Implementation of SchemaSQL – A Language for Relational Multi-Database Systems

Fereidoon Sadri*
Department of Mathematical Sciences
University of North Carolina at Greensboro
Greensboro, NC 27412

Salley Bradford Wilson†
IBM Global Services
Mt. Hope Church Road
McCleansville, NC 27420

February 16, 1997

Abstract

We discuss the implementation of *SchemaSQL* , an extension of SQL for relational multi-database systems. A mediator-based architecture is used to realize the *SchemaSQL* system using (standard) SQL components.

1 Introduction

SchemaSQL is an extension of SQL specifically designed for relational multi-database systems. It was introduced recently by Lakshmanan *et al* [3]. *SchemaSQL* is particularly powerful to handle *schematic heterogeneity*, *i.e.* when similar data is represented by different schemata (structures) in different databases. This paper reports an implementation of *SchemaSQL* . The implementation is based on the architecture presented in [3], and uses (standard) SQL components for the realization of the *SchemaSQL* system. This architecture is based on the *mediator* concept, which was introduced by Wiederhold [7]. The *SchemaSQL* system communicates with information sources in SQL. As such, any source that can exhibit an SQL interface can be a component in a *SchemaSQL* system. These sources include relational SQL databases, but also other (legacy) databases that have an SQL interface, as well as any other sources that are *wrapped* to exhibit an SQL interface.
Our implementation uses PowerBuilder[1], a development environment used to build client/server applications that access databases. We discuss the implementation of *SchemaSQL* , highlighting optimization issues in the processing of queries.

*Supported by grants from NSF and University of North Carolina, Greensboro.

†Work was done when the author was a student at the Department of Mathematical Sciences, University of North Carolina, Greensboro.

[1]PowerBuilder is a registered trademark of PowerSoft.

This paper is organized as follows. We begin by a short description of *SchemaSQL* syntax and semantics in Section 2, and present the implementation of non-aggregate queries in Section 3. In Section 4 we discuss the implementation of aggregate queries in *SchemaSQL* . Conclusions and future directions are presented in Section 5.

2 SchemaSQL

SchemaSQL was introduced by Lakshmanan *et al* in [3]. Its main features can be summarized as follows:

- *SchemaSQL* is a true extension of SQL. It is downward compatible with SQL
- *SchemaSQL* has been designed specifically for multi-database systems consisting of relational databases.
- Data (extension) and *meta data* (intension) are treated uniformly in *SchemaSQL* . This makes, among other things, *schema browsing* and *restructuring view definition and creation* possible.
- *SchemaSQL* is particularly powerful in dealing with structural heterogeneity (where similar data is represented by different schemata in different component databases.)
- It is possible to declare views with *dynamic* schema in *SchemaSQL* . That is, the schema of the view is a function of the content of the (input) databases, and can be determined only at execution time (rather than a static view schema which is determined at compilation time.) The view itself can be a collection of relations, or even a collection of databases (as opposed to regular SQL in which a view is a single relation.)
- *SchemaSQL* provides powerful aggregation facilities. In addition to the regular "vertical" aggregation available in SQL, we can express aggregation over rows (horizontal) or over "blocks" of data. These capabilities are similar in nature to those available in multidimensional databases and OLAP [1], and in the more recent *data cube* [2] facility proposed for relational database systems.
- Finally, the issue of semantic heterogeneity, which is critical in a multi-database system, is orthogonal to *SchemaSQL* concept. Proposed approaches to the resolution and handling of semantic heterogeneity can be combined with *SchemaSQL* to provide a very powerful muli-database management system. We are currently investigating the integration of the *semantic contexts* concept [6] with *SchemaSQL* .

