On Efficiently Implementing SchemaSQL on a SQL Database System

Laks V. S. Lakshmanan*
IIT – Bombay
laks@cse.iitb.ernet.in

Fereidoon Sadri
UNCG
sadri@uncg.edu

Subbu N. Subramanian
IBM Almaden
subbu@almaden.ibm.com

Abstract

SchemaSQL is a recently proposed extension to SQL for enabling multi-database interoperability. Several recently identified applications for SchemaSQL, however, mainly rely on its ability to treat data and schema labels in a uniform manner, and call for an efficient implementation of it on a *single RDBMS*. We first develop a logical algebra for SchemaSQL by combining classical relational algebra with four restructuring operators – *unfold, fold, split*, and *unite* – originally introduced in the context of the tabular data model by Gyssens et al. [GLS96], and suitably adapted to fit the needs of SchemaSQL. We give an algorithm for translating SchemaSQL queries/views involving restructuring, into the logical algebra above. We also provide physical algebraic operators which are useful for query optimization. Using the various operators as a vehicle, we give several alternate implementation strategies for SchemaSQL queries/views. All the proposed strategies can be implemented *non-intrusively* on top of existing relational DBMS, in that they do not require any additions to the existing set of plan operators.

We conducted a series of performance experiments based on TPC-D benchmark data, using the IBM DB2 DBMS running on Windows/NT. In addition to showing the relative tradeoffs between various alternate strategies, our experiments show the feasibility of implementing SchemaSQL on top of traditional RDBMS in a non-intrusive manner. Furthermore, they also suggest new plan operators which might profitably be added to the existing set available to relational query optimizers, to further boost their performance.

*Currently on leave from Concordia Univ., Montreal, Canada. Work performed by this author at IIT–Bombay.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Data warehousing is a technology motivated by decision support applications, which require consolidated high-level information on the OLTP data for the purpose of organizational decision making. A major challenge for this is the integration of data sources that are schematically disparate, in that data values in one source may be modeled as schema (attribute or relation) labels in another. Real examples of such disparity abound (e.g., see [KLK91, LSS97]). As an example, in Figure 1(a), a snap shot of a stock broker's database, the relation names in the ABC brokerage database are stock (ticker) names that could indeed be query-able information. We will use this as a running example in the paper. In an attempt to address this challenge of schematic disparity, researchers have proposed higher-order extensions to relational calculus and SQL that facilitate developing and deploying data warehousing/data integration applications [KLK91, CKW93, Ros92]. SchemaSQL [LSS96] is one such extension to SQL that allows for uniform manipulation of data and schema, and supports: (a) interoperability in a mutlidatabase system and (b) easy specification of various complex queries and computations arising in data warehousing applications.

While SchemaSQL was originally proposed as a language for multi-database interoperability and data warehousing applications, since its proposal, researchers have identified several applications where it 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 *source query optimization in the context of data warehousing*

ABC Brokerage

ibm

date	xge	open	close	low	high
10/01	NYSE	91.43	92.36	90.23	92.38
10/01	TSE	90.06	92.34	90.06	92.78
10/02	LSE	91.78	93.65	91.56	93.86

msft

date	xge	open	close	low	high
10/01	NYSE	86.31	86.79	85.97	86.92
10/01	TSE	85.27	85.20	85.12	85.86
10/02	LSE	86.90	87.35	86.90	87.74

...
...

stock

ticker	busType
ibm	tech
msft	tech
xon	oil
....	...

(a)

isa(ibm, ticker) ←
isa(msft, ticker) ←
isa(hp, ticker) ←
...
isa(open, priceType) ←
isa(close, priceType) ←
...
isa(T.xge, xge) ← *relVar*(S) & *isa*(S, ticker) & *tupleVar*(T, S)
isa(T.date, date) ← *relVar*(S) & *isa*(S, ticker) & *tupleVar*(T, S)
isa(T.PT, price) ← *relVar*(S) & *isa*(S, ticker) & *colVar*(PT, S) & *tupleVar*(T, S) *isa*(PT, priceType)
isa(T.ticker, ticker) ← *tupleVar*(T, stock)
isa(T.busType, busType) ← *tupleVar*(T, stock)

(b)

```
CREATE  VIEW    X(date, priceType, S) AS
        SELECT  T.date, PT, T.PT
        FROM    -> S, S T, S-> PT, T.xge X, stock U
        WHERE   PT <> date AND PT <> xge AND
                S <> stock AND S = U.ticker AND
                U.busType = 'tech'
```

(c)

XgeView

NYSE

date	priceType	ibm	msft	...
10/01	open	91.43	86.31	...
10/01	close	92.36	86.79	...
10/02	low	90.23	85.97	...
...	...	...	...	...

TSE

date	priceType	ibm	msft	...
10/01	open	90.06	85.27	...
10/01	low	90.06	85.12	...
10/02	high	92.78	85.86	...
...	...	...	...	...

...
...

(d)

Figure 1: *Example Stock Market Database: (a) ABC brokerage database; (b) Meta-data specs of schema in (a); (c) An example restructuring* SchemaSQL *view against database (a); (d) Materialization of (c).*

[SV98]. Thus there is a clear motivation for realizing an efficient implementation of SchemaSQL on even a *single database system*, and this is the subject of this paper.

Section 1.1 reviews SchemaSQL by examples, while Section 1.2 elaborates on the applications mentioned above, in order to bring out the motivation for our work. [LSS96, Sub97] contain an elaborate exposition of SchemaSQL.

1.1 SchemaSQL by Examples

SchemaSQL extends SQL in the following ways.

(1) While SQL allows only variables ranging over tuples of a fixed relation, SchemaSQL allows variables ranging over relation labels, column labels, tuples in (several) relations, and domain values appearing in relation columns.[1]. E.g., in the query part of the SchemaSQL view definition of Figure 1(c), the declaration '-> S' in the FROM clause (together with the constraint 'S <> stock' in the WHERE clause) says S is a relation label variable that ranges over the relation labels (i.e.the ticker names) ibm, msft, ... in the ABC brokerage; 'S T' says T is a tuple variable that ranges over the tuples in each of the ticker relations; 'S -> PT' says PT is a column label variable that ranges over the column labels of the ticker relations, and 'T.xge X' says X is a domain variable that ranges over the values in the xge column of the ticker relations.

[1] SchemaSQL, as originally proposed for a multi-database system [LSS96], also permits database label variables. Given the single-database focus of this paper, we drop database label variables here.

