SchemaSQL – An Extension to SQL for Multi-database Interoperability*

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

Abstract

We provide a principled extension of SQL, called *SchemaSQL*, that offers the capability of uniform manipulation of data and schema 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 flavor of SQL while supporting querying of both data and schema. (2) It can be used to transform data in a database in a structure substantially different from original database, in which data and schema 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 useful facility for interoperability and data/schema manipulation in relational multi-database systems. We provide many examples to illustrate our claims. We clealy spell out the formal semantics of *SchemaSQL* that accounts for all these features. We describe an architecture for the implementation of *SchemaSQL* and develop implementation algorithms based on available database technology that allows for powerful integration of SQL based relational DBMS. We also discuss the applicability of *SchemaSQL* for handling semantic heterogeneity arising in a multi-database system.

1 Introduction

Significant strides in relational database technology have resulted in the autonomous creation and proliferation of database systems and applications, tailored to specific needs and user communities. A single organization often contains a large number of databases designed, created, and maintained by numerous groups independently of each other. These databases can be on different operating

* 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, and IIT – Bombay. laks@cs.concordia.ca

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

§ IBM Almaden Research Center, San Jose, CA. subbu@us.ibm.com

systems, different hardware, different platforms, and use different data models. The need to share information, within a single organization as well as across different organizations, has long been identified.

There are several applications that create a need for sharing data and programs across the different databases. An example is a large organization whose operations may be divided on functional or departmental lines. Therefore, it is natural to have independent database systems that cater to different functional units and handle different organizational tasks. Normally, such an organization maintains separate databases for purchase, payroll, patents, and human resources. Suppose the organization decides to reward employees who have filed and successfully obtained $\geq n$ patents, for some suitable number n, by giving them a raise, and by providing them with better benefits. Given the size of the organization, it would like to have this process automated. If all relevant information were available in one database, this would amount to implementing some kind of trigger, a functionality readily supported in most commercial DBMSs. On the other hand, the reality is that information pertinent to applications is scattered across many database systems, on a multitude of platforms, with different DBMSs and even data models. Given the popularity of relational databases, it is reasonable to expect that in a significant number of cases, the database systems holding information of interest are relational. A challenge to developing such applications is one of being able to express and efficiently compute "cross-queries" that relate information in different databases.

The issue of *interoperability* among databases has received considerable attention as a result of such a need. Interoperability is the ability to uniformly *share*, *interpret*, *query*, and *manipulate* information across many component databases. A *multi-database system* (MDBS) provides such interoperability across a distributed network encompassing a heterogeneous mix of computers, operating systems, communication links, and local database systems. The terms *heterogeneous database systems* and *federated database systems* have also been used for these systems, sometimes with slight differences in the intended meaning. We will use the term multi-database system in this paper. Some commercial systems (*e.g.* IBM DataJoiner [DJ], UniSQL/M [KGK$^+$95]) have already appeared on the market and significant research has been conducted in this area but there are many issues that still remain unresolved. For surveys on MDBS, see [ACM90] (in particular, Sheth and Larson [SL90], and Litwin, Mark, and Roussopoulos [LMR90]), Hsiao [Hsi92], and [ERS98].

Almost all factors of heterogeneity in a MDBS pose challenges for interoperability. These factors can be classified into *semantic* 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 and query language issues in this paper. *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], even in this case, the requirements imposed by interoperability are beyond the functionalities provided by conventional query 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* of the *schema* used by the database. For instance, in most conventional relational languages, *some* queries (*e.g.*, "find all department names"), expressible against the database `univ-A` in Figure 2, Section 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 (e.g., the database `univ-A` in Figure 2) to conform to the schema of another (say, that of `univ-B` in that figure).
3. The language must be easy to use and yet sufficiently expressive.
4. The language must provide 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. This is a must given 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.

In this paper, we propose an extension to SQL which meets all the criteria above. Specifically, our contributions are the following.

- We propose a language called *SchemaSQL* which meets the above criteria. We review the syntax and semantics of SQL, and develop *SchemaSQL* as a *principled extension* of SQL (Section 2). As a result, for a SQL user, adapting to *SchemaSQL* is relatively easy.
- We study the semantics of *SchemaSQL* in depth (Sections 3 and 4), and illustrate 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* (e.g., `univ-B` in Figure 2); and (iii) the ability to express *sophisticated aggregate computations* far beyond those expressible in conventional languages like SQL.
- 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 (Theorem 5.1 and 5.2).
- We sketch novel query optimization opportunities that arise in the context of implementing *SchemaSQL*. We develop a lightweight extension to *SchemaSQL* for addressing *semantic heterogeneity* in MDBS in an elegant way (Section 6).

- We provide a comparison with related work. Section 7 discusses related work and places our work in context. Since its proposal, it has been observed by researchers that *SchemaSQL* can provide useful enhancement to the functionality of SQL even in the context of *single database systems* for such applications as database publishing on the web [Mil98], query optimization in a data warehouse [SV98], and scalable classification algorithms in data mining [WIV98]. We highlight these in Section 7.

- Finally, Section 8 summarizes the paper and discusses future work.

2 Syntax

Our goal is to develop *SchemaSQL* as a principled extension of SQL. To this end, in this section, we briefly analyze the syntax of SQL, and then develop the syntax of *SchemaSQL* as a natural extension. Later, in Section 3, we review the semantics of SQL and obtain the semantics of *SchemaSQL* as a simple extension. Our discussion in these sections 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 emp T declares T as a variable that ranges over the (tuples of the) relation emp (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 over which var ranges.

```
select T.name         select emp.name         select name
from   emp T          from   emp              from   emp
where  T.dept =       where  emp.dept =       where  dept =
         "Marketing"             "Marketing"             "Marketing"
      (a)                    (b)                     (c)
```

Figure 1: Syntax of Simple SQL Queries

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), emp 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.

In *SchemaSQL* we refer to relation `rel` in database `db` as `db::rel`, thus distinguishing between relations in different databases.

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`[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`.

[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. 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 a 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*.

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

Example 2.1 *[Comparing salaries in* univ-A *and* univ-B*]* List the departments in univ-A that pay a higher salary floor to their technicians compared with the same department in univ-B.

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

univ-A

salInfo

category	dept	salFloor
Prof	CS	65,000
AssocProf	CS	50,000
Technician	CS	45,000
Prof	Math	60,000
AssocProf	Math	55,000
Technician	Math	45,000

univ-B

salInfo

category	CS	Math
Prof	55,000	65,000
AssocProf	50,000	55,000
Technician	43,000	44,000

univ-C

CS

category	salFloor
Prof	60,000
AssocProf	55,000
Technician	42,000

Math

category	salFloor
Prof	70,000
AssocProf	60,000
Technician	46,000

univ-D

salInfo

dept	Prof	AssocProf	Technician
CS	75,000	60,000	40,000
Math	60,000	45,000	38,000

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

Example 2.2 *[Comparing salaries in* `univ-C` *and* `univ-D`*]* List the departments in `univ-C` that pay a higher salary floor to their technicians compared with the same department in `univ-D`.

```
(Q2)   select  RelC
       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
```

Explanation: The variable `RelC` is declared as a relation name in the database `univ-C`. Note that in this database there is one relation per department, and the relation name coincides with department name. Variable `C` is then declared as a tuple variable on this (variable) relation `RelC`. The variable `D` is an (SQL-like) tuple variable ranging over the relation `univ-D::salInfo`. Note that in `univ-D::salInfo` categories are represented by attribute names, whose domains consist of the salary floors of the corresponding category. Hence, `D.technician` is the salary floor for category `technician` (for the tuple represented by tuple variable `D`). ■

3 Semantics I: Fixed Output Schema

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.

We first quickly review the semantics of SQL and express it in a manner that makes it possible to realize the semantics of *SchemaSQL* by a simple extension.

3.1 SQL Semantics Reviewed

A query in SQL assumes a *fixed* scheme for the underlying database, and maps each database to a relation over a fixed scheme, called the *output* scheme associated with the query. Let $\mathcal{D}$ be the set of all database instances over a fixed scheme. Let a query Q be of the form[2]

```
select    attrList, aggList
from      fromList
where     whereConditions
group by  groupbyList
having    havingConditions
```

Let $\mathcal{R}$ be the set of all relations over the output scheme of the query Q. The query Q induces a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from databases to relations over a fixed scheme, defined as follows. Let $D \in \mathcal{D}$ be an input database, and $\mathcal{T}_D$ the set of all tuples appearing in any relation in D. Let τ be the set of tuple variables occurring in Q. We define an *instantiation* as a function $\imath : \tau \rightarrow \mathcal{T}_D$ which instantiates each tuple variable in Q to some tuple over its appropriate range. The conditions `whereConditions` in the `where` clause induce a boolean function, denoted $sat_{\mathtt{w}}(\imath, Q)$, on the set of all instantiations, reflecting whether the conditions are satisfied by an instantiation. This is defined in the obvious manner. Let $\mathcal{I}_Q = \{\imath \mid \imath \text{ is an instantiation for which } sat_{\mathtt{w}}(\imath, Q) = true\}$ denote the set of instantiations satisfying `whereConditions`. The query assembles each satisfying instantiation into a tuple for the answer relation, via a tuple assembly function, defined below. Let $\mathcal{T}_{\mathtt{attrList}}$ denote the set of all tuples over the scheme `attrList` such that each value in each tuple appears in the database D. Then the tuple assembly function is a function $tuple_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\mathtt{attrList}}$ defined as follows.

$$tuple_Q(\imath) = \bigotimes_{\text{“}t.A\text{”} \in \mathtt{attrList}} \imath(t)[A]$$

[2]In this paper, we restrict attention to single block *SchemaSQL* queries. In keeping with this, we only consider single block SQL queries here.