Example 2.1 Consider databases holding census information over many years. We will concentrate on `census90` and `census94` databases, containing data from 1990 and 1994 census. The `census90` database stores population information in the relation `population` with the following scheme

`census90::population`{`category, state, number`}

The same type of information is stored in a different format in a relation with the same name, `population`, in the database `census94`:

`census94::population`{`category, Alabama, ..., Missouri, ..., Wyoming`}

In other words, `census90::population` lists the states *vertically*, while `census94::population` lists them *horizontally*. State names are attribute values in the first relation, and form a portion of the relation *instance* (extension), while, in the second relation, they are *attribute names* forming a portion of the relation *scheme* (intension). Examples of the two relations are shown in Figures 1 and 2.

category	state	number
...	...	...
programmer1	Alabama	5000
...	...	...
programmer1	Missouri	6000
...	...	...
programmer1	Wyoming	1000
...	...	...

Figure 1. `census90::population`

category	Alabama	...	Missouri	...	Wyoming
...	...	...	...	...	...
programmer1	5400	...	6500	...	1200
...	...	...	...	...	...

Figure 2. `census94::population`

Now, assume we are interested to find those states whose population has grown, over the four year period 1990 to 1994, by 10% or more for a specific category such as "programmer1" Even if the two relations existed in a single database, expressing this query in SQL would be quite difficult. We could write SQL queries, one for each state, checking for the condition of the query individually. The query can be written easily in *SchemaSQL* :

```
select  P1.state
from    census90::population P1,
        census94::population->S,
        census94::population P2
where   P1.category = "programmer1"  and
        P2.category = "programmer1"  and
        P1.state = S                 and
        P2.S >= 1.10*P1.number
```

The declaration `census94::population->S` in the `from` clause declares `S` as an *attribute name* variable ranging over the names of the attributes on the relation `census94::population`. The variables `P1` and `P2` are the usual tuple variables (called *aliases* in SQL terminology). Here we do not need these variables: The following is an equivalent query. But the first version is, in general, more readable.

```
select  census90::population.state
from    census94::population->S
where   census90::population.category = "programmer1"  and
        census94::population.category = "programmer1"  and
        census90::population.state = S                 and
        census94::population.S >= 1.10*census90::population.number
```

2.1 Syntax of *SchemaSQL*

The syntax of *SchemaSQL* is similar to that of SQL. Syntactical constructs are added in *SchemaSQL* to

- Permit the specification of databases (in addition to relations and attributes.)
- Permit the declaration of variables that range over database names, relation names, and attribute names (in addition to the usual SQL tuple variables, which range over the tuples in a relation).

Similar to SQL, variables are declared in the `from` clause in *SchemaSQL* [3]. Here is a listing of possible declarations:

`db::rel T`	Declares `T` as a tuple variable (alias) of relation `rel` in database `db` `T` ranges over the tuples of relation `db::rel`. The components `db` and/or `rel` can be variables themselves.
`db::rel.att V`	Declares `V` as a domain variable of attribute `att` of relation `db::rel`. `V` ranges over the values of the attribute `att` in the relation `db::rel`. The components `db`, `rel` and/or `att` can be variables themselves.
`T.att V`	Also declares `V` as a domain variable when `T` is a tuple variable. `V` ranges over the values of the attribute `att` of the tuple `T`.
`db::rel-> A`	Declares `A` as an attribute-name variable of relation `db::rel`. `A` ranges over the names of the attributes of the relation `db::rel`. The components `db` and/or `rel` can be variables themselves.
`db-> R`	Declares `R` as a relation-name variable of database `db`. `R` ranges over the names of the relations of the database `db`. The component `db` can be a variable.
`-> D`	Declares `D` as a database-name variable. `D` ranges over the names of the databases in the multi-database system.

The choice of arrow "`->`" in the above declarations was motivated by the fact that attributes (attribute-names) of a relation are usually listed horizontally above the relation (instance). Hence `db::rel-> A` was chosen to declare `A` as an attribute-name variable of the relation `db::rel`. Similarly, `db-> R` and `-> D` are relation-name and database-name variables, respectively.

