RelC, C.salFloor
from   VIT_2, VIT_3
where RelC = D.dept and
     C.salFloor > D.technician
```

■

A *SchemaSQL* system on the PC-Windows platform is currently under implementation.

Query Optimization

There are several opportunities for query optimization which are peculiar to the MDBS environment. In the following, we identify the major optimization possibilities and sketch how they can be incorporated in Algorithm 5.1.

1. The conditions in the where clause of the input *SchemaSQL* query should be pushed inside the local spawned SQL queries so that they are as 'tight' as possible. Algorithm 5.1 incorporates this optimization to some extent.
2. Knowledge of the variables in the select and where clauses can be made use of to minimize the size of the VIT's generated in Phase I. For example, if certain attributes are not required for processing in Phase II, they can 'dropped' while generating the local SQL queries.
3. If more than one tuple variable refers to the same database, and their relevant where conditions do not involve data from another database, the SQL statements corresponding to these variable declarations should be combined into one. This would have the effect of combining the VIT's corresponding to these variable declarations and thus reducing the number of spawned local SQL queries. This can be incorporated by modifying the step I(c) of our algorithm.
4. One of the costliest factors for query evaluation in a multidatabase environment is database connectivity. We should minimize the number of times connections are made to a database during query evaluation. Thus, the spawned SQL statements need to be submitted (in batches) to the component databases in such a way that they are evaluated in minimal number of connections to the databases.
5. In view of the *sideways information passing (sip)* [BR86] technique inherent in our algorithm, reordering of variable declarations would result in more efficient query processing. However, the heuristics that meta-variables obtain a significantly less number of bindings when compared to other variables in a multidatabase setting,

presents novel issues in reordering. For instance the order `'db::r.a R, -> D, D-> R'` suggested by the conventional reordering strategies could be worse than `'-> D, D-> R, db::r.a R'` because of the lower number of bindings R obtains for VIT_2 in the latter.

6. We should make use of works such as [LN90, LNS90] to determine which of the VIT's should be generated first so that the tightest bindings are passed for generating subsequent VITs.

7. If parallelism can be supported, SQL queries to multiple databases can be submitted in parallel.

Replication and Inconsistency

Replication of data, and inconsistency among data from local databases are common in multidatabase systems. The view facility of *SchemaSQL* and our architecture provide the means to cope with these difficulties.

Controlled (intentional) replication can be addressed through the Federation System Table, FST. A copy of the replicated data is identified as the *primary copy*, and the FST routes all references to the replicated data to the primary copy. The choice of the primary copy is influenced by factors such as efficiency of query processing, network connectivity, and the load at local sites. In a dynamic scheme, the FST is updated in response to changes in the network (e.g., network disconnection) and the load at local sites.

Data replication and overlap among (independent) local sites, with the possibility of inconsistency, is much subtler. The view facility of *SchemaSQL* can be used to resolve inconsistencies by exposing only the appropriate data through the view. This is similar to the approach taken in multidatabase systems utilizing an (integrated) global schema. Our architecture is more flexible, and does not require a global schema, yet, the view facility can mimic the role played by the global schema for resolving data inconsistency.

6 Semantic Heterogeneity

One of the roadblocks to achieving true interoperability is the heterogeneity that arises due to the difference in the meaning and interpretation of similar data across the component systems. This *semantic heterogeneity* problem has been discussed in detail in [Sig91], [KCGS93], [HM93]. A promising approach to dealing with semantic heterogeneity is the proposal of Sciore, Siegel, and Rosenthal [SSR94]. The main idea behind their proposal is the notion of *semantic values*, obtained by introducing an explicit context information to each data object in the database. In applying this idea to the relational model, they develop an extension of SQL called Context-SQL (C-SQL) that allows for explicitly accessing the data as well as its context information.

In this section, we sketch how *SchemaSQL* can be extended with the wherewithal to tackle the semantic heterogeneity problem. We extend the proposal of [SSR94], by associating the context information to relation names as well as attribute names, in addition to the values in a database. Also, in the *SchemaSQL* setting, there is a natural need for including the type information of an object as part of its context information. We propose techniques for intensionally specifying the semantic values as well as for algorithmically deriving the (intensional) semantic value specification of a restructured database, given the old specification and the *SchemaSQL* view definition. The following example illustrates our ideas. Details can be found in [LSS96b].