(2) SQL allows only one type of domain expression, namely tupleVar.attr. In contrast, SchemaSQL allows domain expressions of the form tupleVar.col (e.g., U.ticker and T.PT in Figure 1(c)), relVar (e.g., S), colVar (e.g., PT), and domVar (e.g., X). Note that attr denotes a fixed attribute name, while col may be an attribute name or a column label variable.

(3) The syntax of SchemaSQL permits restructuring views which involve a clear interplay between schema (i.e. relation/column) labels and data. For example, the SchemaSQL view definition in Example 1(c) maps the ABC brokerage database to the view XgeView of Figure 1(d). Corresponding to each exchange 'x' in the ABC brokerage, XgeView has a relation labeled 'x'. This is achieved by using the domain variable X in the relation label position in the CREATE VIEW statement. Similarly using the variable S in the column label position generates a column labeled 't' in XgeView, for each distinct ticker 't' in the input database ABC.

(4) Aggregations more general than the conventional vertical aggregation of SQL can be readily expressed in SchemaSQL: examples include horizontal and rectangular block aggregations [LSS96].

1.2 SchemaSQL in the context of a single database system

Several researchers including Miller [Mil98], Wang et al. [WIV98], and Subramanian and Venkataraman [SV98] have observed the useful functionality that SchemaSQL can bring to a *single* DBMS. Below, we summarize these "killer-apps" that form the basis of their observations.

Database Publishing on the Web: Making information in relational databases Web-available is an important problem in the context of databases and the internet. In [Mil98], Miller makes a compelling argument for the role of SchemaSQL in this context: A web user cannot be expected to know/learn the schema of a data source, which might be quite complex. This necessitates support for 'schema independent querying'. E.g., consider a keyword search interface that permits users to find stocks (tickers) in the ABC brokerage whose price is more than a certain amount on any date. The user may not specify or even care about the price type nor know that tickers appear as relation labels in the database. Under these assumptions, this is a higher-order query on the ABC database. The ability of SchemaSQL to quantify over schema labels can be used to support such schema independent querying required in database publishing environments.

Techniques for Physical Data Independence: An important technique used in indexing architectures for integrating new indexes into a query optimizer is the usage of views for describing the indexes [CKPS95, TSY96]. Conventional techniques (such as [TSY96]) are restricted in that they can only describe indexes that conform to the class of select-project-join views [Mil98]. Miller [Mil98] gives examples of B^+-tree indexes for all subclasses of a class, that can be expressed in SchemaSQL but not in SQL. Besides, she also shows that the optimization of important classes of queries including the *data fusion* queries [YPAG98], ubiquitous on internet databases, can benefit from higher-order views definable *only* in languages like SchemaSQL.

Scalable Classification Algorithms in Data Mining: Classification is a fundamental operation in data mining. The literature abounds with stand-alone algorithms for doing classification on data stored in files [AIS93]. In [WIV98], Wang et al. make a strong case for a tightly coupled implementation of classification algorithms, well integrated with SQL, and give a scalable such algorithm whose operations are expressed using SQL queries. Despite the advantages of tight coupling, they point out that the SQL queries produced by the algorithm are numerous, complex, verbose, and hard to optimize. They also provide a SchemaSQL-based algorithm and show that the resulting queries (and the algorithm) are concise, far fewer, and are readily optimizable w.r.t. database scans. In order to achieve this, they exploit SchemaSQL's ability to quantify over schema labels.

Query Optimization in a Data Warehouse: Optimization using materialized views is a popular and useful technique in the context of traditional database query optimization [BLT86, GMS93, CKPS95, LMSS95, SDJL96] which has been successfully applied for optimizing data warehouse queries [GHQ95, HGW$^+$95, HRU96, GM96, GHRU97]. However, the materialized views considered by *all* of the above works are traditional views expressed in SQL. In [SV98], Subramanian and Venkataraman establish the surprising result that the use of materialized SchemaSQL views involving data/schema restructuring can have an order of magnitude improvement in the execution times of even *traditional* SQL queries. Thus, it is worth expanding the class of views that are candidates for materialization, beyond the class of queries being optimized, in this case from those expressible in SQL to those expressible in SchemaSQL.

The preceding applications demonstrate the need for efficient implementation of SchemaSQL even in the context of an *individual database system*. Given the popularity and extensive use of SQL systems, implementations of SchemaSQL that are *non-intrusive* are particularly attractive: non-intrusive means that the implementation should not require modifications to the SQL engine, in particular, to the set of plan operators used by existing query optimizers.

Contributions: Efficient implementation of SchemaSQL on top of SQL systems with minimal intrusion is the main objective of this paper. To this end, we develop a logical algebra consisting of classical relational algebra together with four restructuring operators – unfold, fold, split, and unite – originally introduced in [GLS96] and simplified and adapted to SchemaSQL setting (Section 3.1). We give an algorithm that can detect whether schema label variables in a given SchemaSQL query are properly constrained and then translate it to an equivalent expression in the logical algebra (Section 3.2). A key idea used by our algorithm is the notion of *meta-data specification*, which can be viewed as an augmentation to system catalog tables, normally maintained in RDBMS. We also provide a physical algebra (Section 4). Using that as a vehicle, we develop several implementation strategies (Section 5). Essentially, the optimizer can choose either to use direct strategies for the logical operators or reduce them to physical operators and use strategies for the latter. We illustrate logical and physical query optimization issues arising with SchemaSQL queries (Section 5.1). None of our strategies requires additions/modifications to the existing set of plan operators used by today's RDBMS. To test the feasibility of our strategies and their relative performance, we conducted a series of experiments based on TPC-D benchmark data, the results of which we report (Section 6). We conclude the paper by summarizing the main results and discussing future work (Section 7). *For lack of space, we only consider single block* SchemaSQL *queries without aggregation in this paper. Details on processing arbitrary* SchemaSQL *queries as well as proofs of various results we present can be found in the full paper [LSS99].* We use the terms queries and views interchangeably for convenience, without causing confusion.