2.2 Semantics of *SchemaSQL*

Semantics of *SchemaSQL* is discussed in detail in [4]. Here we provide a brief summary. Let's first start with the semantics of SQL, which is usually defined as follows: Let $R_1, \ldots, R_n$ be the relations (or aliases) listed in the `from` clause of the SQL query, *e.g.* as in

```
select  <target>
from    R1, ..., Rn
where   <condition>
```

Then the answer to the query can be written (in relational algebra) as follows:

$$\Pi_{<target>}\sigma_{<condition>}(r_1 \times \cdots \times r_n)$$

where $r_1, \ldots, r_n$ are the instances of the relations (or aliases) $R_1, \ldots, R_n$. Note that $R_1, \ldots, R_n$ can be considered as *tuple variables* that range over the tuples of the relations (instances) $r_1, \ldots, r_n$.

Alternatively, the semantics of SQL can be expressed in terms of the *instantiations* of the tuple variables declared in the from clause. Intuitively, an instantiation of a tuple variable R_i associates a tuple from the relation instance r_i as a value for R_i. Now the result of an SQL query can be defined in terms of the set of instantiations of tuple variables that satisfy the `where`-clause conditions, restricted to the attributes listed in the `select`-clause target.

We can extend this alternative description of SQL semantics to *SchemaSQL* . The result of a *SchemaSQL* query is defined exactly as above except that *SchemaSQL* variables include database-name, relation-name, attribute-name, and attrivute-value variables in addition to tuple variables. Hence the instantiation of a *SchemaSQL* variable associates a value from its corresponding domain. For example, an instantiation of a database-name variable, declared as `-> D`, associates the name of a database in the multi-database system with the variable `D`.

The above semantics description is only for simple *SchemaSQL* queries. Semantics of *SchemaSQL* queries with dynamic output schema, aggregate operations (*i.e.* `count, sum, avg, min, max`) with (possibly) `group by` and `having` clauses, and combination of dynamic output schema with aggregation are much more interesting and involved. Rather than attempting a description here, we will present some examples in later sections for these types of queries to provide an intuitive understanding of *SchemaSQL* advanced features. Semantics of *SchemaSQL* is discussed in detail in [4].

3 Implementation

Lakshmanan *et al* discuss the architecture of a *SchemaSQL* system in [3]. One of the important features of this architecture is to build the *SchemaSQL* system using (regular) SQL components. The architecture incorporates a regular SQL database, called the *resident* SQL engine, which is used to generate the final answer to an *SchemaSQL* query. It is assumed that meta-data (*i.e.* database, relation, attribute names and their types) is available at the resident database. This information is stored in a relation called FST (*Federation System Table*) with the scheme `(db-name, rel-name, attr-name, attr-type)`. This is just a simplifying assumption. In the absence of FST, the needed meta-data information can be obtained from local databases. The system accepts a (global) *SchemaSQL* query over the multi-database system, and generates a series of (regular) SQL queries that are dispatched to local databases. The answers to these local queries are collected and stored in temporary relations in the resident database. The *SchemaSQL* system then generates a final SQL query that is executed at the resident database, and accesses data from the temporary tables.

Intuitively, each temporary relation corresponds to the set of the instantiations of a variable declared in the *SchemaSQL* query `from` clause. These temporary relations are called VITs (*Variable Instantiation Tables*). Many optimization techniques can be employed such as "pushing" selection conditions in the `where` clause of the query, to the extent possible, *down* to local databases, and restricting the retrieved tuples to the attributes mentioned in the *select* clause and those needed to enforce the (remaining) `where`-clause conditions. Advanced *SchemaSQL* queries, such as queries involving dynamic output schema declaration and/or aggregation, provide very interesting optimization opportunities and challenges. Some of these optimization techniques are discussed in [3].