Example 6.1 Consider the database univInfoA having a single relation stats with scheme {cat, cs, math, ontario, quebec}. This database stores information on the floor salary of various employee categories for each department (as in univ-B of the university federation) as well as information on the average number of years it takes to get promoted to a category, in each province in the country. The type information of the objects in the database univInfoA is stored in a relation called isa and is captured using the following rules[4]:

$isa(cs, dept) \leftarrow$
$isa(math, dept) \leftarrow$
$isa(ontario, prov) \leftarrow$
$isa(quebec, prov) \leftarrow$
$isa(C, cat) \leftarrow stats[cat \rightarrow C]$
$isa(S, sal) \leftarrow stats[D \rightarrow S], isa(D, dept)$
$isa(Y, year) \leftarrow stats[P \rightarrow Y], isa(P, prov)$

Now, consider restructuring univInfoA into univInfoB which consists of two relations salstats{dept, prof, assoc-prof} and timestats{prov, prof, assoc-prof}. salstats has tuples of the form $< d, s_1, s_2 >$, representing the fact that d is a department that has a floor salary of s_1 for category professor, and s_2 for associate professor. A tuple of the form $< p, y_1, y_2 >$ in timestats says that p is a province in which the average time it takes to reach the category professor is y_1 and to reach the category associate professor is y_2. The following *SchemaSQL* statements perform the restructuring that yields univInfoB.

```
create view
       univInfoB::salstats(dept, T.cat) as
select D, T.D
from   univInfoA::stats T,
       univInfoA::stats-> D,
where  D isa 'dept'
create view
```

[4] The syntax of the type specification rules is based on the syntax of SchemaLog [LSS96a].

```
       univInfoB::timestats(prov, T.cat) as
select P, T.P
from   univInfoA::stats T,
       univInfoA::stats-> P,
where  P isa 'prov'
```

Note how the type information is used in the *where* clause to elegantly specify the range of the attribute variables. Our algorithm that processes the restructuring view definitions derives the following intensional type specification for univInfoB:
$isa(prof, cat) \leftarrow$
$isa(assoc\text{-}prof, cat) \leftarrow$
$isa(D, dept) \leftarrow salstats[dept \rightarrow D]$
$isa(S, sal) \leftarrow salstats[C \rightarrow S], isa(C, cat)$
$isa(P, prov) \leftarrow timestats[prov \rightarrow P]$
$isa(Y, year) \leftarrow timestats[P \rightarrow Y], isa(P, prov)$

Query processing in this setting involves the following modification to the processing of comparisons mentioned in the user's query. The comparison is performed after (a) finding the type information using the specification, (b) finding the associated context information, and (c) applying the appropriate conversion functions. [LSS96b] has the details.

7 Comparison with Related Work

In this section, we compare and contrast our proposal against some of the related work for meta-data manipulation and multidatabase interoperability.

The features of *SchemaSQL* that distinguishes it from similar works include

- Uniform treatment of data and metadata.
- No explicit use of object identifiers.
- Downward compatibility with SQL.
- Comprehensive aggregation facility.
- Restructuring views, in which data and meta-data may be interchanged.
- Designed specifically for interoperability in multi-database systems.

Further, we also discuss the implementation of *SchemaSQL* on a platform of SQL servers.

In [Lit89, GLRS93], Litwin *et al.* propose a multi-database manipulation language called MSQL that is capable of expressing queries over multiple databases in a single statement. MSQL extends the traditional functions of SQL to the context of a federation of databases. The salient features of this language include the ability to retrieve and update relations in different databases, define multi-database views, and specify compatible and equivalent domains across different databases. [MR95] extends MSQL with features for accessing external functions (for resolving semantic heterogeneity) and for specifying a global schema against which the component databases could be mapped. Though MSQL (and its extension) has facilities for ranging variables over multiple database names, its treatment of data and meta-data is non-uniform in that relation names and attribute names are not given the same status as the data values. The issues of schema independent querying and resolving schematic discrepancies of the kind discussed in this paper, are not addressed in their work.

Many object-oriented query languages, by virtue of treating the schema information as objects, are capable of powerful meta-data querying and manipulation. Some of these languages include XSQL (Kifer, Kim, and Sagiv [KKS92]), HOSQL (Ahmed *et al.* [ASD+91]), and OSQL (Chomicki and Litwin [CL93]).

XSQL ([KKS92]) has its logical foundations in F-logic ([KLW95]) and is capable of querying and restructuring object-oriented databases. However, it is not suitable for the needs addressed in this paper as its syntax was not designed with interoperability as a main goal. Besides, the complex nature of this query language raises concerns about effective and efficient implementability, a concern not addressed in [KKS92]. The Pegasus Multi-database system ([ASD+91]) uses a language called HOSQL as its data manipulation language. HOSQL is a functional object-oriented language that incorporates non-procedural statements to manipulate multiple databases. OSQL ([CL93]), an extension of HOSQL is capable of tackling schematic discrepancies among heterogeneous object-oriented databases with a common data model. Both HOSQL and OSQL do not provide for ad-hoc queries that refer to many local databases in the federation in one shot. While XSQL, HOSQL, and OSQL have a SQL flavor, unlike *SchemaSQL* , they do not appear to be downward compatible with SQL syntax and semantics. In other related work, [Ros92] proposes an interesting algebra and calculus that treats relation names at par with the values in a relation. However, its expressive power is limited in that attribute names, database names, and comprehensive aggregation capabilities are not supported.