2 Related Work

Several languages have been proposed to deal with schematic disparity in multi-databases. These include (higher-order) logics [KLK91, LSS97, SAB+95], algebras [Ros92, GLS96], and SQL extensions [KKS92, GLRS93, LSS96, GL98]. The work in [LSS96, GL98] is particularly relevant to this paper. Of these, [LSS96], the paper which introduced SchemaSQL, mainly concerns itself with deploying it in the context of multi-database systems and related query processing issues. The approach to query processing in [LSS96] is based on compiling federation queries expressed in SchemaSQL to SQL queries which are then dispatched to component DBMS, answers eventually collected at the server and restructured if necessary before being presented to the user. To a large extent nd-SQL, an offshoot of the SchemaSQL project, has a similar focus, except that it can express a substantially larger class of OLAP queries than SchemaSQL. None of these papers has considered the implementation of SchemaSQL or a variant, on a *single* DBMS.

Miller [Mil98] made powerful observations concerning the value that SchemaSQL can bring to a single DBMS (see Section 1.2). This, together with the SQL/SchemaSQL-based classification algorithm developed by Wang et al. [WIV98], and the work of Subramanian and Venkataraman on query optimization in data warehousing applications [SV98] form a core motivation for our work. Miller calls a schema *first-order* with respect to a set of queries $\mathcal{Q}$, provided all queries in $\mathcal{Q}$ can be expressed in SQL. Intuitively, this means all information of interest (to the queries in $\mathcal{Q}$) is modeled as data. Our notion of "flat schema" (Section 3.1) is similar to that of first-order schema in [Mil98], but a major difference is flatness is an absolute notion, in that it does not depend on the query class being considered. Besides, flat schemas play merely a technical role in query processing in that conceptually we can regard as though all source data is flattened so as to conform to a flat schema before being manipulated further. Miller restricts the integration schema in a federation of databases, or a restructuring schema in a single database, to be first-order. In addition, all sources should be expressed as views of the integration/restructuring schema in the fragment of SchemaSQL that does not use relation label or column label variables. For this setting, she addressed the problem of determining whether an SQL query on the integration/restructuring schema is answerable, as an SQL or SchemaSQL query, over the sources, and provides algorithms to translate the given federation query to a query over the sources.

The main focus of our work is implementing SchemaSQL on a single relational DBMS. In this context, we show that the full-fledged language of SchemaSQL can be implemented as SQL applications with no modification to SQL engines. We also study the issue of optimizing SchemaSQL queries non-intrusively, as well as the feasibility of highly efficient (intrusive) implementation of SchemaSQL.

In [SV98] Subramanian and Venkataraman study the utility of materialized SchemaSQL views for improving the performance of even traditional SQL queries. In this context, they propose algorithms that given an SQL query against the base tables and a set of SchemaSQL views that define the materialized restructured tables, generates alternative queries that exploit the SchemaSQL views. They also propose techniques for incorporating these alternative choices for cost based query optimization. The contributions in [SV98] are complementary to our work in this paper.

3 A Logical Algebra for SchemaSQL

Our approach to implementing SchemaSQL on top of a relational DBMS is based on first transforming SchemaSQL queries into equivalent queries in an extended relational algebra, which are then translated into a physical algebra, consisting of the standard suite of physical operators for relational DBMS, together with new physical operators introduced in Section 4. The logical algebra is the topic of this section. It consists of classical relational algebra augmented with four restructuring operators – *unfold, fold, split,* and *unite*, originally introduced in the context of the tabular algebra, by Gyssens et al. [GLS96]. Our aim in this paper is to develop *practical* and *efficient* strategies for implementing SchemaSQL. With this in mind, we adapt the definitions of the restructuring operators above to our context.

3.1 Restructuring Operators Simplified

We assume infinite pairwise disjoint sets of names $\mathcal{N}$ and values $\mathcal{V}$. We assume that $dom : \mathcal{N} \to 2^{\mathcal{V}}$ is a partial function such that whenever $dom(N)$ is defined, it associates name N with a non-empty set of values $dom(N) \subset \mathcal{V}$. The following basic notion is needed in the rest of the paper.

Definition 1 (Flat Scheme) A relation scheme $R(A_1, \ldots, A_n)$ is said to be *flat* provided all the entries $R, A_1, \ldots, A_n$ are names. A database scheme is flat if all relation schemes in it are.

The relational model implicitly assumes that schemes are flat. The intuition is that normally one assumes all information of interest is in the values (data). By requiring that no values appear in schema labels, most if not all useful queries against a flat database scheme can be expressed in a first-order query language like SQL. E.g., the relation scheme stock(ticker, busType) in Figure 1(a) is flat, assuming all the labels stock, ticker, busType are names. However, as the same figure illustrates (e.g., see relations ibm(date, xge, open, close, low, high), msft(...), ...), databases corresponding to non-flat schemes *do* get implemented using standard RDBMS.

We next give the intuition behind the restructuring operators.[2] The unfold operator takes a relation over a set of names as input and produces a cross-tab like output which is equivalent in information content to the input, as shown in Figure 2: (a) & (b). The fold operator does the converse. Similarly, the split operator takes a relation as input and produces a set of relations, one for each distinct value of a specified attribute, as output. The labels of the latter relations are set to the said attribute values. This is illustrated in Figure 2: (a) & (c). Finally, unite is the converse of split.

stock	xge	price
att	nyse	91.56
lucent	nyse	89.45
att	tse	92.35
lucent	tse	87.45

(a) Input table `close`

stock	nyse	tse
att	91.56	92.35
lucent	89.45	87.45

(b) Table `uf-close` result of UNFOLD$_{\text{by } \texttt{xge} \text{ on } \texttt{price}}$(`close`)

stock	price
att	91.56
lucent	89.45

(i) table `nyse`

stock	price
att	92.35
lucent	87.45

(ii) table `tse`

Result of SPLIT$_{\text{by } \texttt{xge}}$(`close`)

Figure 2: Illustration of operator definitions.

We now give the formal definitions of the four restructuring operators, adapted from [GLS96]. We use the abbreviation $\vec{A}_{i:j}$ to denote the sequence $A_i, ..., A_j$.

Definition 2 (Unfold) Let r be a relation over a scheme $\{\vec{A}_{1:n}\}$, where A_i are names. Then UNFOLD$_{\text{by } A_i \text{ on } A_j}(r)$ is a relation s over the scheme $\{\vec{A}_{1:(i-1)}, \vec{A}_{(i+1):(j-1)}, \vec{A}_{(j+1):n}, u_1, ..., u_m\}$, where $\{u_1, ..., u_m\}$ is the set of distinct values appearing in column A_i of r. The contents of s are defined as $s = \{(\vec{a}_{1:(i-1)}, \vec{a}_{(i+1):(j-1)}, \vec{a}_{(j+1):n}), \vec{v}_{1:m} \mid (\vec{a}_{1:(i-1)}, u_\ell, \vec{a}_{(i+1):(j-1)}, v_\ell, \vec{a}_{(j+1):n} \in r, 1 \leq \ell \leq m\}$.