Here, the predicate "$t.A$" $\in$ `attrList` indicates the condition that the attribute denotation $t.A$ literally appears in the list of attributes `attrList` in the `select` statement. The symbol $\otimes$ denotes concatenation, and $\imath(t)[A]$ denotes the restriction of the tuple $\imath(t)$ to the attribute A. For an instantiation $\imath$, $tuple_Q(\imath)$ produces a tuple over the attributes `attrList` listed in the `select` statement. Suppose Q is a regular query, *i.e.* a query without aggregation. In this case, the `aggList` is empty and the `having` and `group by` clauses are absent. So, the *result of a* SQL *query without aggregation* is captured by the function

$$Q(D) = [tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q]$$

Note that SQL has a *multiset* semantics. We use $[\ldots]$ instead of $\{\ldots\}$ to denote multisets.

To account for aggregation, we need the following extension.

Definition 3.1 *[Equivalence relation induced by* `group by` *clause]* For $\imath$, $\jmath \in \mathcal{I}_Q$, $\imath \sim \jmath$ iff $\forall$"$t.A$" $\in$ `groupbyList`, $\imath(t)[A] = \jmath(t)[A]$. It is easy to see that $\sim$ is an equivalence relation on $\mathcal{I}_Q$. In words, two instantiations (satisfying the conditions in the `where` clause) are $\sim$-equivalent provided they agree on all attributes appearing in the `group by` clause. ■

The conditions `havingConditions` in the `having clause` are a boolean combination of atomic conditions of the form `agg(t.att) relOp c` and of the form `agg1(t.att1) relOp agg2(t.att2)`. Intuitively, we are only interested in those equivalence classes of $\sim$ that satisfy `havingConditions`. For an equivalence class e of $\sim$, let $sat_{\mathtt{h}}(e, Q) = true$ denote that e satisfies `havingConditions`. Note that, in practice, checking whether $sat_{\mathtt{h}}(e, Q) = true$ for an equivalence class e requires the calculation of any additional aggregations that may appear in the `having` clause but not necessarily in the `select` clause. We have the following

Definition 3.2 *[Valid equivalence classes]*

$$\mathcal{E}_Q = \{e \mid e \text{ is an equivalence class of } \sim \text{ and } sat_{\mathtt{h}}(e, Q) = true\}$$

■

Let $\mathcal{T}_{\mathtt{aggList}}$ denote the set of all tuples over the scheme `aggList`. We define a function $aggregate_Q$: $\mathcal{E}_Q \rightarrow \mathcal{T}_{\mathtt{aggList}}$ as follows.

$$aggregate_Q(e) = \bigotimes_{\text{“}agg(t.B)\text{”} \in \mathtt{AggList}} agg([\jmath(t)[B] \mid e \in \mathcal{E}_Q \text{ and } \jmath \in e])$$

For a given equivalence class $e \in \mathcal{E}_Q$, $aggregate_Q$ considers all instantiations in e, and, for each aggregate operation, say agg, indicated on the attribute $t.B$ in `aggList`, it performs the operation *agg* on the *multiset* of values associated with this attribute by instantiations in e. Again, we use $[\ldots]$ to denote multisets.

Now, we are ready to describe the tuple assembly associated with aggregate queries. Let Q be a query involving aggregation. Define a function $aggtuple_Q : \mathcal{E}_Q \rightarrow \mathcal{T}_{\texttt{attrList}} \times \mathcal{T}_{\texttt{aggList}}$ as follows.

$$aggtuple_Q(e) = tuple_Q(\imath) \bigotimes aggregate_Q(e)$$

where $\imath$ is any instantiation in e. Note that SQL requires the set of attributes in `attrList` to be a subset of those in `groupbyList`. Hence, $tuple_Q(\imath)$ is the same for all instantiations $\imath \in e$, and thus $aggtuple_Q(e)$ is well-defined, for any equivalence class e in $\mathcal{E}_Q$.

Finally, the *result of an aggregate* SQL *query* is captured by the function

$$Q(D) = [aggtuple_Q(e) \mid e \in \mathcal{E}_Q]$$

3.2 Semantics of *SchemaSQL* Queries

The semantics of *SchemaSQL* is obtained as a natural extension of that of SQL. A *SchemaSQL* query Q is of the form:

```
select    itemList, aggList
from      fromList
where     whereConditions
group by  groupbyList
having    havingConditions
```

where `itemList` is a list of db-name, rel-name, attr-name, and domain variables; `aggList` is a list of expressions of the form `agg(X)` where `agg` is an aggregate function, and `X` is a db-name, rel-name, attr-name, or domain variable; `fromList` is a list of variable declarations; `groupbyList` is a list of db-name, rel-name, attr-name, and domain variables; and the conditions in the where and having clauses are analogous to SQL. The main difference with an SQL query is the availability of additional variable types, in addition to the usual SQL tuple variables.

Let $\mathcal{D}$ be the set of all federation database instances. Let $\mathcal{R}$ be the set of all relations over the output scheme of the query Q. The query Q induces a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from federations to relations, defined as follows. Let $D \in \mathcal{D}$ be an input federation, and $\mathcal{O}_D$ the set of all items (database names, relation names, attribute names, tuples, and values) appearing in D. Let V be the set of variables occurring in Q. We define an *instantiation* as a function $\imath : V \rightarrow \mathcal{O}_D$ which instantiates each variable in Q to some item over its appropriate range. *Throughout this* *paper, we assume that any instantiation $\imath$ is extended in such a way that for a literal constant c,* $\imath(c) = c$. In defining the semantics of *SchemaSQL* queries, we will find the following definitions useful. Identifiers in typewrite font (*e.g.* `db`) can be constants or variables.

Definition 3.3 *[Admissibility]* An instantiation $\imath$ is *admissible* provided, it satisfies the following conditions.

- whenever `-> D` is a declaration in the `from` clause, $\imath$(`D`) is the name of a database in the federation.
- whenever `db-> R` is a declaration in the `from` clause, $\imath$(`R`) is the name of a relation in the database $\imath$(`db`).
- whenever `db::rel-> A` is a declaration in the `from` clause, $\imath$(`A`) is an attribute name in the relation $\imath$(`rel`) in the database $\imath$(`db`).
- whenever `db::rel T` is a declaration in the `from` clause, $\imath$(`T`) is a tuple in the relation $\imath$(`rel`) in the database $\imath$(`db`).
- whenever `db::rel.attr V` is a declaration in the `from` clause, $\imath$(`V`) is a value that appears in the column $\imath$(`attr`) of the relation $\imath$(`rel`) in the database $\imath$(`db`).
- whenever `T.attr V` is a variable declaration in the `from` clause, $\imath$(`V`) $=$ $\imath$(`T`)[$\imath$(`attr`)]. ■

Definition 3.3 precisely captures the notion of an appropriate range for a variable.

Definition 3.4 *[Validity]* Let $sat(\imath, Q)$ be a boolean function on the set of all instantiations, induced by the conditions in the `where` clause. An instantiation $\imath$ is *valid* provided (a) it is admissible, and (b) $sat(\imath, Q)$ is true. ■

The conditions in the where clause as well as additional conditions induced by the presence of certain patterns involving tuple variables is captured in Definition 3.4. We now define,

$$\mathcal{I}_Q = \{\imath \mid \imath \text{ is a valid instantiation}\} \tag{1}$$

The query assembles each satisfying instantiation into a tuple for the answer relation, as follows. Let $\mathcal{T}_{\texttt{itemList}}$ denote the set of all tuples over the scheme `itemList` such that each value in each tuple appears in the federation D. Then the tuple assembly function is a function $tuple_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\texttt{itemList}}$ defined as follows.

$$tuple_Q(\imath) = \bigotimes_{s \in \texttt{itemList}} \imath(s) \tag{2}$$

where s is a db-name, rel-name, attr-name, or domain variable. For an instantiation $\imath$, $tuple_Q(\imath)$ produces a tuple over the list of objects `itemList` listed in the `select` statement.

3.2.1 Queries with Fixed Output Schema and no Aggregation

Suppose Q is a *SchemaSQL* query without aggregation. In this case the result of the query is captured by the function

$$Q(D) = [tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q] \tag{3}$$

Similar to SQL, *SchemaSQL*'s semantics is based on multisets. Multisets are distinguished from sets with the use of [..] instead of {..}.

It is not hard to see that the formal semantics captured by these definitions exactly correspond to the intuitive semantics discussed earlier for Queries $Q1$ and $Q2$ of Section 2.

3.3 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. Before we illustrate these points with examples, we provide a formal development of the semantics.

3.3.1 Semantics of Aggregation with Fixed Output Schema

Let Q be a *SchemaSQL* query involving aggregation. Similar to the development of SQL semantics, we define the equivalence relation $\sim$ on the instantiations $\mathcal{I}_Q$, and the set $\mathcal{E}_Q$ of equivalence classes of $\sim$ that satisfy the `havingConditions` for *SchemaSQL* queries.

Definition 3.5 *[Equivalence relation induced by* `group by` *clause]* For $\imath,\ \jmath \in \mathcal{I}_Q$, $\imath \sim \jmath$ iff $\forall$"v" $\in$ `groupbyList`, $\imath(v) = \jmath(v)$. It is straightforward to see that $\sim$ is an equivalence relation on $\mathcal{I}_Q$. Intuitively, two instantiations are $\sim$-equivalent provided they agree on all variables appearing in the `group by` clause. We denote by $\mathcal{E}_Q$ the set of equivalence classes of $\mathcal{I}_Q$ under $\sim$. ■

Definition 3.6 *[Valid equivalence classes]*

$$\mathcal{E}_Q = \{e \mid e \text{ is an equivalence class of } \sim \text{ and } sat_{\mathtt{h}}(e, Q) = true\} \tag{4}$$

where $sat_{\mathtt{h}}(e, Q) = true$ means equivalence class s satisfies the `havingConditions`. ■

Let $\mathcal{T}_{\mathtt{aggList}}$ denote the set of tuples over the scheme `aggList`. We define a function $aggregate_Q$: $\mathcal{E}_Q \rightarrow \mathcal{T}_{\mathtt{aggList}}$ as follows.

$$aggregate_Q(e) = \bigotimes_{\text{“}agg(v)\text{”} \in \mathtt{aggList}} agg([\jmath(v) \mid e \in \mathcal{E}_Q \text{ and } \jmath \in e]) \tag{5}$$

For a given equivalence class $e \in \mathcal{E}_Q$, $aggregate_Q$ considers all instantiations in e, and, for each aggregate operation, say agg, indicated on the variable v in `aggList`, it performs the operation agg on the *multiset* of values associated with this variable by instantiations in e.