In [LBT92], Lefebvre, Bernus, and Topor use F-logic ([KLW95]), to reconcile schematic discrepancies in a federation of relational databases. Unlike *SchemaSQL* which can provide a 'dynamic global schema', ad hoc queries that refer the data and schema components of the local databases in a single statement cannot be posed in their framework.

UniSQL/M [KGK+95] is a multidatabase system for managing a heterogeneous collection of relational database systems. The language of UniSQL/M, known as SQL/M, provides facilities for defining a global schema over related entities in different local databases, and to deal with semantic heterogeneity issues such as scaling and unit transformation. However, it does not have facilities for manipulating metadata. Hence features such as restructuring views that transform data into metadata and vice versa, dynamic schema definitions, and extended aggregation facilities supported in *SchemaSQL* are not available in SQL/M. The emerging standard for SQL3 ([SQL96, Bee93]) supports ADTs and oid's, and thus shares some features with higher-order languages. However, even though it is computationally complete, to our knowledge it does not *directly* support the kind of higher-order features in *SchemaSQL*.

Krishnamurthy and Naqvi [KN88] and Krishnamurthy, Litwin, and Kent [KLK91] are early and influential proposals that demonstrated the power of using variables that uniformly range over data and metadata, for schema browsing and interoperability. While such 'higher-order variables' admitted in *SchemaSQL* have been inspired by these proposals, there are major differences that distinguish our work from the above proposals. (i) These languages have a syntax closer to that of logic programming languages, and far from that of SQL. (ii) More importantly, these languages do not admit tuple variables of the kind permitted in *SchemaSQL* (and even SQL). This limits their expressive power. (iii) Lastly, aggregate computations of the kind discussed in Sections 3.1 and 4.1 are unique to our framework, and to our knowledge, not addressed elsewhere in the literature.

In the context of multi-dimensional databases (MDDB) and on-line analytical processing (OLAP), there is a great need for powerful languages expressing complex forms of aggregation ([CCS95]). The powerful features of *SchemaSQL* for horizontal and block aggregation will be especially useful in this context (*e.g.* see Examples 3.3, 4.3). We have recently observed that the *Data Cube* operator proposed by Gray et al. ([GBLP96]) can be simulated in *SchemaSQL*. Unlike the cube operator, *SchemaSQL* can express any subset of the data cube to any level of granularity.

In other related work, Gyssens et al. ([GLS96]) develop a general data model called *Tabular Data Model*, which subsumes relations and spreadsheets as special cases. They develop an algebra for querying and restructuring tabular information and show that the algebra is complete for a broad class of natural transformations. They also demonstrate that the tabular algebra can serve as a foundation for OLAP. Restructuring views expressible in *SchemaSQL* can also be expressed in their algebra but they do not address aggregate computations.

In [LSS96a, LSS93], we proposed a logic-based query/restructuring language SchemaLog, for facilitating interoperability in multidatabase systems. SchemaLog admits a simple syntax and semantics, but allows for expressing powerful queries and programs in the context of schema browsing and interoperability. A formal account of SchemaLog's syntax and semantics can be found in [LSS96a]. SchemaLog can also express the complex forms of aggregation discussed in this paper.

SchemaSQL has been to a large extent inspired by SchemaLog. Indeed, the logical underpinnings of *SchemaSQL* can be found in SchemaLog [LSS96b]. However, *SchemaSQL* is *not* obtained by simply "SQL-izing" SchemaLog. There are important differences between the two languages. (i) *SchemaSQL* has been designed to be as close as possible to SQL. In this vein, we have developed the syntax and semantics of *SchemaSQL* by extending that of SQL. SchemaLog on the other hand has a syntax based on logic programming. (ii) Answers to *SchemaSQL* queries come with an associated schema. In SchemaLog, as in other logic programming systems, answers to queries are simply a set of (tuples of) bindings of variables in the query (unless explicitly specified using a restructuring rule). (iii) The aggregation semantics of *SchemaSQL* is based on a 'merging' operator ([LSS96b]). There is no obvious way to simulate merging in SchemaLog. (iv) To facilitate an ordinary SQL user to adapt to *SchemaSQL* in an easy way, we have designed *SchemaSQL* without the following features present in SchemaLog – *(a)* function symbols and *(b)* explicit access to tuple-id's. As demonstrated in this paper, the resulting language is simple, yet powerful for the interoperability needs in a federation.