As an illustration, applying UNFOLD$_{\text{by } \texttt{xge} \text{ on } \texttt{price}}$(`close`) to the table `close` of Figure 2(a) produces the table `uf-close` of Figure 2(b). Note that for unfold, all input column labels must be names.

Definition 3 (Fold) Let r be a relation over the scheme $\{\vec{A}, u_1, ..., u_m\}$, where $\vec{A}$ is a vector of names. Suppose the u_i are values from *dom*(B), and all entries appearing in columns $u_1, ..., u_m$ of r are elements of *dom*(C), for some names $B, C \notin \{\vec{A}\}$. Then FOLD$_{\text{by } C \text{ on } B}(r)$ is a relation s over the scheme $S(\vec{A}, B, C)$, defined as $s = \{(\vec{a}, u_i, v_i) \mid \exists \text{ a tuple } t \in r : t[\vec{A}] = \vec{a} \ \& \ t[u_i] = v_i\}$.

Fold requires input column labels to be either names or values coming from the domain of a common attribute. The effect of applying fold is to "flatten" the relation over such a scheme. E.g., applying FOLD$_{\text{by } \texttt{price} \text{ on } \texttt{xge}}$(`uf-close`), where `uf-close` is the table in Figure 2(b) yields the table `close` in Figure 2(a).

[2] Each operator can be defined to manipulate a *set* of relations, thus giving closure. For easy exposition, we only show their definition on a single input relation.

```
CREATE VIEW  uf-close   (stock, X) AS
(Q1)         SELECT     stock, T.price
             FROM       close T, T.xge X
CREATE VIEW  close     (stock, xge, price) AS
(Q2)         SELECT    stock, X, T.X
             FROM      uf-close -> X, uf-close T
             WHERE     X <> 'stock'
CREATE VIEW  X         (stock, price) AS
(Q3)         SELECT    stock, price
             FROM      close T, T.xge X
CREATE VIEW  close     (stock, xge, price) AS
(Q4)         SELECT    T.stock, X, T.price
             FROM      -> X, X T
```

Figure 3: SchemaSQL Queries corresponding to the four Restructuring Operators.

Definition 4 (Split) Let r be a relation over a scheme $R(A_1, ..., A_n)$ such that R is a name. Then SPLIT$_{\text{by } A_i}(r)$ is a set of relations obtained as follows. For each distinct value $u \in \pi_{A_i}(r)$, SPLIT$_{\text{by } A_i}(r)$ contains a unique relation u over the scheme $U(\vec{A}_{1:(i-1)}, \vec{A}_{(i+1):n})$, defined as $u = \{t[\vec{A}_{1:(i-1)}, \vec{A}_{(i+1):n}] \mid t \in r \ \& \ t[A_i] = u\}$.

As an illustration, applying SPLIT$_{\text{by } \texttt{xge}}$(`close`) to the table `close` in Figure 2(a) produces the two tables 'nyse' and 'tse' in Figure 2(c). Note that split requires the input relation label to be a name.

Definition 5 (Unite) Let $u_1, ..., u_m$ be the set of all relations in a given database, such that each relation label u_i is an element of *dom*(B), for some fixed name B. Suppose also that they all have a common scheme $\{\vec{A}\}$. Then UNITE$_{\text{by } B|\text{Cond}}()$, where Cond is any boolean combination of conditions of the form B relOp const, is a relation r over the scheme $\{B, \vec{A}\}$, defined as $r = \{t \mid \exists \tau \in u_i, \text{ for some } u_i \text{ where Cond}(u_i) : t[\vec{A}] = \tau[\vec{A}] \ \& \ t[B] = u_i\}$. When Cond is "true", we omit it.

The operation UNITE$_{\text{by } \texttt{xge}}()$, when applied against the database consisting of the tables in Figure 2(c):(i) & (ii), yields the table in Figure 2(a). Note that unlike the preceding operators, unite does not take arguments. It is always implicitly applied against the current state of a database. When so applied, the relation labels in the database are evaluated against the conditions Cond, and those that satisfy them are manipulated by unite.

For further illustration, in Figure 3, we present examples of SchemaSQL queries corresponding to the four operators above, against the database scheme of Figure 2: Q1: UNFOLD$_{\text{by } \texttt{xge} \text{ on } \texttt{price}}$(`close`); Q2: FOLD$_{\text{by } \texttt{price} \text{ on } \texttt{xge}}$(uf-close); Q3: SPLIT$_{\text{by } \texttt{xge}}$(`close`); and Q4: UNITE$_{\text{by } \texttt{xge}}()$.

3.2 Translating from SchemaSQL to Extended Algebra

We start with an overview of our translation process. First, we check whether the schema label variables in a given SchemaSQL query are *properly constrained.* Intuitively, this means no column label variable ranges over labels whose attribute types are different, and no relation label variable ranges over relation labels whose schemes are different. E.g., a column label variable ranging over both `date` and `xge` is *not* properly constrained, nor is a relation label variable ranging over both 'ibm' and `stock`. Queries with schema label variables that are not properly constrained are not well-defined. With each schema label L, we can associate an *attribute type* as follows. If L is a name, its attribute type is L. If it is a value, then the name A such that $L \in dom(A)$ is said to be its attribute type. Finally, if a schema label variable `V` ranges over a set of schema labels all of whose attribute type is A, we say the attribute type of `V` is A. Essentially, properly constrained column label variables are those that have a well-defined attribute type, while properly constrained relation label variables range over relation labels with an identical scheme. In the full paper [LSS99], we give an algorithm for checking whether a `SchemaSQL` query is well-defined, which at once can also determine the attribute types associated with various labels and label variables.

A key idea used in attribute type detection is *meta-data specification*, expressed in the form of a Datalog program[3]. For instance, for the database scheme of Figure 1(a), the corresponding meta-data specification is given in Figure 1(b). The predicate $isa(X, \texttt{attribute})$ indicates the entry X is a value from the domain of `attribute`; predicate $colVar(X)$ means X is a column label variable, etc. The facts in the top part of the program say 'ibm', 'msft', etc. have `ticker` as their attribute type. The first rule says whenever `S` is a relation label variable with attribute type `ticker` and `T` is a tuple variable ranging over the tuples in the relations `S`, `T.xge` has `xge` as its attribute type. More interestingly, `T.PT` has `price` as its attribute type, whenever `T` is as above and `PT` is a column label variable with attribute type `priceType`. This idea was inspired by a similar idea, used in [LSS96] for handling semantic heterogeneity in multi-databases.