Let Q be a query involving aggregation. We define the tuple assembly function $aggtuple_Q : \mathcal{E}_Q \rightarrow \mathcal{T}_{\mathtt{itemList}} \times \mathcal{T}_{\mathtt{aggList}}$ as follows.

$$aggtuple_Q(e) = tuple_Q(\imath) \bigotimes aggregate_Q(e) \tag{6}$$

where $\imath$ is any instantiation in e. In *SchemaSQL* we require the set of variables in `itemList` to be a subset of those in `groupbyList`. Hence, $tuple_Q(\imath)$ is the same for all instantiations $\imath \in e$, and thus $aggtuple_Q(e)$ is well-defined, for any equivalence class e in $\mathcal{E}_Q$.

Finally, the result of a *SchemaSQL* query with aggregation is captured by the function

$$Q(D) = [aggtuple_Q(e) \mid e \in \mathcal{E}_Q] \tag{7}$$

We now provide some examples illustrating aggregation in *SchemaSQL*.

Example 3.1 *[Horizontal aggregation and aggregation through multiple relations]* The query

```
(Q3)  select    T.category, avg(T.D)
      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.*, using the condition (D = "Math" or D = "CS" or ...)[3]. By contrast, the query

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

[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 6.2, addresses this issue.

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 Semantics II: 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 contents of 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 3.1, Q3 and Q4). In this section, we shall see that the combination of dynamic output schema and meta-data variables, namely, db-name, rel-name, and attr-name variables, allows us to express more powerful aggregations such as vertical aggregation on a *variable number of columns* and aggregation on a *variable number of blocks* as well.

4.1 Restructuring Without Aggregation

We first illustrate the ideas and expressive power of *SchemaSQL* for performing restructuring, using examples. Formal development will follow.

Example 4.1 *[Restructuring* `univ-B` *database into the schema of* `univ-A` *database]* 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`.

```
(Q5)    create view  BtoA::salInfo(category, dept, salFloor) as
        select       T.category, D, T.D
        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 corresponding to each tuple in the `univ-B::salInfo` format, `Q(5)` 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 *[Restructuring* `univ-A` *database into the schema of* `univ-B` *database]* This view definition restructures data in `univ-A::salInfo` into the format of the schema `univ-B::salInfo`.

```
(Q6)   create view  AtoB::salInfo(category, D) as
       select       A.category, A.salFloor
       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. ■

As demonstrated by the previous examples, the semantics of restructuring in the context of a dynamically declared output schema has two aspects to it: `(i)` the determination of the output schema itself, and `(ii)` the formatting of data to conform to the schema determined in `(i)`. In this section, we formalize these concepts. We illustrate our development of the semantics by revisiting Example 4.2.

Let a *SchemaSQL* query Q be a view definition of the form

```
create view  db::rel(attr1, ..., attrn) as
select       obj1, ..., objn
from         fromList
where        whereConditions
```

Where `db`, `rel` are constants or db-name, rel-name, att-name, or domain variables; `attr1`, ..., `attrn` are all constants or one of them is a db-name, rel-name, att-name, or domain variable and the rest are constants; and `obj1`, ..., `objn` are db-name, rel-name, att-name, or domain variables.

We first define some useful notions. Recall that $\mathcal{I}_Q$ is the set of valid instantiations (of the variables declared in the `from` clause) that satisfy the conditions in the `where` clause, as defined in Section 3.2.

Definition 4.1 *[Instantiations contributing to the same output relation]* For two instantiations $\imath, \jmath \in \mathcal{I}_Q$, we define $\imath \equiv \jmath$, provided $\imath(\mathtt{db}) = \jmath(\mathtt{db})$ and $\imath(\mathtt{rel}) = \jmath(\mathtt{rel})$. Clearly $\equiv$ is an equivalence relation. ■

Determination of output schema: Each instantiation $\imath \in \mathcal{I}_Q$ produces a view of a database namely, $\imath(\mathtt{db})$, containing a relation named $\imath(\mathtt{rel})$, whose scheme consists of the attribute set $attrset_Q(\imath) = \{\jmath(\mathtt{attr}) \mid \mathtt{attr} \in \{\mathtt{attr1}, ..., \mathtt{attrn}\},\ \jmath \in \mathcal{I}_Q,\ \jmath \equiv \imath\}$. Thus, each $\equiv$-equivalence class of instantiations defines one relation scheme in the output view. *E.g.*, in Example 4.2, all instantiations $\imath \in \mathcal{I}_Q$ are $\equiv$-equivalent, and this one equivalence class produces a view containing a database called `AtoB`, containing one relation named `salInfo`, with the scheme $\{\mathtt{category}, \mathtt{CS}, \mathtt{Math}, ...\}$.

Formatting data to fit the schema: There are two aspects to this. Firstly, the output computed by the *SchemaSQL* query defining the view has to be properly *allocated* to conform to the output schema declared in the `create view` statement. This by itself might in general result in null values, which we can eliminate by identifying maximal subsets of "related" tuples and "merging" them. These ideas are made precise below.

As seen above, an instantiation $\imath \in \mathcal{I}_Q$ contributes to a view of a database $\imath(\mathtt{db})$ containing a relation named $\imath(\mathtt{rel})$ with a scheme given by $attrset_Q(\imath)$. The instantiations $\equiv$-equivalent to $\imath$ contribute to a relation, $allocate_Q(\imath)$, over the attribute set $attrset_Q(\imath)$, as follows. For each instantiation $\jmath \in \mathcal{I}_Q$ such that $\jmath \equiv \imath$, $allocate_Q(\imath)$ contains a tuple t, defined as follows. Let $A \in attrset_Q(\imath)$. Then

$$t[A] = \begin{cases} \jmath(\mathtt{objk}), & \text{whenever } A = \jmath(\mathtt{attrk}) \\ null, & \text{otherwise.} \end{cases}$$

Figure 3(i) shows $allocate_{Q6}(\imath)$ for the view (Q6) defined in Example 4.2, where $\imath$ is any instantiation (recall all of them are $\equiv$-equivalent).

Secondly, merging of tuples in $allocate_Q(\imath)$ is formalized as follows. Let DOM denote the union of all domains of all attributes of all relations involved in the federation, together with the null value,

AtoB

salInfo (allocated)

category	CS	Math
Prof	65,000	null
AssocProf	50,000	null
Technician	45,000	null
Prof	null	60,000
AssocProf	null	55,000
Technician	null	45,000

(i)

AtoB

salInfo

category	CS	Math
Prof	65,000	60,000
AssocProf	50,000	55,000
Technician	45,000	45,000

(ii)

Figure 3: (i) The relation $allocate_{Q6}(\imath)$ and (ii) the final result after merging

null. Define a partial order on DOM, by setting $null \leq v$, $\forall v \in$ DOM. In particular, note that any two distinct non-null values are incomparable. The *least upper bound*, *lub*, of two values in DOM is defined in the obvious way.

$$lub(u, v) = \begin{cases} u, & \text{if } v \leq u \\ v, & \text{if } u \leq v \\ undefined, & \text{otherwise.} \end{cases}$$

We now have the following

Definition 4.2 *[Merging a pair of tuples]* Two tuples t_1, t_2 over a relation scheme $R = \{A_1, \ldots, A_n\}$ are mergeable provided for each $i = 1, \ldots, n$, either $t_1[A_i] = t_2[A_i]$, or at least one of $t_1[A_i]$ or $t_2[A_i]$ is a null. Suppose t_1 and t_2 are mergeable. Then their merge, denoted $t = t_1 \odot t_2$, is defined as $t[A_i] = lub(t_1[A_i], t_2[A_i])$, $i = 1, \ldots, n$. ■

Clearly, the operator $\odot$ is commutative and associative, and it can be easily extended to any set of *mergeable* tuples. It will be convenient below to extend the operator $\odot$ to any relation containing an arbitrary (*i.e. not* necessarily mergeable) set of tuples. The idea is to partition the relation into sets of mergeable tuples, and merge the tuples in each partition. Notice that mergeability is not a transitive relation: e.g., tuple $t_1 = (a, \perp)$ is mergeable with each of the tuples $t_2 = (a, b)$ and $t_3 = (a, b')$, which themselves are not mergeable. Thus, in partitioning a relation into sets of mergeable tuples, choice arises in grouping tuples. From the point of view of the meaning of the tuples, the choices may be made arbitrarily. So, for instance, we may merge t_1 and t_2 above (and leave t_3 alone) or merge t_1 and t_3. While the results look physically different, the meaning is the same. Intuitively, we need any "maximal" partition of a relation that respects mergeability, formalized below.

Definition 4.3 *[Maximal partitions]* A partition $\mathcal{P} = \{r_1, \ldots, r_k\}$ of a relation r is *valid* provided for each block r_i, the tuples in r_i are pairwise mergeable, $1 \leq i \leq k$. Define a partial order $\leq$ on partitions as follows:[4] $\mathcal{P}_1 \leq \mathcal{P}_2$ provided $\forall r_i \in \mathcal{P}_1 : \exists s_j \in \mathcal{P}_2 : r_i \subseteq s_j$. A partition $\mathcal{P}$ is a *maximal* valid partition provided for every valid partition $\mathcal{P}'$: $\mathcal{P} \leq \mathcal{P}' \Rightarrow \mathcal{P}' = \mathcal{P}$. ■

The following easily established result reveals the significance of maximal valid partitions.

Lemma 4.1 *[Maximal partitions]* Let r be a relation and $\mathcal{P}$ be any maximal valid partition of r. Then the following holds: (i) for each block $r_i \in \mathcal{P}$, the tuples in r_i are pairwise mergeable; (ii) for any two distinct blocks $r_i, r_j \in \mathcal{P}$, the tuples $r_i \cup r_j$ are not pairwise mergeable, i.e. there is a pair of tuples $t, t' \in r_i \cup r_j$ such that they are not mergeable. ■

We suppress the obvious proof. Finally, we can define the result of applying a merge operator to a relation as follows.

Definition 4.4 *[Merging a relation]* Let r be a relation and $\mathcal{P} = \{r_1, \ldots, r_k\}$ be any maximal valid partition of r. Then, $\bigodot r = \{ \ \bigodot(r_i) \mid r_i \in \mathcal{P} \ \}$. ■

Finally, we can define the semantics of view definitions in *SchemaSQL* without aggregation as follows:

Definition 4.5 *[Semantics of restructuring views without aggregation]* Let Q be the *SchemaSQL* query that defines a view V. Then the materialization of V consists of one relation for each $\equiv$-equivalence class of instantiations in $\mathcal{I}_Q$. For an equivalence class $[\imath]$, for any $\imath \in \mathcal{I}_Q$, the corresponding relation is determined by $\bigodot allocate_Q(\imath)$. ■

As an example, the final output produced by the view definition (Q6) in Example 4.2 is a view of a database `A2B` containing a relation `salInfo(category, CS, Math)` as shown in Figure 3(ii).

4.2 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 one or more 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 or blocks of data, determined dynamically by the input instance. The following examples illustrate this point.

[4] This is the dual of the standard refinement ordering on partitions.

Example 4.3 *[Block aggregation: fixed and dynamic output schemas]* Suppose that in the database `univ-D` in Figure 2, there is an additional relation `faculty(dname, fname)` relating each department to its faculty, such as, (math, arts and sciences), (physics, arts and sciences), (cs, engineering and comp sci), etc.

The first query uses a fixed output schema. The second query is identical to the first except for a `create view` expression that defines a dynamic output schema:

```
(Q7)   select    U.fname, avg(T.C)
       from      univ-D::salInfo-> C,
                 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.

```
(Q8)   create view  averages::salInfo(faculty, C) as
       select       U.fname, avg(T.C)
       from         univ-D::salInfo-> C,
                    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. The schema of the output has the form {`faculty, Prof, AssocProf, Technician, ...`}, namely, the variable `C` in the view definition represents its value in the set of valid instantiations, and hence the output schema depends on the input data. ■

Example 4.4 *[Aggregation on a variable number of blocks]* Let's add yet another relation to the database `univ-D` of Figure 2. The relation `empType(category,type)` lists the categories, and their types, *e.g.* (prof, teaching faculty), (assoc prof, teaching faculty), (technician, technical staff), (secretary, administrative staff), etc. Consider the query:

```
(Q9)   create view   averages::salInfo(faculty, Y) as
       select        U.fname, avg(T.C)
       from          univ-D::salInfo-> C,
                     univ-D::salInfo T,
                     univ-D::faculty U,
                     univ-D::empType E,
                     E.type Y
       where         C <> "dept" and
                     T.dept = U.dname and
                     E.category = C
       group by      U.fname
```

The following figure depicts relation `univ-D::salInfo`, and the "blocks" upon which the aggregation is carried out. Each block corresponds to a faculty and an employee type.

		category type$_1$	$\cdots$	*category type*$_n$
	dept	**<categories>**		
faculty$_1$		faculty$_1$ & category type$_1$	... ...	faculty$_1$ & category type$_n$
$\cdots$		...	...	...
faculty$_k$		faculty$_k$ & category type$_1$	... ...	faculty$_k$ & category type$_n$

Figure 4: `univ-D::salInfo` relation. The schema components appear in bold face font. The "faculty$_i$ & category type$_j$" rectangles are the blocks upon which the aggregation is carried out.

The schema of the output relation is

$$\texttt{averages} :: \texttt{salInfo}(\texttt{faculty}, \texttt{teachingfaculty}, \texttt{technicalstaff}, \texttt{administrativestaff}, ...)$$

and each tuple in the output lists a faculty, such as "arts and sciences", and the corresponding average salary floors for the employees in each of the category types "teaching faculty", "technical staff", "administrative staff", etc. ∎

4.2.1 Semantics of Aggregation with Dynamic View Definition

The semantics of restructuring (via view definition) with aggregation involves putting together the ideas behind each of these operations. Intuitively, as explained in Section 4.1, the instantiations $\equiv$-equivalent to $\imath \in \mathcal{I}_Q$ produce (in the view) one relation in a database whose scheme consists of the attributes $attrset_Q(\imath)$, as defined in that section. The tuples for this relation are obtained by computing the aggregations listed in the `aggList` in the `select` statement with respect to each valid equivalence class of instantiations which agree on the variables listed in the `group by` clause *as well as* on the variables appearing in the `create view` statement, and then performing the necessary merging. The reason for including the variables in the create view statement is that they specify an implicit group by. This intuition is formalized below. Consider the *SchemaSQL* view definition Q below:

```
create view  db::rel(attr1, ..., attrn) as
select       itemList, aggList
from         fromList
where        whereConditions
group by     groupbyList
having       havingConditions
```

Definition 4.6 *[Equivalence relation induced by group by and create view clauses]* For two instantiations $\imath, \jmath \in \mathcal{I}_Q$, we define $\imath \# \jmath$, provided for each $o \in$ `groubyList`, $\imath(o) = \jmath(o)$, and for each variable X occurring in the `create view` statement, $\imath(X) = \jmath(X)$. Clearly, $\#$ is an equivalence relation. ■

Similar to previous cases, we denote by $\mathcal{E}_Q$ the set of equivalence classes of $\mathcal{I}_Q$ under $\#$ that satisfy the `havingConditions`:

Definition 4.7 *[Valid equivalence classes]*

$$\mathcal{E}_Q = \{e \mid e \text{ is an equivalence class of } \# \text{ and } sat_{\texttt{h}}(e, Q) = true\}$$

■

The notions of $aggregate_Q(e)$, $aggtuple_Q(e)$, and $sat_{\texttt{h}}(e, Q)$ for an equivalence class $e \in \mathcal{E}_Q$, are defined analogously to the way they were defined in Section 3.3. The only difference is that the equivalence relation $\#$ is used instead of $\sim$.

$$\begin{aligned} aggregate_Q(e) &= \bigotimes_{\text{``}agg(o)\text{''} \in \texttt{aggList}} agg([\jmath(o) \mid e \in \mathcal{E}_Q \text{ and } \jmath \in e]) \\ aggtuple_Q(e) &= tuple_Q(\imath) \bigotimes aggregate_Q(e) \end{aligned}$$

where $\imath$ is any instantiation in the equivalence class e.

The concept of *allocating* tuples computed above according to the various output schemas dynamically created by the instantiations can be formalized in a way similar to what was done in Section 4.1, and we suppress these obvious details for brevity. Recall that the schema of each output relation is determined by one equivalence class of $\mathcal{I}_Q$ under the $\equiv$ relationship. Let $aggallocate_Q\ (\imath)$ denote the allocated relation determined by the $\equiv$-equivalence class of $\imath \in \mathcal{I}_Q$. The concept of merging, defined in Definitions 4.2 and 4.4, can now be directly applied to compute the final output. Thus, for each instantiation $\imath \in \mathcal{I}_Q$, the $\equiv$-equivalence class of $\imath$ contributes to the relation $\bigodot aggallocate_Q\ (\imath)$, over the schema $\imath(\mathtt{db}) :: \imath(\mathtt{rel})(attrset_Q(\imath))$.

As an example, it is easy to verify that the view defined by Q8 indeed computes for each faculty, the category-wise floor salary averages. Before closing this section, we note that the combination of dynamic output schema declaration with *SchemaSQL* 's aggregation mechanism makes it possible to express many other novel forms of aggregation as well. Further, note that the examples in this and previous sections were in the context of a single database, demonstrating some of the applications of *SchemaSQL* for a single database environment.

5 Implementation

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 of SQL 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 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).

Figure 5 depicts our architecture for implementing *SchemaSQL*. At a high level, 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 against the answers collected above, 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 in Section 6.1). Algorithms 5.1 and 5.2 give detailed accounts of our query processing strategy for the fixed and dynamic output schema,

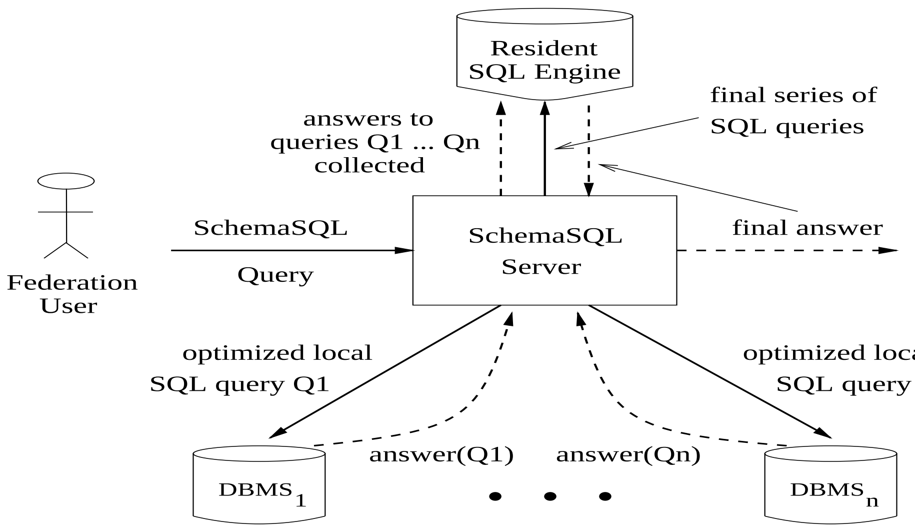

Figure 5: SchemaSQL – Implementation Architecture

respectively.

Query processing in a *SchemaSQL* environment consists of two major phases. In the first phase, tables called *VIT*'s (Variable Instantiation Tables) corresponding to the variable declarations in the from clause of a *SchemaSQL* query are generated. The schema of a VIT consists of all the variables in one or more variable declarations in the from clause, while 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 a series of one or more SQL queries against the VITs with the property that the result of the series of queries against the VITs is identical to that of the original *SchemaSQL* query against the federation. The result so obtained is appropriately restructured for presentation to the user.

For ease of exposition, we first consider (Section 5.1) *SchemaSQL* queries with a fixed output schema and then present the algorithm for handling queries with a dynamic output schema (Section 5.2).

5.1 Processing *SchemaSQL* Queries with a Fixed Output Schema

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*.

Our algorithm below considers *SchemaSQL* queries with a fixed output schema, possibly with aggregation. The algorithm consists of two phases as explained above. In order to facilitate the reader to follow the development, we break the presentation of the algorithm after each phase and follow it with an illustrative example.

Algorithm 5.1 *SchemaSQL Query Processing – Fixed Output Schema*
Input: A *SchemaSQL* query with aggregation and with a fixed output schema.
Output: The result of the query.
Method:

Phase I: 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.

Phase II: Rewrite the original *SchemaSQL* query against the federation into an "equivalent" query against the set of VIT relations and compute it using the resident SQL server.

Phase I

(0) Rewrite the input *SchemaSQL* statement into the following form such that the conditions in the where and having clauses are in conjunctive normal form.

select	itemList, aggList
from	$\langle range_1 \rangle\ V_1,\ \ldots,\ \langle range_k \rangle\ V_k$
where	$\langle cond_1 \rangle$ and ... and $\langle cond_m \rangle$
group by	groupbyList
having	havingConditions

We assume an ordering on the variable declarations in the from clause: If a variable V_j appears in the range of another variable V_i, *i.e.* in $\langle range_i \rangle$, then we say V_i *depends on* V_j, denoted $V_j \prec V_i$. We will assume that whenever $V_j \prec V_i$, the declaration of V_j comes before that of V_i in the from clause. It is trivial to reorder the declarations in a given *SchemaSQL* query to meet this condition.

(1) Consider each variable declaration $\langle range_i \rangle\ V_i, \quad 1 \leq i \leq k$. The following cases arise:

(a) V_i is a meta-variable:

In this case, all variables in the declaration $\langle range_i \rangle\ V_i$ must be meta-variables. As an optimization issue, notice that it is sufficient to create VITs for just those meta variables that are maximal (among meta-variables) with respect to the ordering $\prec$: As we will discuss below, the VIT for a maximal meta-variable V_i also contains the instantiations for all meta-variables V_j such that $V_j \prec V_i$. Create a VIT for V_i, VIT_{V_i} as follows: the schema of VIT_{V_i} consists of V_i and all variables V_j such that $V_j \prec V_i$; the contents of VIT_{V_i} are obtained using an appropriate SQL query against the FST. For example, assume the from clause contains the declarations `-> D` and `D::r-> A` declaring db-name and attr-name variables `D` and `A`, where `r` is a constant (relation name). The VIT for `D` will have the schema {`D, A`}. Its contents can be obtained using the SQL query