This architecture can be regarded as a *mediator / wrapper* architecture [5, 7]. The *SchemaSQL* system is designed as a mediator to access SQL systems. In this respect, *SchemaSQL* system can access any data source that is "wrapped" to expose an SQL interface to outside world. This is particularly significant for *legacy* database systems, such as (DBTG) network databases and hierarchical (such as IMS) databases, most of which already possess such a wrapper, or are capable of being wrapped to expose an SQL interface.

Our implementation uses PowerBuilder[2], a graphical development environment used to build client/server applications that access databases. In the following subsections we present the implementation issues and the algorithms used. The algorithms are from [3], adapted here for the PowerBuilder environment.

3.1 Initial Issues

When initially addressing the issues involved in implementing the *SchemaSQL* language, the idea of using PowerBuilder for the implementation was appealing because PowerBuilder has its own database engine and database interface tools. It was thought that these features would simplify the problems surrounding simultaneous connectivity to multiple databases. By using the PowerBuilder transaction object for connectivity and the PowerBuilder database engine to create the resident database, start-up issues were somewhat minimized, and we were able to address the larger problem of coding the *SchemaSQL* system.

The most commonly used database interface in PowerBuilder is the data window object. It was quickly ascertained that while this tool is quite useful for reporting purposes, it has limited scope when it comes to the dynamic queries that are required to process *SchemaSQL* queries. Because of these limitations, any SQL sub-queries needed to create the final *SchemaSQL* query are, of necessity, embedded in the script. The two PowerBuilder data types of DynamicStagingArea and DynamicDescriptionArea allow creation of dynamic embedded SQL queries and Data Definition Language statements.

There are four formats that can be used for dynamic queries in PowerBuilder. These formats use combinations of the DynamicStagingArea and DynamicDescriptionArea and their predefined functions. The first format is used to execute a statement that has no input parameters and no result set. This format is used for the dropping of temporary relations on the *SchemaSQL* resident engine after the query has been completed. The second format is used to execute a statement that has input parameters and no result set. This format is used to create (*e.g.* declare) temporary relations on the resident database when dynamic parameters are needed, for example when database-name

[2]PowerBuilder is a registered trademark of PowerSoft.

and/or relation-name variables are present. The third format is used for statements that produce a result set in which the columns are known. This format was used in creating attribute-value temporary relations because the attribute column name was already known. The final format is used when a result set is produced, but the number of columns and/or the number of input parameters is not known at compile time. This format was used in the creation of tuple tables and in the execution of the final *SchemaSQL* query, as the number of columns to be retrieved in the result set is not known until compile time.

The first task was to determine that all of the temporary variable instantiation tables (VIT) could be created on the resident database engine. This goal was achieved using a combination of the four PowerBuilder dynamic SQL formats explained above and also several data window objects. Data windows could be used whenever certain queries were predefined (for example, queries against the FST table on the *SchemaSQL* database). The data window provides a simpler and more concise method of database access, so it was used whenever possible. The dynamic creation of data windows was attempted for some of the dynamic queries, but, in the end, the unknowns were too great for the data window tool as it has been defined by PowerBuilder.

3.2 Creating the Variable Instantiation Tables

The first step in the processing of a *SchemaSQL* query is the parsing of the query. The declared variables and their types (database-name, relation-name, attribute-name, tuple, and attribute-value) are determined by examining the `from` clause. Recall that the declaration of a variable may involve other variables. For example, in the declaration `db::rel->A`, which declares `A` as an attribute-name variable, the database and/or relations fields `db` and `rel` may themselves be variables. We say a variable `V` is *dependent* upon a variable `U` if `U` occurs in the declaration of `V`. We require this *dependence* relationship to be acyclic. In fact, we impose a stronger requirement: Variables can only appear in positions that are consistent with their type. The following example clarifies the point.