Once a query is found to be well-defined and the attribute types are detected, the next step is to "flatten" the various relations involved in the query. E.g., if tuple variable `T` ranges over one relation which is in unfolded form. Then we apply a fold operation to this table with relevant parameters. On the other hand, if `T` ranges over many relations which correspond to a split representation, we apply a unite operation to this set of relations. Combinations of the above scenarios may arise and are handled in a similar manner. As a concrete example, in the view definition in Figure 1(c), `T` ranges over many relations whose labels (ibm, msft, ...) are in the range of variable `S`. Thus, a unite operation is called for. However, each of these relations is in unfolded form, so a fold operation is also planned.

After flattening the relevant input relations, the next step is to apply a Cartesian product together with necessary selections and projections. These are identified from the `WHERE` clause and the `SELECT` statement, as usual. Two issues need special attention: (i) if a label variable appears in the relation position in the `CREATE VIEW` statement, the attribute type of that variable should also be included in the project list; (ii) domain expressions that appear in the `SELECT` statement and in the `WHERE` clause need to be modified so that they can be correctly applied as parameters to projection and selection on the Cartesian product of the flattened input tables.

Finally, if necessary an unfold and a split may have to be applied. The parameters for these operations are obtained by query analysis.

Consider a generic `SchemaSQL` query/view Q below.

```
CREATE   VIEW     rel (col1, ..., colk) AS
         SELECT   dom1, ..., domk
(Q):     FROM     decl1, ..., decln
         WHERE    conditions
```

Here, `coli` is either an identifier (i.e. name) or a domain variable, `domi` is a domain expression, and `decli` is a variable declaration.

A concrete example fitting the above template is given in Figure 1(c). It asks "for each tech stock, and for each price type, find the price value on each date and arrange it into one relation per exchange, where each such relation has one column per stock (i.e. ticker)".

Figure 4 gives an algorithm for translating single block `SchemaSQL` queries without aggregation into an expression in the extended algebra described in the preceding section. It invokes the following algorithms: (i) FLATTEN $(r, \mathbf{C})$, an algorithm which takes as input a relation label or variable r and the constraints $\mathbf{C}$ that apply to r (if any), and flattens r as explained above; (ii) $wellDefined(\texttt{Q}, \texttt{S}, \mathbf{C}_{\texttt{S}})$, which takes in a `SchemaSQL` query `Q`, a schema label variable `S` in `Q`, and any applicable constraints $\mathbf{C}_{\texttt{S}}$ on `S`, and checks whether `S` is properly constrained. Details of these algorithms, suppressed here for lack of space, can be found in [LSS99].

Example 1 (Translating `SchemaSQL` to Algebra) Consider the database scheme and `SchemaSQL` query/view shown in Figure 1(c). This query is clearly well defined. Application of step 3 of the algorithm yields the expression template:

$$\text{SPLIT}_{\text{by } splitPar}(\text{UNFOLD}_{\text{by } ufPar1 \text{ on } ufPar2}(\pi_{splitPar, mod(\texttt{T.date}), mod(\texttt{PT}), mod(\texttt{T.PT})}(\sigma_{mod(\texttt{S})=mod(\texttt{U.ticker}) \ \& \ mod(\texttt{U.busType})=\text{`tech'}}[\text{FLATTEN}(\texttt{S}, \{\texttt{S} \neq \texttt{stock}\}) \times \text{FLATTEN}(\texttt{stock}, \text{``}true\text{''})]))).$$

In step 4, we can infer that since `X` ranges over the domain of `xge`, splitPar = `xge`. In step 5, we find that since the attribute type of `S` is `ticker` and that of `T.PT` is `price`, *ufPar1* = `ticker`, *ufPar2* = `price`. In step 6, we can identify the parameters for projection

[3] Our methodology and techniques do *not* depend on the specification language. Indeed, the "specification" could well be a C++ program. However, a declarative specification is much easier to read, and we chose Datalog for this reason.

Algorithm *Translate*(Q);
Input: A single block SchemaSQL query Q without aggregation;
Output: An expression E in the extended algebra, that is equivalent to Q under multiset semantics.

1. identify $r_1, ..., r_m$, the list of relations or relation label variables appearing in the declarations in the FROM clause;

2. for each schema var S {
//test if query is well-defined;

   identify the set of conditions $\mathbf{C}_S$ of the form S relOp const, in the WHERE clause;

   if not wellDefined(Q, S, $\mathbf{C}_S$)

   return(error); }

3. Build the expression:
//at this point, the attribute types of all schema
//vars is known from Algorithm wellDefined;
$E = \text{SPLIT}_{\text{by } splitPar}(\text{UNFOLD}_{\text{by } ufPar1 \text{ on } ufPar2}(\pi_{splitPar, mod(\text{dom1}),...,mod(\text{domk})}\ \sigma_{mod(\text{conditions})}\ [\text{FLATTEN}\ (r1) \times \cdots \times \text{FLATTEN}\ (rm)]))$,

   where mod(domi), mod(sc), *ufPar1*, *ufPar2*, and *splitPar* are determined as follows.

4. if rel (in the CREATE VIEW statement) is a constant {

   remove the split operation;

   remove the parameter *splitPar* from the project list; }

   else {

   let rel be a domain variable X, declared to range over the domain of attribute A;
   //A is thus a splitting column;
   then $splitPar = A$; }

5. if no coli in the CREATE VIEW statement is a domain var

   remove the unfold operation;

   else { //unfold is needed;

   if more than one col is a domain var return(error);

   let coli be the unique col which is a domain var, say X;
   obtain the attribute type B of X from algorithm *wellDefined*;

   obtain the attribute type C of domi from algorithm *wellDefined*;