```
select  db-name as D, attr-name as A
from    FST
where   rel-name = r
```

Notice that $VIT_{\mathtt{D}}$ includes instantiations for the attr-name variable `A`, and thus we can save on creating a separate VIT for `A`. We can further optimize the query processing by enforcing conditions of the form U `relOp` c (or c `relOp` U) in the `where` clause of the original *SchemaSQL* query, if any, at this time, provided U is a variable in the schema of VIT_{V_i}, `relOp` is a comparison operator, and c is a constant.

(b) V_i is a tuple variable:

Let the declaration for `T` be `db::rel T`, where `db` (`rel`) can be constant or variable. Determine the schema of VIT_T as follows.

(i) Include all meta-variables appearing in `db::rel`, if any, in the schema of $VIT_{\mathtt{T}}$.

(ii) If `T.a`, where `a` is a constant, appears in the query (in the `select`, `where`[5], `group by`, or `having` clauses), include `Ta` in the schema of $VIT_{\mathtt{T}}$. Here, the identifier `Ta` is obtained by concatenating the tuple variable `T` with the attribute name `a`.

(iii) For each domain variable declaration `T.a V`, where `a` is a constant, include `V` in the schema of $VIT_{\mathtt{T}}$.
For each domain variable declaration `T.A V`, where `A` is an attr-name variable, include both `A` and `V` in the schema of $VIT_{\mathtt{T}}$.

(iv) If, for an attr-name variable `A`, `T.A` appears in the `select`, `where`, `group by`, or `having` clauses, then include both `A` and `TA` in the schema of $VIT_{\mathtt{T}}$.

Next, obtain the contents of $VIT_{\mathtt{T}}$ by submitting SQL queries to component databases, and computing their union, as follows.

(i) Identify the meta-variables in the schema of $VIT_{\mathtt{T}}$. These include any variables in `db::rel`, and possibly some attr-name variables. Obtain the bindings for these variables by executing an SQL query on their VITs.

(ii) For each binding obtained, generate one SQL query and submit it to the appropriate component database, in order to retrieve the relevant tuples from that database that pertain to the binding on the meta-variables. Push each condition of the form `T.att relOp c` (`c relOp T.att`), `T.A relOp c` (`c relOp T.A`), and `V relOp c` (`c relOp V`), where `att` and `c` are constants, `A` is an attr-name variable, and `V` is a domain variable declared using T, in the `where` clause of the original *SchemaSQL* query into the `where` clause of the SQL query to be submitted, above.

(iii) Finally, obtain the contents of $VIT_{\mathtt{T}}$ as the union of the SQL queries submitted to component databases. If only one component database is involved, *i.e.* if `db` in the declaration `db::rel`

[5] If `T.a` appears *only* in the `where` clause, in the form `T.a relOp c` where `c is a constant`, then there is no need to include `Ta` in the schema of $VIT_{\mathtt{T}}$.

T is a constant, then the union can be expressed *within* one SQL query submitted to the component database db.

This completes Phase I of the algorithm.

Notice that no separate VITs are needed for domain variables, as their instantiations are captured by the VITs for their associated tuple variables. Example 5.1 illustrates generation of VITs for meta and tuple variables.

Example 5.1 *[Illustrating Phase I: Generating VITs]* In this example, we illustrate the Phase I of the algorithm using a variant of query Q2 of Example 2.2:

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

The query contains declarations for one meta variable RelC, which is a rel-name variable, and two tuple variables C and D. Here is how Phase I of the algorithm will proceed to construct the VITs for this query. (Notice that as far as Phase I is concerned, aggregation is completely orthogonal.)

The schema of $VIT_{\texttt{RelC}}$ is {RelC} (one column). Its contents are generated using the following query to the FST:

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

The schema of $VIT_{\texttt{C}}$ is {RelC, CsalFloor}. Note that Ccategory need not be included: The where clause condition C.category = "technician" will be enforced within the SQL query to the component database univ-C. $VIT_{\texttt{C}}$ is generated as the union of SQL queries to database univ-C as follows. First, bindings for the meta-variable RelC are obtained from its VIT via the trivial query:

select RelC
from $VIT_{\texttt{RelC}}$

Let $\{r_1, \ldots, r_n\}$ be the answer to the above query. $VIT_{\texttt{C}}$ is obtained by submitting the following SQL query to the database univ-C.

```
select    'r1' as RelC, salFloor as CsalFloor
from      r1
where     category = "technician"
          UNION
...
          UNION
select    'rn' as RelC, salFloor as CsalFloor
from      rn
where     category = "technician"
```

Finally, the schema of $VIT_{\mathtt{D}}$ is {Ddept,Dtechnician}, and its contents are obtained via the following SQL query to the database `univ-D`:

```
select    dept as Ddept, technician as Dtechnician
from      salInfo
```

■

We now return to Phase II of Algorithm 5.1.

Algorithm 5.1 SchemaSQL *Query Processing (cont'd.)*

Phase II

(1) Execution of this phase happens in the *SchemaSQL* server. The *SchemaSQL* query is rewritten into an equivalent conventional SQL query on the VIT's generated in Phase I, as follows. To simplify the presentation, we use the concept of the *Joined Variable Instantiation Table* (JVIT). As the name suggests, JVIT is the (natural) join of the VITs generated for the various meta-variables and tuple variables during Phase I. In the creation of the JVIT, we enforce the remaining conditions, if any, in the `where` clause of the original *SchemaSQL* query. However, these conditions need to be syntactically rewritten so they make sense against the schema of the JVIT.

Recall that all `where` clause conditions involving a comparison with a constant were already enforced in Phase I. The remaining conditions must have the form `obj1 relOp obj2`, where `obj1` and `obj2` are either variables, or qualified attributes of the form `T.a` or `T.A`. We need to rewrite these conditions by renaming the obj's (variables and/or qualified attributes) as follows:

(i) Replace each occurrence of variable `V` by $VIT_{\mathtt{U}}$.V, where $VIT_{\mathtt{U}}$ is the VIT corresponding to `V`. It is possible that `U` is different from `V`, since VITs are generated only for certain variables and the instantiations for the remaining variables are obtained from these (see Phase I).

(ii) Replace each occurrence of a qualified attribute `T.a` (or `T.A`) by $VIT_{\mathtt{T}}$.`Ta` (or $VIT_{\mathtt{T}}$.`TA`).

(2) Assume $VIT_{V_1}, \ldots, VIT_{V_n}$ are all the VITs generated in Phase I. Let $\{U_1, \ldots, U_m\}$ be the union of their schemas. Properly qualify each U_i with the VIT whose schema contains V_i. For example, if U_i is `Ta`, then its qualified version is $VIT_{\texttt{T}}$.`Ta`. If U_i appears in the schemas of several VITs, break the tie arbitrarily for the purpose of qualification. Let $\{VIT_{U_1}.U_1, \ldots, VIT_{U_m}.U_m\}$ represent the qualified set. Then obtain the joined variable instantiation table using the following SQL query:

```
create view  JVIT(U_1, ..., U_m) as
select       VIT_{U_1}.U_1, ..., VIT_{U_m}.U_m
from         VIT_{V_1}, ..., VIT_{V_n}
where        (rewritten where clause) and (natural join conditions)
```

Here, "`rewritten where clause`" is obtained by rewriting the remaining `where` clause conditions as pointed out above, and "`natural join conditions`" consist of conditions of the form $VIT_{V_i}.attr = VIT_{V_j}.attr$ for all pairs of VITs VIT_{V_i} and VIT_{V_j} and all attributes *attr* that are common to the schemas of these VITs.

(3) Finally, generate a SQL query from the original *SchemaSQL* query to produce the final output, as follows.