Example 3.1 Consider the query "Find the names of the databases in the multi-database system that have the same name as a relation in one of the databases of the multi-database." This query can be specified in *SchemaSQL* as follows

```
select D
from   -> D, -> D', D'-> R
where  D = R
```

But the following query is illegal in *SchemaSQL* since the variable `D` is declared as a database-name as well as a relation-name attribute.

```
select D
from   -> D, -> D', D'-> D
```

It should be noted that we have placed the above (stronger) restriction on *SchemaSQL* mainly for the purpose of clarity of queries. The practice of using the same variable symbol to bind two or more values is legal (and, in fact, preferred) in logical database languages such as Datalog. But we contend that allowing it in *SchemaSQL* could make queries more difficult to understand.

The `select`, `where`, and `group by` clauses (if any) are also parsed and stored in internal data structures to be used by the query decomposition and processing algorithms.

After all preliminary parsing is complete, the next step is to create the temporary variable instantiation tables (VIT) on the resident database for each variable declared in the *SchemaSQL* query `from` clause. Our algorithm orders the variables so that when a variable `V` is being processed, all variables upon which `V` is dependent have already been processed. The VIT for all variables, except for a tuple variable, has one column to hold the values for the variable itself, plus one column for each variable upon which this variable depends. The value of a tuple variable is a tuple, and the corresponding VIT should have columns to accommodate the tuple.

As we mentioned before, `where`-clause optimization is used to push the selections to the local databases. Further optimization is possible for tuple variables by accessing only the required attributes, namely, the attributes appearing in the `select` clause, and those appearing in the `where`-clause conditions that were not pushed to the local database.

For *meta* variables, that is, database-name, relation-name, and attribute-name variables, the requisite information for the temporary table is already in the Federation System Table (FST) in the resident database. The algorithm for creating a VIT for a meta-variable is as follows:

Algorithm 3.1 Algorithm to generate the variable instantiation table (VIT) for a meta-variable `V` declared in the `from`-clause of a *SchemaSQL* query.

/* Generate the scheme of the VIT for `V` */

Check to find out if `V` is dependent upon any other variable `U`. Add one column to the scheme for each variable `U` upon which `V` depends. Add one column for `V` itself.

Create (declare) the VIT in the resident database. Also generate the `select`-clause of the SQL query `Q` that will be used to fill out this VIT.

/* Generate an SQL query `Q` to the resident database FST to fill out the VIT */

Check the `where`-clause conditions of the *SchemaSQL* query. Add to the `where`-clause of `Q` those conditions using only `V`, constants, and other variables upon which `V` is dependent.

Submit `Q` to the resident database to retrieve the data from the federation system table (FST) and load into the VIT for `V`.

The algorithms used to generate the variable instantiation table for tuple and attribute-value variables are more complicated then the above algorithm. The main differences are:

- The required data is no longer available at the resident database. SQL queries should be generated and dispatched to local databases to fill out the VIT. The fourth PowerBuilder dynamic SQL format has been utilized for this purpose.

- It is possible that the required data should be fetched from several relations and/or databases. This happens when the attribute-value or the tuple variable is dependent upon relation and/or database variables.

    1. If only the relation-name is a variable, then the data needed to fill out the VIT of the tuple or attribute-value variable can be obtained from a single database. The SQL query to this database can be formed as the union of queries, one for each value of the relation-name variables. The values of the relation-name variable are obtained from its VIT at the resident database.
    2. If the database-name is a variable, or both database-name and relation-name are variables, then the data needed to fill out the VIT of the tuple or attribute-value variable should be obtained from several databases, one per value of the database-name variable. The values of the database-name variable are also obtained from its VIT at the resident database. SQL queries for the databases are formed similar to the previous case.

       We should note here that `where`-clause optimization, *i.e.* pushing the *SchemaSQL* `where`-clause conditions into the SQL queries for the VITs of meta attributes, also achieves a significant reduction of data transmission for query processing. This reduction is due to having fewer values in the VITs of meta attributes, and hence fewer databases and relations will be accessed and less data will be transmitted when populating the VITs of tuple and attribute-value variables in the above cases.