   $ufPar1 = B$; $ufPar2 = C$; }

6. for each domi {

   if domi is of the form tupleVar.col {
   find the relation label (variable) reln associated with tupleVar;

   find the attribute type attr of the domain expression tupleVar.col from the meta-data specs;

   mod(domi) = flat-reln.attr, where flat-reln refers to the name of the relation obtained by flattening reln; }

   if domi is of the form domVar, declared as FROM tupleVar.col domVar, where tupleVar and col are as above, the treatment is similar to the above;

   if domi is of the form colVar, declared as FROM reln-> colVar, where reln is either a relation label or a relation label variable {
   find the attribute type attr of colVar from algorithm *wellDefined*;

   mod(domi) = flat-reln.attr, where flat-reln refers to the name of the relation obtained by flattening reln; }

   if domi is of the form relVar, declared as FROM -> relVar {
   find the attribute type attr of relVar from algorithm *wellDefined*;

   mod(domi) = flat-relVar.attr, where flat-relVar refers to the name of the relation obtained by flattening relVar; }

   replace each domain expression in conditions using the same rules as above and set the result to mod(conditions); }

Figure 4: Algorithm for translating SchemaSQL queries into extended algebra.

as flat-S.date, flat-S.priceType, flat-S.price. The first three conditions in the WHERE clause simply ensure the variables S, PT are properly constrained. We then obtain the modified conditions flat-S.ticker = flat-stock.ticker and flat-stock.busType = 'tech' corresponding to the last two. These may be further simplified to flat-S.ticker = stock.ticker and stock.busType = 'tech'. These two conditions would be parameters for selection. The final algebraic expression obtained by the translation algorithm is:
$\text{SPLIT}_{\text{by xge}}(\text{UNFOLD}_{\text{by ticker on price}}(\pi_{\text{xge,flat-S.date,flat-S.priceType.flat-S.price}}(\sigma_{\text{flat-S.ticker=stock.ticker \& U.busType='tech'}}[\text{FLATTEN}\ (\mathtt{S}, \{\mathtt{S} \neq \mathtt{stock}\}) \times \mathtt{stock})])))$. ■

We have the following result, the proof of which is suppressed for lack of space.

Theorem 1 (Correctness of Algorithm Translate) *Algorithm Translate(*Q*) always translates a* SchemaSQL *query* Q *into an expression in the extended algebra that is equivalent to it whenever* Q *is well-defined and reports an error otherwise.* ■

4 A Physical Algebra

In this section, we propose some physical operators into which the restructuring operators of Section 3 can be compiled. The advantage is that the compiled expression can be conveniently used for choosing query execution strategies. As mentioned earlier, we are particularly interested in efficient but *non-intrusive* strategies, i.e. those that do not require any modifications or additions to the set of plan operators used by current relational query optimizers. At a conceptual level, an interesting observation that forms the basis of logical to physical algebra transformation is that SchemaSQL, with all its higher-order power, can be essentially reduced to SQL together with looping over attribute domains. For example, $\text{SPLIT}_{\text{by } A}(r)$ can be seen as repeatedly doing the selection $\sigma_{A=a_i}(r)$, $\forall a_i \in \pi_A(r)$.

The first physical operator is called *iterated selection.* It is a generalization of the idea discussed above. It comes in two flavors – *iterated selection by name* and *by value.*

Definition 6 (Iterated Selection) Let r be a relation with scheme $\{A_1, ..., A_n\}$, where A_i are all names, and let $\mathcal{C}$ be a set of conditions of the form A relOp const, or of the form A relOp B, where $A, B \in \{A_1, ..., A_n\}$. Suppose $\{\vec{B}\}$ is a set of attributes such that $\{\vec{B}\} \subseteq (\{\vec{A}\} - \{A_i\})$ and $A_j \in \{\vec{B}\}$, and $\{\vec{C}\}$ is a set of attributes such that $\{\vec{C}\} \subseteq (\{\vec{A}\} - \{A_i, A_j\})$. Then the operators *iterated selection by name*, *iSel*$_n(r, A_i, \mathcal{C}, \vec{B})$, applied to relation of r, by A_i. subject to conditions $\mathcal{C}$, and *iterated selection by value*, *iSel*$_v(r, A_i, A_j, \mathcal{C}, \vec{C})$, applied to relation of r, by A_i on A_j, subject to conditions $\mathcal{C}$, are defined as follows.

Iterated Selection by name: $iSel_n(r, A_i, \mathcal{C}, \{\vec{B}\})$: Let $\{a_1, ..., a_k\}$ be the set of distinct A_i-values in $\pi_{A_i}(r)$. Then this operator produces k output tables $rel_{a_p}(\vec{B})$, $1 \leq p \leq k$, such that $rel_{a_p} = \{t[\vec{B}] \mid t \in r \ \& \ t[A_i] = a_p \ \& \ t \text{ satisfies } \mathcal{C}\}$.

Iterated Selection by value: $iSel_v(r, A_i, A_j, \mathcal{C}, \{\vec{C}\})$: Let $\{a_1, ..., a_k\}$ be the set of distinct A_i-values in $\pi_{A_i}(r)$. Then the operator produces k tables $rel_{a_p}(\vec{C}, a_p)$, $1 \leq p \leq k$, such that $rel_{a_p} = \{t[\vec{C}, A_j] \mid t \in r \ \& \ t[A_i] = a_p \ \& \ t \text{ satisfies } \mathcal{C}\}$. ■

Remarks: (1) $iSel_n$ intuitively partitions relation r on column A_i (after filtering out tuples violating conditions $\mathcal{C}$), and writes each partition into a separate output table, dropping attributes not in $\vec{B}$, including A_i. The scheme of each output relation produced is set to $\{\vec{B}\}$. Note that the value of attribute A_j is written under column named A_j in the output relations. (2) $iSel_v$ is almost identical to $iSel_n$ except the output scheme of $\texttt{rel}_a$ consists of $\{\vec{C},\ a\}$, so tuples in the partition associated with $\texttt{rel}_a$ are written into this relation, so that all A_j-values are lined up against column 'a'. (3) For the sake of generality, the above definition avoids committing to any implementation strategy. Several strategies will be proposed in the sequel. (4) Like the restructuring operators, the physical operators also can be easily extended to manipulate sets of relations. We focus on applications of these operators to single relations, for clarity. Here is an example of iterated selection.

tabA

stock	price
att	91.56

tabB

stock	price
att	92.35

tabC

stock	xge	price
att	nyse	91.56
lucent	nyse	89.45

tabD

stock	xge	price
att	tse	92.35
lucent	tse	87.45

tabE

stock	nyse
att	91.56
lucent	89.45

tabF

stock	tse
att	92.35
lucent	87.45

Figure 5: Examples for Physical Operators.