```
select    itemList', aggList'
from      JVIT
group by  groupbyList'
having    havingConditions'
```

where, `itemList'` (resp., `aggList', groupbyList', havingConditions'`) is obtained from `itemList` (resp., `aggList, groupbyList, havingConditions`) in the original *SchemaSQL* query by replacing every occurrence of `T.a` by `Ta` and of `T.A` by `TA`.

Note that the generation of JVIT and the final SQL query can be combined in one step. We chose to divide it into two steps for clarity of exposition. Further, it also simplifies our presentation of the query processing algorithm for *SchemaSQL* queries with dynamic output schemas in Section 5.2. In fact, since we generated JVIT as a view, a natural option for the the resident SQL engine's query optimizer is to use query rewriting (rather than materializing JVIT) to process the final query in one step.

This completes Phase II of the algorithm.

We now present an example that illustrates our algorithm.

Example 5.2 *[Illustrating Phase II: computing the rest]* We continue with the processing of query `(Q2')` of Example 5.1. There we demonstrated how the VITs $VIT_{\texttt{RelC}}$, $VIT_{\texttt{C}}$ and $VIT_{\texttt{D}}$ are generated. Figure 6 shows these VITs for the federation of Figure 2.

VIT_{RelC}
RelC
CS
Math

VIT_C	
RelC	CsalFloor
CS	42,000
Math	46,000

VIT_D	
Ddept	Dtechnician
CS	40,000
Math	38,000

Figure 6: Example – Query Processing

The following SQL query produces the JVIT.

```
create view  JVIT(RelC, CsalFloor, Ddept, Dtechnician) as
select       VIT_RelC.RelC, VIT_C.CsalFloor, VIT_D.Ddept, VIT_D.Dtechnician
from         VIT_RelC, VIT_C, VIT_D
where        VIT_RelC.RelC = VIT_D.Ddept and
             VIT_C.CsalFloor > VIT_D.Dtechnician and
             VIT_RelC.RelC = VIT_C.RelC
```

In this query, the first two conditions in the `where` clause came from the (rewritten versions of) the first two conditions in the `where` clause of (Q2'). The last condition in the SQL query enforces natural join. Here is the final SQL query.

```
select    RelC, CsalFloor
from      JVIT
```

Notice that `C.salFloor` is replaced by `CsalFloor`.

Next, let us consider the effect of aggregation. As with Phase I, aggregation remains orthogonal to the processing being done in Phase II. As an illustration, consider the following query from Example 3.1.

(Q3): "find the average salary floor across all departments for each employee category in database `univ-B`." This query involves a horizontal aggregation.

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

Computation of the JVIT for this query is analogous to that for (Q2'): aggregation does not influence this step. Then, the final SQL query would be:

```
select    Tcategory, avg(TD)
from      JVIT
group by  Tcategory
```

■

The following lemma forms the basis for the correctness of our query processing strategy for both fixed and dynamic output schema.

Lemma 5.1 *[Computing valid instantiations]* Let JVIT be the joined variable instantiation table for a given *SchemaSQL* query Q. Then JVIT contains exactly the set of valid instantiations $\mathcal{I}_Q$ for query Q, restricted to the columns of JVIT.

Proof: It is straightforward to see that for each variable V, VIT_V generated in Phase I only contains *admissible* instantiation (restricted to V and variables on which V depends, in case V is a meta-variable, or restricted to all relevant attribute and domain variables related to V, in case V is a tuple variable). We only need to verify that the JVIT produced from these in Phase II will ensure that each instantiation in JVIT will satisfy all conditions in the `where` clause of Q. To see why this is true, notice that each condition of the form `x relOp y` where one of `x, y` is a constant, that constrains some column of VIT_V is enforced in the formation of VIT_V in Phase I. The remaining conditions are enforced while computing the join of the VITs. The natural join ensures that pieces of the same (global) instantiation are put together in the formation of JVIT. It follows that JVIT contains exactly the valid instantiations $\mathcal{I}_Q$, restricted to the columns of JVIT. ■

Theorem 5.1 *[Correctness for fixed output schema]* Algorithm 5.1 correctly computes answers to *SchemaSQL* queries with fixed output schemas.

Proof:

By Lemma 5.1, JVIT contains the set of valid instantiations $\mathcal{I}_Q$, restricted to the columns of JVIT. The final SQL query of Phase II assembles the result. For a *SchemaSQL* query with no aggregation, this final query is simply

```
select    itemList'
from      JVIT
```

which corresponds to the semantics given by Equations 2 and 3 (Section 3.2), repeated here for convenience

$$Q(D) = [tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q], \text{ where, } tuple_Q(\imath) = \bigotimes_{s \in \texttt{itemList}} \imath(s)$$

If the *SchemaSQL* query contains aggregation, then the final SQL query generated in the algorithm is:

```
select     itemList', aggList'
from       JVIT
group by   groupbyList'
having     havingConditions'
```

Semantics of *SchemaSQL* queries with aggregation was defined using the set $\mathcal{E}_Q$ of valid equivalence classes of $\mathcal{I}_Q$ with respect to the group by clause groupings of the original *SchemaSQL* query, which was characterized by the equivalence relation $\sim$. It is easy to verify that the group by clause of the rewritten SQL query above partitions JVIT, *i.e.* the set of valid instantiations $\mathcal{I}_Q$, exactly into equivalence classes of $\mathcal{I}_Q$ with respect to $\sim$. The having clause restricts these partitions to the valid equivalence classes, namely, $\mathcal{E}_Q$. Finally, the select clause assembles the result. This corresponds exactly to the semantics which was defined by Equations 5, 6, and 7 (Section 3.3), repeated here for convenience.

$$Q(D) = [aggtuple_Q(e) \mid e \in \mathcal{E}_Q]$$

where

$$aggtuple_Q(e) = tuple_Q(\imath) \bigotimes aggregate_Q(e)$$

and

$$aggregate_Q(e) = \bigotimes_{\text{“}agg(t.B)\text{”} \in \texttt{AggList}} agg([\jmath(t)[B] \mid e \in \mathcal{E}_Q \text{ and } \jmath \in e])$$

This completes the proof of Theorem 5.1. ■

5.2 Processing *SchemaSQL* Queries with Dynamic Output Schema

Our algorithms for the processing of *SchemaSQL* queries with dynamic output schema are directly based on the *allocate* and *merge* operations discussed in Section 4.2. Recall that the output schema is dynamic, *i.e.* it depends on the input data. This happens when variables appear in the `create view` statement.

Consider the *SchemaSQL* view definition

```
create view  db::rel(attr1, ..., attrn) as
select       itemList, aggList
from         fromList
where        whereConditions
group by     groupbyList
having       havingConditions
```

If `db`, `rel` and `attr1`, ..., `attrn` are all constants, then the query has a fixed output schema and will be processed according to Algorithm 5.1. In the following, we assume the `create view` clause has at least one variable. Also, recall that at most one of `attr1`, ..., `attrn` can be a variable, while both or either of `db` and `rel` can be variable.

Algorithm 5.2 *SchemaSQL Query Processing – Dynamic Output Schema*
Input: A *SchemaSQL* query with aggregation and with a dynamic output schema
Output: The result of the query.
Method: First, we compute the JVIT as for fixed output schema. Then for each database d and for each relation r for which an output view must be created, we implement the allocate and merge operations discussed in Section 4.1. These operations are implemented in cascade, without materializing the result of the allocate.

1. Compute the VITs and the JVIT exactly as in Algorithm 5.1.

2. If `db` and `rel` in the `create view` clause are both constants, then create a single relation (view) `rel` in the database `db`. Otherwise, query the JVIT to obtain the values of `db` and/or `rel`, to form distinct (d, r) pairs dictated by the create view statement. For example, if both `db` and `rel` are variables, compute the (d, r) pairs using the following SQL query: :

   ```
   select    distinct db as d, rel as r
   from      JVIT
   ```

3. Assume $\texttt{attr}_i$ in the `create view` clause is a variable. For each (d, r) pair of the previous step, obtain the instantiations for the $\texttt{attr}_i$ by applying the following SQL query on JVIT:

   ```
   select    distinct attr_i
   from      JVIT
   where     db = d and rel = r
   ```

   Note that one query is needed for each (d, r) pair, and one relation is constructed (as the output of the *SchemaSQL* query) for each (d, r) pair, as described in next steps.

4. Let (d, r) be an arbitrary, but fixed pair, from above. Let $a_1, \ldots, a_m$ be the output of the previous step for this pair (d, r). The schema of the relation corresponding to this (d, r) pair in the result of the *SchemaSQL* query is $\{\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_1, \ldots, a_m, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n\}$. For each a_k, $k = 1, \ldots, m$, we generate a relation $R_k(\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_k, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n)$ as follows.

   To simplify the presentation, we expand the `select` clause of the *SchemaSQL* query, namely `select itemList, aggList`, as `select` $\texttt{obj}_1, \ldots, \texttt{obj}_n$. Note that each of $\texttt{obj}_1, \ldots, \texttt{obj}_n$ can be a simple or an aggregate operation. Compute relation $R_k(\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_k, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n)$ using the following SQL query on JVIT.