- Attribute names and type information for each attribute on the given relation (or relations, if variables are involved) are needed. This information can be obtained directly from the FST, so a hidden data window is used in this situation.

- For tuple variables, further optimization is possible by restricting the retrieval from local databases to the required attributes.

Example 3.2 Consider the query in Example 2.1. Here we discuss the creation of the variable instantiation tables for this query. The creation of the final SQl query is discussed in the next section.

The *SchemaSQL* query has three variables: `P1` and `P2` are tuple variables, and `S` is an attribute-name variable. The SQL query that will be dispatched to the `census90` database is as follows:

```
select  state, number
from    population
where   category = "programmer1"
```

Note that appropriate `where` clause condition has been pushed to the local database, and only relevant attributes are retrieved. The result of this query is stored as the variable instantiation table VIT$_{\mathtt{p1}}$ in the resident database. Note that the declaration of VITp1 in the resident database needs the types of the attributes `state` and `number`. This information is available in the federation system table (FST) in the resident database.
The SQL query to be dispatched to the `census94` database, and the creation of VITp2 is similar to the above.
The variable instantiation table for the attribute-name variable `S` is obtained from the resident database FST (Federation System Table). The SQL query is as follows:

```
select  attr-name
from    FST
where   db-name = "census94" and
        rel-name = "population"
```

3.3 The Final Query

After all of the temporary tables are created, the (final) SQL query to the resident database must be constructed. The `select` clause of the final query is obtained from the `select` clause of the original *SchemaSQL* query as follows:

- If a variable `V` appears in the *SchemaSQL* query `select` clause, then add `VITv.V` to the `select` clause of the final query, where `VITv` is the variable instantiation table of `V`.
- If `V.att` appears in the *SchemaSQL* query `select` clause, where `att` is a constant, then add `VITv.att` to the `select` clause of the final query, where `VITv` is the variable instantiation table of `V`.
- The case where `V.A` appears in the *SchemaSQL* query `select` clause, where `A` is an attribute-name variable, is more complicated. The final query is the union of a series of SQL queries, one per value of the variable `A`. The `select` clause for each of these SQL queries is obtained by adding `VITv.`$\mathtt{a}_i$, where `VITv` is the variable instantiation table of `V`, and $\mathtt{a}_i$ is a value for `A`, obtained from its own VIT. The same value $\mathtt{a}_i$ should be substituted for other occurrences of `A`, if any, in the `where` clause of the final query. Also, the VIT of the variable `A` is no longer needed in this case, and will not be included in the `from` clause of the final SQL query.

The `from` clause of the final query has the format

```
from VIT1, VIT2, ..., VITn
```

where `VIT1, VIT2, ..., VITn` are the variable instantiation tables of the variables in the *SchemaSQL* query, except for the VITs of attribute-name variables as discussed above.

The creation of the `where` clause of the final query is more involved. First, we should include a *join* condition for every pair of VITs that have a common attribute. That is, if VITi and VITj have a common attribute `V`, then add

```
VITi.V = VITj.V
```

to the `where` clause of the final query.
In addition, all remaining conditions in the *SchemaSQL* query (*i.e.* conditions that were not pushed to local SQL queries) should be transformed and included in the `where` clause of the final query. The transformation of each condition is obtained by making the same substitutions listed for the `select` clause. In particular, if an attribute-name variable `A` appears in a condition of the *SchemaSQL* query, then a series of SQL queries are generated, one per value of `A`. Each of these SQL queries is obtained by substituting a value $\mathtt{a}_i$ for *all* occurrences of `A` in the `select` and `where` clauses. The values of `A` are available from the VIT of `A` in the resident database. The final query is the union of these SQL queries.

Lastly, the final query is sent to the resident database, and the results are shown to the user in multi-line edit format.