Example 2 (Illustrating $iSel_n$ and $iSel_v$) Applying $iSel_n(\texttt{close}, \texttt{xge}, \{\texttt{price} > 90\}, \{\texttt{stock}, \texttt{price}\})$ to the relation `close` in Figure 2(a) yields the two output relations `tabA` and `tabB` of Figure 5. As another example, applying $iSel_v(\texttt{close}, \texttt{xge}, \texttt{price}, \text{“}true\text{”}, \{\texttt{stock}\})$ to the same input table as above produces the two tables `tabE` and `tabF` of Figure 5. ■

The second operator is called *iterated projection.* The basic idea is that if an input table contains many column labels $a_1, a_2, ...$ corresponding to values from the domain of some name A, this operator will output projections of the input table on different sets of column labels such that each label a_i ends up in a different projection. Like iterated selection, there are two flavors – corresponding to "by name" and "by value".

Definition 7 (Iterated Projection) Let r be a relation with scheme $\{\vec{A}, a_1, ..., a_m\}$, where As are names, $a_i \in dom(A), 1 \leq i \leq m$, for some name $A \notin \{\vec{A}\}$, and for $1 \leq i \leq m$, all entries in columns a_i are values from $dom(B)$, for some fixed name $B \notin \{\vec{A}\}$, $B \neq A$. Suppose $\mathcal{C}$ is a set of conditions of the form L_1 relOp const, or of the form L_1 relOp L_2, where L_1, L_2 are each, either one of the attributes in $\vec{A}$, or a label a_i, for some $1 \leq i \leq m$. Then the operators *iterated projection by name*, $iProj_n(r, A, B, \mathcal{C})$, and *iterated projection by value*, $iProj_v(r, A, B, \mathcal{C})$, applied to relation of r, by B on A, and subject to conditions $\mathcal{C}$, are defined as follows.

Iterated Projection by name $iProj_n(r, A, B, \mathcal{C})$: This operator produces m output tables $rel_{a_p}(\vec{A}, A, B)$, $1 \leq p \leq m$, such that $rel_{a_p} = \{t[\vec{A}] \cdot \text{‘}a_p\text{’} \cdot t[a_p] \mid t \in r \ \& \ t \text{ satisfies } \mathcal{C}\}$.

Iterated Projection by value $iProj_v(r, A, B, \mathcal{C})$: This operator produces m output tables $rel_{a_p}(\vec{A}, a_p)$, $1 \leq p \leq m$, such that $rel_{a_p} = \{t[\vec{A}, a_p] \mid t \in r \ \& \ t \text{ satisfies } \mathcal{C}\}$. ■

Remark: Iterated projection by name and value are similar: the core step involved in each of these operators is projecting r onto each of the column label sets $\{\vec{A}, a\}, \forall a \in \{a_1, ..., a_m\}$; in the former case, we also append the corresponding A-value 'a' and re-label column 'a' as B, whereas in the latter we don't. Once again, notice the lack of commitment to any specific implementation strategy.

Example 3 (Illustrating $iProj_n$ and $iProj_v$) Applying $iProj_n(\texttt{uf-close}, \texttt{xge}, \texttt{price}, \text{“}true\text{”})$ to the table in Figure 2(b) results in the two tables `tabC` and `tabD` of Figure 5. As another example, applying $iProj_v(\texttt{uf-close}, \texttt{xge}, \texttt{price}, \text{“}true\text{”})$ to the same input table results in the tables `tabE` and `tabF` of Figure 5. ■

We next show that each of the four restructuring operators – unfold, fold, split, and unite – can be computed using iterated selection and projection together with classical relational algebraic operations. For a set of relations $\mathcal{R}$ we denote their natural join as $\bowtie(\mathcal{R})$ and their union, whenever their schemes are identical, as $\bigcup(\mathcal{R})$.

Theorem 2 1. Let r be any relation over the scheme $\{\vec{A}\}$ consisting of only names, and let $A_i, A_j \in \{\vec{A}\}$ be any two distinct names in the scheme. Then the output computed by the expressions $\text{UNFOLD}_{\text{by } A_i \text{ on } A_j}(r)$ and $\bowtie(iSel_v(r, A_i, A_j, \text{“}true\text{”}, (\{\vec{A}\} - \{A_i, A_j\})))$ are equivalent.

2. Let r be a relation over the scheme $\{\vec{A}, b_1, ..., b_m\}$, where $\{\vec{A}\}$ is a set of names, $b_1, ..., b_m$ are values from the domain of some name B, and the

entries in columns b_i of r are from the domain of another name C. Then the output computed by the expression $\text{FOLD}_{\text{by } C \text{ on } B}(r)$ and $\bigcup(iProj_n(r, B, C, \text{"}true\text{"}))$ are equivalent.

3. Let r be a relation over the scheme $\{\vec{A}\}$, such that $\{\vec{A}\}$ includes two names B, C. Then the output computed by the expressions $\text{SPLIT}_{\text{by } B}(r)$ and $iSel_n(r, B, C, \text{"}true\text{"}, (\{\vec{A}\} - \{B\}))$ are equivalent.

4. Let $b_1, ..., b_k$ be relations with the identical scheme $\{\vec{A}\}$, such that $b_1, ..., b_k$ are values from the domain of some name B. Then the output computed by the expression $\text{UNITE}_{\text{by } B}()$ and the expression $\bigcup_{1 \leq i \leq k}(\{\text{'}b_i'\} \times \mathtt{rel}_{b_i})$ over the scheme $\{B, \vec{A}\}$ are equivalent. ■

5 Implementation Strategies

We can obtain several implementation strategies for computing `SchemaSQL` queries by considering ways in which the physical operators introduced in the previous section can be implemented. In addition, we will see that sometimes it makes sense to compute a logical operation such as unfold directly. The exact choice of the operator to be used for query execution and the strategy for the chosen operator should be a cost-based decision made by the query optimizer. While these issues are beyond the scope of the paper, we illustrate by examples that different operators and strategies may lead to efficient executions under different circumstances. Also, for lack of space, we only give strategies for iterated selection by value, iterated projection by name, unfold, and unite. Strategies for other operators are discussed in the full version [LSS99].

Strategies for Iterated Selection:

We give two strategies for iterated selection by name. The first strategy does not involve sorting, while the second strategy incurs an initial sorting overhead.

