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.

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 surveys 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 [KLK01], 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

---

* 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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996
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 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.
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.
   
   (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. 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 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 depti (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,
where A.category = "technician" and
A.dept = B.dept
and A.salFloor > B.salFloor

Explanation: Variables A and B are (SQL-like) tuple variables ranging over the relations univ-A::salInfo and univ-B::salInfo, respectively. The variable B.dept is declared as an attribute name of the relation univ-B::salInfo. It is intended to be a dept1 attribute (hence the condition B.dept = "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.
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

\[
\text{select } \text{category, avg(T.D) from univ-B::salInfo\rightarrow D, }
\]

\[
\text{where D <> "category"}
\]

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

\[
\text{select T.category, avg(T.salFloor) from univ-C::D T group by T.category}
\]

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.

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 metadata 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.

\[
\text{create view BtoA::salInfo(category, dept, salFloor) as select T.category, D.T.D from univ-B::salInfo\rightarrow D, univ-B::salInfo T where D <> "category"}
\]

³An 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.

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

<table>
<thead>
<tr>
<th>univ-A</th>
<th>univ-B</th>
<th>univ-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>salInfo</td>
<td>salInfo</td>
<td>salInfo</td>
</tr>
<tr>
<td>category</td>
<td>category</td>
<td>category</td>
</tr>
<tr>
<td>dept</td>
<td>CS</td>
<td>Math</td>
</tr>
<tr>
<td>Prof</td>
<td>65,000</td>
<td>68,000</td>
</tr>
<tr>
<td>Assoc Prof</td>
<td>50,000</td>
<td>55,000</td>
</tr>
<tr>
<td>Assoc Prof</td>
<td>Math</td>
<td>60,000</td>
</tr>
<tr>
<td>Prof</td>
<td>75,000</td>
<td>60,000</td>
</tr>
<tr>
<td>Assoc Prof</td>
<td>50,000</td>
<td>45,000</td>
</tr>
</tbody>
</table>

```
select RelC from univ-C::RelC, univ-D::RelC where RelC.dept = D.dept and C.category = "technician" and C.salFloor > D.technician
```
Explanation: Two variable 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
from univ-A::salInfo A, A.dept D
(Q6)

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

\[
\text{select U.fname, avg(T.C)}
\text{from univ-D::salInfo -> C,}
\]

\[(Q7) \text{univ-D::salInfo T, univ-D::faculty U}
\text{where C <> "dept" and T.dept = U.dname}
\text{group by U.fname}
\]

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.

\[
\text{create view averages::salInfo(faculty, C) as}
\text{select U.fname, avg(T.C)}
\text{from univ-D::salInfo -> C,}
\]

\[(Q8) \text{univ-D::salInfo T, univ-D::faculty U}
\text{where C <> "dept" and T.dept = U.dname}
\text{group by U.fname}
\]

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 be 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 VIT's 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.

\[
\text{select } S_1, \ldots, S_n \text{ from } \langle \text{range}_1 \rangle V_1, \ldots, \langle \text{range}_k \rangle V_k \\
\text{where } \langle \text{cond}_1 \rangle \text{ and } \ldots \text{ and } \langle \text{cond}_m \rangle \\
\text{groupby groupList} \text{ having } \text{haveConditions}
\]

(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 \(\langle \text{range}_i \rangle \) \(V_i\) range over meta-data. Create VIT \(V_i\) with schema consisting of \(V_i\) and any variables appearing in \(\langle \text{range}_i \rangle\), and contents obtained using an appropriate SQL query against the FST. For example, let \(\text{db::rel} \rightarrow V_i\) be the declaration and one of the conditions in the where clause be \(V_i\text{ .op. c}\) where \(\text{op}\) is a (in)equality operator and \(c\) is a constant. Obtain VIT \(V_i\) corresponding to \(V_i\) as:

\[
\text{select db-name as D, attr-name as attr from FST} \\
\text{where rel-name = 'rel' and attr-name .op. c}
\]

(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 \(V_i\) 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 \(V_i\) 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 \(\text{op}\) 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:

\[
\text{select a as V_t, attr as V_f from rel} \\
\text{where a .op. c}
\]

(iii) Obtain VIT \(V_i\) 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.
If $V_i$ is a tuple variable: Generate bindings for the meta-variables in (range) 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 $\text{avg}(V_i.C)$, the variable declaration be $\text{db->R}$, and name it $T$. The schema of $VIT_i$ is \{$V_i.C, V_i.a_1$\}.

(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.'
**Figure 4: Example – Query Processing**

<table>
<thead>
<tr>
<th>$VIT_1$</th>
<th>$VIT_2$</th>
<th>$VIT_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelC</td>
<td>RelC</td>
<td>D.dept</td>
</tr>
<tr>
<td>cs</td>
<td>cs</td>
<td>55,000</td>
</tr>
<tr>
<td>math</td>
<td>C.salFloor</td>
<td>40,000</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td></td>
</tr>
</tbody>
</table>

**Phase I**

$VIT_1$, corresponding to the variable declaration $\text{univ-C} \rightarrow \text{RelC}$ is created using:

```sql
select \text{rel-name} as \text{RelC}
from \text{FST}
where \text{db-name} = \text{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:

```sql
select \text{attr-name}
from \text{FST}
where \text{db-name} = \text{univ-C}'
and \text{rel-name} = \text{cs}'
```

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

```sql
select 'cs' as \text{RelC}, \text{cs}.salFloor as \text{C.salFloor}
from \text{cs}
where \text{cs}.category = \text{technician}'
UNION
select 'math' as \text{RelC}, \text{math}.salFloor as \text{C.salFloor}
from \text{math}
where \text{math}.category = \text{technician}'
```

To obtain $VIT_3$ corresponding to $\text{univ-D}::\text{salInfo} \ D$, querying is first done on the FST to obtain the names of the attributes in relation $\text{salInfo}$ of database $\text{univ-D}$:

```sql
select \text{attr-name}
from \text{FST}
where \text{db-name} = \text{univ-D}' & \text{rel-name} = \text{salInfo}'
```

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

```sql
select \text{dept} as \text{D.dept}, \text{technician} as \text{D.technician}
from \text{salInfo}
```

**Phase II**

Having obtained all the $VIT_1's$ corresponding to the variable declarations, Phase II now consists of rewriting the SchemaSQL statement into the following SQL statement to obtain the final answer:

```sql
select \text{RelC}, \text{C.salFloor}
from $VIT_2, VIT_3$
where \text{RelC} = \text{D.dept} and \text{C.salFloor} > \text{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 $VIT_i's$ 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 $VIT_i's$ 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 VIT2 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], [KC93], [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 defi-

cition. The following example illustrates our ideas. Details can be found in [LSS96b].

Example 6.1 Consider the database univlnfoA 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 univlnfoA is stored in a relation called isa and is captured using the following rules:

\[
\text{isa(cs, dept)} \leftarrow \\
\text{isa(math, dept)} \leftarrow \\
\text{isa(ontario, prov)} \leftarrow \\
\text{isa(quebec, prov)} \leftarrow \\
\text{isa(C, cat)} \leftarrow \text{stats[cat -> C]} \\
\text{isa(S, sal)} \leftarrow \text{stats[D -> S], isa(D, dept)} \\
\text{isa(Y, year)} \leftarrow \text{stats[P -> Y], isa(P, prov)}
\]

Now, consider restructuring univlnfoA into univlnfoB which consists of two relations salstats{dept, prof, assoc-prof} and timesteps{prov, prof, assoc-prof}. salstats has tuples of the form \(\langle d, s_1, s_2 \rangle\), 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 \(\langle p, y_1, y_2 \rangle\) in timesteps 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 univlnfoB.

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

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

247
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:

\[
\begin{align*}
&\text{isa(prof, cat)} \leftarrow \\
&\text{isa(assoc-prof, cat)} \leftarrow \\
&\text{isa(D, dept)} \leftarrow \text{salstats}[\text{dept} \rightarrow D] \\
&\text{isa(S, sal)} \leftarrow \text{salstats}[C \rightarrow S], \text{isa}(C, \text{cat}) \\
&\text{isa(P, prov)} \leftarrow \text{timestats}[\text{prov} \rightarrow P] \\
&\text{isa(Y, year)} \leftarrow \text{timestats}[P \rightarrow Y], \text{isa}(P, prov)
\end{align*}
\]

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 specialization, (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 multidatabase manipulation language called MSQIL that is capable of expressing queries over multiple databases in a single statement. MSQIL 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 MSQIL 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 MSQIL (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 schemaic 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 [KK92]), HOSQL (Ahmed et al. [ASD+91]), and OSQL (Chomicki and Litwin [CL93]). XSQL ([KK92]) 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 [KK92]. 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 schemaic 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 schemaic 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 UnixSQL/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 is-
sues 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 \textit{SchemaSQL} are not available in SQL/M. The emerging standard for SQL3 ([SQL96, Dee93]) 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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{SchemaSQL}. Unlike the cube operator, \textit{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 \textit{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 \textit{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 \textit{SchemaLog}, for facilitating interoperability in multidatabase systems. \textit{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 \textit{SchemaLog}'s syntax and semantics can be found in [LSS96a]. \textit{SchemaLog} can also express the complex forms of aggregation discussed in this paper.

\textit{SchemaSQL} has been to a large extent inspired by \textit{SchemaLog}. Indeed, the logical underpinnings of \textit{SchemaSQL} can be found in \textit{SchemaLog} [LSS96b]. However, \textit{SchemaSQL} is not obtained by simply "SQL-izing" \textit{SchemaLog}. There are important differences between the two languages. (i) \textit{SchemaSQL} has been designed to be as close as possible to SQL. In this vein, we have developed the syntax and semantics of \textit{SchemaSQL} by extending that of SQL. \textit{SchemaLog} on the other hand has a syntax based on logic programming. (ii) Answers to \textit{SchemaSQL} queries come with an associated schema. In \textit{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 \textit{SchemaSQL} is based on a 'merging' operator ([LSS96b]). There is no obvious way to simulate merging in \textit{SchemaLog}. (iv) To facilitate an ordinary SQL user to adapt to \textit{SchemaSQL} in an easy way, we have designed \textit{SchemaSQL} without the following features present in \textit{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 Fréderic 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


Kelley, W., Gala, S. K., Kim, W., Reves, T.C., and Graham, B. Schema architecture of the UniSQL/M multibase system. in Modern Database Systems. 1995.