Example 3.3 We will continue Example 3.2, in which the SQL queries to component databases and the creation of the variable instantiation tables for the execution of the query of Example 2.1 was discussed.
This example demonstrates the case where an attribute-name variable is used in combination with a tuple variable, such as `P2.S` in Example 2.1. The final query is a union of SQL queries and will be generated by the *SchemaSQL* system as follows:

```
select  VITp1.state
from    VITp1, VITp2
where   VITp1.state = "Alabama" and
        VITp2.Alabama >= 1.10*VITp1.number

union
...
union

select  VITp1.state
from    VITp1, VITp2
```

```
where   VITp1.state = "Missouri" and
        VITp2.Missouri >= 1.10*VITp1.number

union
...
union

select  VITp1.state
from    VITp1, VITp2
where   VITp1.state = "Wyoming" and
        VITp2.Wyoming >= 1.10*VITp1.number
```

4 Aggregate Queries

SchemaSQL provides powerful aggragation facilities. We can specify queries that apply aggregation functions `sum, count, avg, min, max` over rows or portion of rows of data (*horizontal aggregation*), and over rectangular blocks of data (*block aggregation*), in addition to the usual SQL aggregation over columns (*vertical aggregation*). The following is an example of a horizontal aggregation in *SchemaSQL* .

Example 4.1 The following query computes the average population in each category over all states:

```
select    T.category, avg(T.S)
from      census94::population-> S,
          census94::population T
where     S <> "category"
group by  T. category
```

The attribute-name variable `S` ranges over all attributes of the relation `census94::population` except for "`category`". Hence it ranges over all the states. Then, `T.S` ranges over all values of listed under the states in `census94::population`, which are the population, for the category specified in the tuple, for each state. The `group by` clause specifies to apply the aggregation function to each category of people.

SchemaSQL also provides view specifications with a dynamic schema, that is, the schema of the view is dependent on the data and is not fixed in advance. The dynamic schema is determined at execution time, while static (or fixed) schema is known at compile time. This is achieved by a `create view` clause with *SchemaSQL* variables used to specify the schema of the view. This dynamic output schema facility provides the means for *schema restructuring views*, which make it possible to restructure data into a different format in which data and meta-data may be interchanged. In addition, the dynamic output schema also provides more aggregation facilities for *SchemaSQL* (see Lakshmanan *et al*[3].)

Below we will discuss the implementation of *SchemaSQL* aggregate queries with fixed output schema. The implementation of dynamic output schema, and other aggregation facilities made possible by dynamic output schema, are beyond the scope of this paper.

4.1 Implementation of Aggregate Queries

The `select` clause of a *SchemaSQL* aggregate query contains (at least) one of the following aggregations (`agg` is one of the aggregate functions `sum, count, avg, min, max`.)

- `agg(V)`, where `V` is a *SchemaSQL* variable.
- `agg(T.att)`, where `T` is a tuple variable and `att` is a constant attribute name.
- `agg(T.A)`, where `T` is a tuple variable and `A` is an attribute-name variable. This case was demonstrated in Example 4.1.

In the first two cases, the variable instantiation table (VIT) of the variable `V` or `T` readily has the values subject to the aggregation in appropriate (column) setting for the final SQL query. The aggregation `agg(V)` (similarly `agg(T.att)`) is transformed to `agg(VITv.V)` (similarly `agg(VITt.att)`) and added to the `select` clause of the final SQL query, where `VITv` and `VITt` are the VITs of the variables `V` and `T`, respectively.