```
select    obj_1 as attr_1, ..., obj_{i-1} as attr_{i-1}, obj_i as a_k,
          obj_{i+1} as attr_{i+1}, ..., obj_n as attr_n
from      JVIT
where     db = d and rel = r and attr_i = a_k
group by  groupbyList
having    havingConditions
```

We observe here that the outer union of $R_1, \ldots, R_m$ computes the result of the allocate operation of Sections 4.1 and 4.2. More precisely, the outer union is equal to $allocate_Q(\imath)$ (for *SchemaSQL* queries with no aggregation), or $aggallocate_Q\ (\imath)$ (for *SchemaSQL* queries containing aggregation), as discussed in Sections 4.1 and 4.2. However, we do *not* compute this outer union, and instead compute the result of the cascade of allocate followed by merge directly, in the next step.

5. Perform an outer join of the relations R_k, $1 \leq k \leq m$, of the previous step using the following SQL query:

```
select    R_1.obj_1 as attr_1, ..., R_1.obj_{i-1} as attr_{i-1},
          R_1.a_1 as a_1, R_2.a_2 as a_2, ... R_m.a_m as a_m,
          R_1.obj_{i+1} as attr_{i+1}, ..., R_1.obj_n as attr_n
from      R_1 outer join R_2 outer join ... outer join R_m
where     R_1.obj[1..i-1] = R_2.obj[1..i-1] and ... and
          R_1.obj[1..i-1] = R_m.obj[1..i-1] and
          R_1.obj[i+1..n] = R_2.obj[i+1..n] and ... and
          R_1.obj[i+1..n] = R_m.obj[i+1..n] and
          R_1.sid = R_2.sid and ... and R_1.sid = R_m.sid
```

where $R_1.\mathtt{obj}[j..k] = R_2.\mathtt{obj}[j..k]$ abbreviates $R_1.\mathtt{obj}_j = R_2.\mathtt{obj}_j$ and $\cdots$ and $R_1.\mathtt{obj}_k = R_2.\mathtt{obj}_k$. We have assumed that the relations R_k, $k = 1, \ldots, m$, have a special attribute `sid` (sequence id), that records a sequential row number within each group of tuples with the same values for $\mathtt{obj}_1, \ldots, \mathtt{obj}_{i-1}, \mathtt{obj}_{i+1}, \ldots, \mathtt{obj}_n$. Generating these sequence id values using SQL applications or user defined functions is quite simple in most commercial database systems as explained below.

Our technique for sequence id generation assumes the availability of a system supplied, user query-able unique row identifier column such as the *rid* column of ORACLE or the *identity* column of SYBASE SQL SERVER. Now, consider a tuple t in a relation R_k whose sequence id (within the group to which it belongs) needs to be generated. The following SQL statement returns the *sid* for t. We use the Oracle convention to refer to the unique row identifier column as *rid*.

```
select    count(*)
from      R_k
where     R_k.obj[1..i − 1] = t.obj[1..i − 1] and
          R_k.obj[1 + 1..n] = t.obj[i + 1..n] and
          R_k.rid < t.rid
```

In the above statement, *sid* for tuple t is obtained by counting the number of tuples in t's group that have a lesser rid value than the rid value of t.

Note that the join condition involving the sid column in the above outer join SQL query correctly captures the merge semantics of Section 4.2.

This completes Algorithm 5.2.

Example 5.3 *[Processing a* SchemaSQL *query with aggregation and dynamic output schema]* Consider query `Q8` of Example 4.3, repeated here for convenience.

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

The first step of the algorithm is the generation of $VIT_{\texttt{C}}$ (schema: {`C`}), $VIT_{\texttt{T}}$ (schema: {`C, TC, Tdept`}), and $VIT_{\texttt{U}}$ (schema: {`Udname, Ufname`}), and then the generation of JVIT. Since the details of the generation of these tables are identical to that for fixed output schema, we will suppress these details and assume that the JVIT for this query, with schema (`C, TC, Tdept, Ufname, Udname`) is created.

Since the `db` and `rel` components of the `create view` clause of the *SchemaSQL* query are constant, we proceed to the third step. The following query is used to enumerate the distinct values of the variable `C` appearing in the `create view` statement.

```
select    distinct C
from      JVIT
```

We obtain (Prof, AssocProf, Technician) as the answer to the query above. (We are using the federation of Figure 2).

Step 4 of the algorithm involves the creation of relations R_k, one per each value obtained in step 3. For this example there are three relations: Relation R_1 has the schema {faculty, Prof} and is obtained as:

```
select    Ufname as faculty, avg(TC) as Prof
from      JVIT
where     C = "Prof"
group by  Ufname
```

Relations R_2 and R_3 are similar, with Prof substituted by AssocProf and Technician, respectively. Note that the schemas of these three relations are {faculty, Prof} , {faculty, AssocProf} , and {faculty, Technician} , respectively.

The final step of the algorithm performs an outer join of the three relations generated in step 4:

```
select    R1.faculty as faculty, R1.Prof as Prof,
          R2.AssocProf as AssocProf, R3.Technician as Technician
from      R1 outer join R2 outer join R3
where     R1.faculty= R2.faculty and R1.faculty= R3.faculty and
          R1.sid = R2.sid and R1.sid = R3.sid
```

■

Theorem 5.2 *[Correctness for dynamic output schema]* Algorithm 5.2 correctly computes answers to *SchemaSQL* queries with dynamic output schemas.

Proof: We proceed in two steps: First we show the schema of the output generated by Algorithm 5.2 is correct, then we show the contents generated by the algorithm is also correct. This parallels the semantics definitions of Sections 4.1 and 4.2. Recall that by Lemma 5.1, JVIT contains the set of valid instantiations $\mathcal{I}_Q$, restricted to the columns of JVIT.

(i) *Schema of the output*: The semantics of dynamic output schema was discussed in Section 4.1 with the aid of the equivalence relation $\equiv$ (Definition 4.1). Each equivalence class of $\equiv$ corresponds to one value of the (db, rel) pair in the create view statement. The output schema corresponding to a (d, r) pair is simply the set of values in the corresponding equivalence class of $\equiv$ for the attributes in the create view statement. It is evident in Steps 4 and 5 of the algorithm that the schema of the output for a (d, r) pair, namely, $\{\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_1, \ldots, a_m, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n\}$, indeed corresponds to this semantics.

(ii) *Contents of the output*: We concentrate on queries with aggregation and dynamic output schema. Queries with no aggregation are simpler and can be regarded as a special case of the former. The semantics of aggregation with dynamic view definition was discussed in Section 4.2 with the aid of the equivalence relation # (Definition 4.6). Each equivalence class of # corresponds to the set of instantiations with the same values for *SchemaSQL* variables appearing in the `create view` statement, *plus* the objects appearing in the `group by` list. The variables in the `create view` statement can include `db`, `rel`, and one attribute such as $\texttt{attr}_i$.

In Step 4 of the algorithm, each relation R_k, $k = 1, \ldots, m$, corresponds to one value a_k of $\texttt{attr}_i$, and is for a given (d, r) pair. This is evident by examining the `where` clause of the (regular) SQL query that generates R_k. In fact, since this SQL query also incorporates the `group by` and `having` clauses from the original *SchemaSQL* query, we can see that each R_k "packs" the output corresponding to equivalence classes of # that have the same values d, r, and a_k, for `db`, `rel`, and $\texttt{attr}_i$, respectively. Also, notice that the schema of R_k is $\{\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_k, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n\}$. We observe that if an outer union is performed on $R_1, \ldots, R_m$, tuples of each relation R_k is extended by nulls to the full schema $\{\texttt{attr}_1, \ldots, \texttt{attr}_{i-1}, a_1, \ldots, a_m, \texttt{attr}_{i+1}, \ldots, \texttt{attr}_n\}$, and unioned. This coincides exactly with the semantics of the allocate operation.

Finally, we need to show that Step 5 of the algorithm generates the merge (Definition 4.4) of the allocated relation. Note that our algorithm does not materialize the allocated relation. Rather, Step 5 performs an outer join of the relations R_k, $k = 1, \ldots, m$. We observe that a set of tuples $\{t_1, \ldots, t_m\}$, $t_j \in R_j$, are mergeable (Definition 4.2) if and only if, for all $j, k \in \{1, \ldots, m\}$, $t_j(\texttt{attr}_1) = t_k(\texttt{attr}_1), \ldots, t_j(\texttt{attr}_{i-1}) = t_k(\texttt{attr}_{i-1}), t_j(\texttt{attr}_{i+1}) = t_k(\texttt{attr}_{i+1}), \ldots, t_j(\texttt{attr}_n) = t_k(\texttt{attr}_n)$. The SQL query in Step 5 achieves the merging of these tuples. Further, the conditions in the `where` clause of this query that involve the `sid` attribute make sure each tuple from a relation R_k participates in the formation of exactly one output tuple. Without these conditions each tuple could possibly participate in the formation of multiple output tuples violating the multi-set semantics of *SchemaSQL*.

This completes the proof of Theorem 5.2. ■

Based on the algorithms presented in this section, prototype implementations of *SchemaSQL* have been developed [GLS+97, SW97].

6 Discussion

In this section, we discuss a variety of optimization opportunities in the context of *SchemaSQL* implementation. We also discuss some novel database applications facilitated by a *SchemaSQL*-based system.

6.1 Query Optimization Opportunities

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 used 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(b) 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 fewer 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 its VIT 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.

6.2 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 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.

Example 6.1 *[Semantic heterogeneity]* 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[6]:

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

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.

[6]The syntax of the type specification rules is based on the syntax of *SchemaLog* [LSS97].

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

create view  univInfoB::timestats(prov, C) as
select       P, T.P
from         univInfoA::stats T,
             T.cat C,
             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.

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 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]), Noodle (Mumick and Ross [MR93]), 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 meta-data, 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 3.3 and 4.2 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 3.1, 4.3). Interestingly, 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. Substantial amount of work has been done on efficient computation of the data cube. This is a large body of work and we refer the reader to the pioneering work by Agarwal et al. [AAD+96] for details. It would be interesting to investigate how some of these fast algorithms for the cube can be leveraged for efficient computation of *SchemaSQL* queries involving complex aggregations.

As pointed out in Section 7, the aggregation and restructuring features of *SchemaSQL* would be very useful in OLAP style computations. However, *SchemaSQL* by itself is not a full-fledged OLAP query language. Gingras [Gin98] and Gingras and Lakshmanan [GL98] document the limitations of *SchemaSQL* for OLAP applications and propose a language they call *nD-SQL* specifically designed for OLAP computations. An example of such a limitation is that in *SchemaSQL*, one cannot create a column as a function of more than one domain value. *SchemaSQL* was designed with interoperability in mind and there seems to be no natural need for this capability.

Statistical Databases (SDB) applications, in addition to requiring complex statistical operations, also have natural use for restructuring and for handling data meta-data conflicts [Sho97]. Languages developed in the SDB community do possess such features (*e.g.* see [MRS92, OMO89, OOM85]). However, on comparing SDB languages with *SchemaSQL*, we find that they are complementary with respect to what they have to offer. It might be interesting to investigate how useful features from these languages can be combined to support sophisicated SDB and OLAP applications.

While we propose *SchemaSQL* as a language for multi-database interoperability and data warehousing applications, researchers have identified several applications where *SchemaSQL* can enhance the functionality of a *single* DBMS significantly. These applications include *publishing of relational data on the web* [MTW97, Mil98, KZ95], *techniques for providing physical data independence* [Mil98], *developing tightly coupled scalable classification algorithms in data mining* [WIV98], and *query optimization in the context of data warehousing* [SV98]. Motivated by these observations, in [LSS99], we discuss efficient implementation of *SchemaSQL* on a *single* RDBMS. We develop logical as well as physical algebraic operators for *SchemaSQL* and using these operators as a vehicle, present several alternate implementation strategies for *SchemaSQL* queries/views on a single database setting. We also test the effectiveness of these strategies by means of a series of tests based on TPC-D benchmark data.

In other related work, Gyssens et al. ([GLS96]) develop a general data model called the *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 the restructuring aspects of OLAP. Restructuring views expressible in *SchemaSQL* can also be expressed in their algebra but they do not address aggregate computations.

In [LSS97, 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 [LSS97]. 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. 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. 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.

8 Conclusions

We introduced *SchemaSQL*, a principled extension of SQL for relational multi-database systems. In *SchemaSQL* data and meta-data, *i.e.* database instance and its schema, are treated uniformly, thus making it possible to query both the contents and the structure of a database. In a multi-database environment, *SchemaSQL* provides the means for handling schematic (structural) heterogeneity, *i.e.* similar information represented in different structures. View definition in *SchemaSQL* makes it possible to define *restructuring* views, namely, views that can change the structure of input data in a manner that exploits the data/meta-data interplay, while preserving the information content. Data (*i.e.* attribute values) in one representation can play the role of meta-data objects (database name, relation name, and attribute name) in a restructured view. *SchemaSQL* also provides novel aggregation capabilities, akin to some aggregations only available in OLAP systems. In addition to the usual SQL column aggregations, in *SchemaSQL* it is possible to aggregate on a set of columns, individually or collectively, whether the columns are in one relation or in several. This can be done using a *single query*. The latter case is basically aggregation over a "block" of data. The set of columns participating in the aggregation is determined dynamically, at execution time, and is dependent on the database instance. Horizontal (*i.e.* row) aggregation is also possible in *SchemaSQL*.

SchemaSQL was introduced specifically in the context of multi-database interoperability, but it has since found significant usefulness in the context of *single database applications*. These applications include database publishing on the web, query optimization in a data warehouse, and scalable classification algorithms in data mining, and were briefly reviewed in Section 7.

In this paper we presented the syntax of *SchemaSQL*, developed a formal semantics for it, and discussed an architecture and algorithms for the implementation of *SchemaSQL* using available (SQL) database technology. We also proved the correctness of the algorithms. Many interesting research directions are worth pursuing, and we mention just a few. Given the single database applications, can we identify useful primitive plan operators that can be pushed into a SQL engine to facilitate efficient computation of *SchemaSQL* queries? How can we extend the *SchemaSQL* system to handle semantic heterogeneity in a manner that is transparent to the user? Can we use the rich expressive power of the language to solve more general data mining problems? In ongoing work, we are studying some of these questions.

References

[AAD+96] Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J.F., Ramakrishnan, R., and Sarawagi, S. On the Computation of Multidimensional Aggregates. In *Proc. 22nd VLDB Conf.*, pages 506–521, 1996.

[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.

[DJ] IBM Corp. DB2 DataJoiner. http://www.software.ibm.com/data/datajoiner

[ERS98] Elmagarmid, A., Rusinkiewicz, M., and Sheth, A., editors. *Management of Heterogeneous and Autonomous Database Systems.* Morgan Kaufmann, 1998.

[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.

[Gin98] Gingras, F. Extending SchemaSQL Towards Multidimensional Databases and OLAP. Master's thesis, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1998.

[GL98] Gingras F. and Lakshmanan L.V.S. nD-SQL: A multi-dimensional language for interoperability and olap. In *Proc. of the Conf on Very Large Databases (VLDB)*, 1998.

[GLRS93] Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. Query languages for relational multidatabases. *VLDB Journal*, 2(2):153–171, 1993.

[GLS+97] Gingras, F., Lakshmanan, L.V.S., Subramanian, I.N., Papoulis, D., and Shiri, N. Languages for Multi-database Interoperability. In *Proc. ACM SIGMOD*, pages 536–538, 1997.

[GLS96] Gyssens, M., 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, M., Kim, W., and Sagiv, Y. 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.

[KZ95] Krishnamurthy, R., and Zloof, M. RBE: Rendering By Example In *Proc. Intl Conference on Data Engineering (ICDE)*, 1995.

[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.

[LMR90] Litwin, W., Mark, L., and Roussopoulos, N. Interoperability of multiple autonomous databases. *ACM computing surveys*, 22(3):267–293, Sept 1990.

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

[LNS90] Lipton, R., Naughton, J., and Schneider, D. 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.

[LSS96] Lakshmanan, L.V.S., Sadri, F., and Subramanian, S. N. SchemaSQL – a language for interoperability in relational multidatabase systems. In *Proc. International Conference on Very Large Databases (VLDB 96)*. September 1996.

[LSS97] Lakshmanan, L.V.S., Sadri, F., and Subramanian, I. N. Logic and algebraic languages for interoperability in multidatabase systems. *Journal of Logic Programming*, 33:2, November 1997, pp. 101–149.

[LSS99] Lakshmanan, L.V.S., Sadri, F., and Subramanian, S. N. On efficiently implementing SchemaSQL on a SQL database system. In *Proc. International Conference on Very Large Databases (VLDB 99)*. September 1999.

[Mil98] Miller R.J. Using Schematically Heterogeneous Structures. *Proc. ACM SIGMOD*, 189–200, Seattle, WA, May 1998.

[MR93] Mumick, I.S. and Ross, K.A. Noodle: A language for declarative querying in object-oriented database. In *Proc. 3rd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD '93)*, December 1993.

[MR95] Missier, P. and Rusinkiewicz, M. 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.

[MRS92] Meo-Evoli, L., Ricci, F.L. and Shoshani, A. On the semantic completeness of macro-data operators for statistical aggregation. In *Proc. SSDBM*, pages 239–258, 1992.

[MTW97] Miller R.J., Tsatalos O.G., and Williams, J.H. DataWeb: Customizable Database Publishing for the Web. *IEEE MultiMedia* 4(4): 14-21 (1997)

[OMO89] Ozsoyoglu, G., Matos, V., and Ozsoyoglu, Z.M. Query processing techniques in the summary-table-by-example query language. *ACM TODS*, 14(4):526–573, 1989.

[OOM85] Ozsoyoglu, G., Ozsoyoglu, Z.M., and Mata, F. A language and a physical organization technique for summary tables. In *Proc. ACM SIGMOD*, pages 3–16, 1985.

[Ros92] Ross, K. 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.

[Sho97] Shoshani, A. Olap and statistical databases: Similarities and differences. In *Proc. ACM PODS*, pages 185–196, 1997.

[SL90] Sheth, A. and Larson, J.A. Federated database system for managing distributed, heterogeneous and autonomous databases. *ACM computing surveys*, 22(3):183–236, Sept. 1990.

[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.

[SV98] Subramanian, I.N. and Venkataraman, S. Query Optimization Using Restructuring-Views. IBM Internal Report, 1998.

[SW97] Sadri, F. and Wilson, S.B. Implementation of SchemaSQL – a language for relational multi-database systems. Technical Report, Department of Mathematical Sciences, University of North Carolina at Greensboro, February 1997.

[WIV98] Wang, M., Iyer, B., and Vitter, J. Scalable Mining for Classification Rules in Relational Databases. *Int. Database Engg and Applications Symposium (IDEAS'98),* Cardiff, Wales, U.K., July 1998.