Acknowledgements: We would like to thank Frédéric Gingras for his valuable comments at various stages of evolution of this paper and the anonymous referees for their suggestions that helped improve the paper.

References

[ACM90] ACM. *ACM Computing Surveys*, volume 22, Sept 1990. Special issue on HDBS.

[ASD+91] Ahmed, R., Smedt, P., Du, W., Kent, W., Ketabchi, A., and Litwin, W. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, December 1991.

[Bee93] Beech, D. Collections of objects in SQL3. In *Proc. 19th VLDB Conference*, 1993.

[BR86] Bancilhon, F. and Ramakrishnan, R. An amateur's introduction to recursive query-processing strategies. In *Proc. ACM SIGMOD*, 1986.

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

[CL93] Chomicki, J. and Litwin, W. Declarative definition of object-oriented multidatabase mappings. In Ozsu, M.T, Dayal, U, and Valduriez, P, editors, *Distributed Object Management*. M. Kaufmann Publishers, Los Altos, California, 1993.

[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 International Conference on Data Engineering*, pages 152–159, 1996.

[GLRS93] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. Query languages for relational multidatabases. *VLDB Journal*, 2(2):153–171, 1993.

[GLS96] Gyssens, Marc, Lakshmanan, L.V.S., and Subramanian, I. N. Tables as a paradigm for querying and restructuring. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, June 1996.

[HM93] Hammer, J. and McLeod, D. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Intl Journal of Intelligent & Cooperative Information Systems*, 2(1), 1993.

[Hsi92] Hsiao, D.K. Federated databases and systems: Part-one – a tutorial on their data sharing. *VLDB Journal*, 1:127–179, 1992.

[KCGS93] Kim, W., Choi, I., Gala, S.K., and Scheevel, M. On resolving schematic heterogeneity in multidatabase systems. *Distributed and Parallel Databases*, 1(3), 1993.

[KGK+95] Kelley, W., Gala, S. K., Kim, W., Reyes, T.C., and Graham, B. Schema architecture of the UniSQL/M multidatabase system. In *Modern Database Systems*. 1995.

[KKS92] Kifer, Michael, Kim, Won, and Sagiv, Yehoshua. Querying object-oriented databases. In *Proc. ACM SIGMOD*, pages 393–402, 1992.

[KLK91] Krishnamurthy, R., Litwin, W., and Kent, W. Language features for interoperability of databases with schematic discrepancies. In *Proc. ACM SIGMOD*, 1991.

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

[KN88] Krishnamurthy, R. and Naqvi, S. Towards a real horn clause language. In *Proc. 14th VLDB Conf.*, pages 252–263, 1988.

[LBT92] Lefebvre, A., Bernus, P., and Topor, R. Query transformation for accessing heterogeneous databases. In *Workshop on Deductive Databases in conjunction with JICSLP*, pages 31–40, November 1992.

[Lit89] Litwin, W. MSQL: A multidatabase language. *Information Science*, 48(2), 1989.

[LN90] Lipton, Richard and Naughton, Jeffrey. Query size estimation by adaptive sampling. In *Proc. ACM PODS*, 1990.

[LNS90] Lipton, Richard, Naughton, Jeffrey, and Schneider, Donovan. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD*, 1990.

[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.

[LSS96a] 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.*

[LSS96b] Lakshmanan, L.V.S., Sadri, F., and Subramanian, I. N. SchemaSQL – a language for querying and restructuring multidatabase systems. Technical report, Concordia University, Montreal, 1996. *In Preparation.*

[MR95] Missier, P. and Rusinkiewicz, Marek. Extending a multidatabase manipulation language to resolve schema and data conflicts. In *Proc. Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)*, Atlanta, May 1995.

[Ros92] Ross, Kenneth. Relations with relation names as arguments: Algebra and calculus. In *Proc. 11th ACM Symp. on PODS*, pages 346–353, June 1992.

[Sig91] Semantic Issues in Multidatabase Systems. *Sigmod Record*, 20(4), December, 1991. Special Issue Edited by Amit Sheth.

[SQL96] SQL Standards Home Page. SQL 3 articles and publications, 1996. URL: www.jcc.com/sql_articles.html.

[SSR94] Sciore, E., Siegel, M., and Rosenthal, A. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, 19(2):254–290, June 1994.