In the third case, when the aggregation has the form `agg(T.A)`, another temporary table must be created. This is necessary because the data must be reformatted from horizontal aggregation to SQL-friendly vertical aggregation. The new table, which we will call $\text{VIT}_{\texttt{agg}}$, has one column corresponding to the `group by` field. We name this column `groupByColumn`. The $\text{VIT}_{\texttt{agg}}$ table has additional columns corresponding to the values to be aggregated. We name a column corresponding to the aggregation `agg(T.A)` in the `select` clause of the original *SchemaSQL* query `aggragateColumnTA`. Values of the attribute-name variable `A` are retrieved from the appropriate previously created temporary table, $\text{VIT}_{\texttt{A}}$. The rows for the new table are generated from a union of selects on the group by field and each domain value being the values of $\text{VIT}_{\texttt{T}}.\texttt{a}_i$, where $\texttt{a}_i$ is a value of `A`. In this case, only $\text{VIT}_{\texttt{agg}}$ will be used in the final SQL query.

Example 4.2 The horizontal aggregation query of Example 4.1 is executed as follows:
Similar to Example 3.2 the variable instantiation table for the variable `S` is obtained from the federation system table, FST, at the resident database, and the VIT for the tuple variable `T` is obtained by dispatching an SQL query to the `census94` database. In this case, an additional temporary table, which we will call $\text{VIT}_{\texttt{agg}}$, is needed. For our example, $\text{VIT}_{\texttt{agg}}$ has two columns and is generated by the following SQL query on the resident database:

```
select  VITt.category, VITt.Alabama
from    VITt

union
...
union

select  VITt.category, VITt.Missouri
from    VITt

union
...
union

select  VITt.category, VITt.Wyoming
from    VITt
```

Note that the above query *transposes* the rows of the $\mathrm{VIT}_{\mathrm{T}}$ table into columns in $\mathrm{VIT}_{\mathtt{agg}}$. The final SQL query is an aggregation query executed on $\mathrm{VIT}_{\mathtt{agg}}$ relation, as follows. We have named the two columns of the $\mathrm{VIT}_{\mathtt{agg}}$ relation `groupByColumn` and `aggregateColumnTS`, respectively.

```
select    VITagg.groupByColumn, avg(VITagg.aggregateColumnTS)
from      VITagg
group by  VITagg.groupByColumn
```

5 Conclusions

We discussed the implementation of *SchemaSQL* , an extension of SQL for relational multi-database systems. The *SchemaSQL* system is implemented as a *mediator*, capable of accessing sources that provide an SQL interface. As such, any source that can exhibit an SQL interface can be a component in a *SchemaSQL* system. These sources include relational SQL databases, but also other (legacy) databases that have an SQL interface, as well as any other sources that are *wrapped* to exhibit an SQL interface.

We are currently studying the issue of semantic heterogeneity. The *semantic value* approach of [6] is a promising technique that can be incorporated in *SchemaSQL* to provide a powerful multi-database language and system capable of handling schematic (structural) as well as semantic heterogeneity.

References

[1] Arbor Software, "Multidimensional Analysis: Converting Corporate Data into Strategic Information." At `http://www.arborsoft.com/papers/multiTOC.html`.

[2] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing Data Cubes Efficiently." Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp 205-216.

[3] V. S. Lakshmanan, F. Sadri, and I. N. Subramanian, "SchemaSQL: A Language for Interoperability in Relational Multi-Database Systems." Proceedings of the 1996 International Conference on Very Large Databases, VLDB'96, September 1996, pp 239-250.

[4] V. S. Lakshmanan, F. Sadri, and I. N. Subramanian, "SchemaSQL: A Language for Querying and Restructuring Multi-Database Systems." Technical Report, 1997. *In preparation.*

[5] Y. Papakonstantinou, H. Garcia-Molina, and J. D. Ullman, "MedMaker: A Mediation System Based on Declarative Specifications." Proceedings of the 1996 IEEE International Conference on Data Engineering, pp 132-141.

[6] E. Sciore, M. Siegel, and A. Rosenthal, "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems." ACM Transactions on Database Systems, Vol. 19, No. 2, 1994, pp 254-290.

[7] G. Wiederhold, "Mediators in the Architecture of Future Information Systems." *IEEE Computer*, Vol. 25, No. 3, March 1992, pp 38-49.