Strategy *iSel I* **for** $iSel_v$(**rel, By, On, Conditions, ProjList**):

1. find the set of distinct values $u_1, ..., u_k$ in column By of relation rel, using `SELECT DISTINCT ...`;
2. for each distinct value found u_i {
   create table tab_i(ProjList, u_i) ;
   `INSERT INTO` $\mathbf{tab}_i$
   `SELECT` ProjList, On
   `FROM` rel
   `WHERE` Conditions `AND` By = u_i }

Strategy *iSel II* **for**
$iSel_v$(**rel, By, On, Conditions, ProjList**):

1. sort relation rel on the column By;
2. while there are unread blocks {
   - read the next block of tuples from rel;
   - if the By-value of the block, u_i, is new create a new table tab_i(ProjList, u_i);
   - project the block on the columns (ProjList, On) and write it into tab_i; }

Strategies for Unfold:

Strategies for the unfold operator can be derived by exploiting the equivalence result in Theorem 2. First an iterated selection is performed, then the resulting relations are joined to obtain the unfolded relation. The unfold strategy *unfold I* is obtained by using strategy *iSel I* for the iterated selection, while *unfold II* uses *iSel II*.

It turns out there is a third, direct strategy for the unfold operator.

(Direct) Strategy *unfold III*:

1. find the set of distinct values $u_1, ..., u_k$ in column By of relation rel, using `SELECT DISTINCT ...`;
2. do `CREATE TABLE` **`uf-rel`**(ProjList, $u_1, ..., u_k$);
3. form a partition, say $\langle P_1, ..., P_m \rangle$ of **`rel`** such that each P_i is a maximal set of tuples which agree on ProjList but disagree on column By;
   //this may be done by sorting on ProjList, By, and using a bitvector
   //representation for the membership of the u_is in each partition;
4. for each cell in the partition, $P_i = \{(\vec{p}, u_1, v_1), ..., (\vec{p}, u_k, v_k)\}$
   write the tuple $(\vec{p}, v_1, ..., v_k)$ into **`uf-rel`**;

The direct strategy *unfold III* avoids intermediate storage costs for storing the output of iterated selection (unlike Strategies *unfold I* and *unfold II*) as well as the final join cost. However, it incurs a sorting overhead. While it may superficially appear as though Strategy III, implemented judiciously to benefit from block I/O, will always outperform the other strategies, we will show later in the section that expressing a query using the physical operators sometimes allows to exploit unique optimization opportunities. Our non-intrusive implementation of *unfold III* strategy (Section 6) is less efficient than *unfold I* strategy due to the tuple-at-a-time nature of the implementation. But our C++ implementation of *unfold III* turned out to be significantly more efficient.

Strategies for Iterated Projection:

In this paragraph, we give two strategies for iterated projection by name. The first one involves k separate SPJ queries each of which will be used to populate one output table. The second one starts by creating the output tables (schemas); for each input tuple t, it writes the appropriate "piece" of t into the relevant output table. A block-based variant of this strategy can be applied where a block of records is read into memory and processed instead of one tuple, at a time.

Strategy *iProj I* **for** $iProj_n$(**rel, By, On, Conditions, ProjList**):

1. let $u_1, ..., u_k$ be the column labels of rel that correspond to $dom(B)$;
2. for $(i = 1; i <= k; i++)$
   `CREATE TABLE` $\mathbf{tab}_i$(ProjList, B, C);
   `INSERT INTO` $\mathbf{tab}_i$
   `SELECT` ProjList, 'u_i', u_i
   `FROM` rel
   `WHERE` Conditions

Strategy *iProj II* **for**
$iProj_n$(**rel, By, On, Conditions, ProjList**):

1. let $u_1, ..., u_k$ be the column labels of rel that correspond to $dom(B)$;
2. for $(i = 1; i <= k; i++)$
   - `CREATE TABLE` tab_i(ProjList, B, C);
3. while there are unread tuples {
   - read the next tuple t from rel and test if it satisfies Conditions;
   - write t[ProjList] $\cdot$ 'u_i'$\cdot t[u_i]$ into tab_i; }

Strategies for Unite

Strategy *unite I* involves explicit creation of intermediate tables by "padding" each input table u_i by a column whose values are 'u_i'' for all tuples, and then unioning these intermediate tables. Strategy *unite II*, given below, avoids the creation of the intermediate tables.

(Direct) Strategy *unite II*:

let $u_1(col\vec{Labels}), ..., u_k(col\vec{Labels})$ be the set of all relations such that the labels $u_1, ..., u_k \in dom(B)$, for some name B;

1. `CREATE TABLE unite-rel`$(B, \text{colL}\vec{\text{abels}})$;
2. for $(i = 1; i <= k; i++)$ {
   - read each tuple t in relation u_i;
   - write the tuple '$u_i' \cdot t$ into `unite-rel`; }

In the following section, we illustrate some query optimization opportunities available to `SchemaSQL` processing – both at a logical and a physical level. In the full paper [LSS99], we establish several identities among expressions in the logical algebra and also discuss several physical optimizations in detail.

5.1 Query Optimization

To appreciate logical optimization, consider the queries `Q5, Q6` given below, expressed against the database of Figure 2. `Q5` is a straight projection of `uf-close` (Figure 2(b)) on column `stock`. In this case, the translated query $\pi_{\texttt{stock}}(\text{FLATTEN}\,(\texttt{T}, \text{"true"}))$ is equivalent to $\pi_{\texttt{stock}}(\text{FOLD}_{\texttt{by price on xge}}(\texttt{uf-close}))$ which in turn is equivalent to $\pi_{\texttt{stock}}(\texttt{uf-close})$. In general, whenever the projectList is disjoint with both the parameter lists of fold, we can push down projection, which in this case ends up removing fold. For `Q6`, by a similar reasoning, we can show that the translated query can be rewritten into the equivalent query $\pi_{\texttt{stock}}(\sigma_{\texttt{nyse}>90 \vee \texttt{tse}>90}(\texttt{uf-close}))$, which does not involve any restructuring operator. Likewise, there are several opportunities for logical optimization, which are based on operator identities. The decision of when to apply flatten or not should, however be based not only on logical equivalence, but on available indexes and properties of data. Another class of query equivalences have to do with operator commutativity: e.g., unfold and split commute whenever their parameter lists do not overlap.

```
SELECT   T.stock
FROM     uf-close T
        Q5
```

```
SELECT   T.stock
FROM     uf-close -> X, uf-close T
WHERE    X <> stock AND T.X > 90
                Q6
```

Next we consider physical optimization. Recall the translation of the `SchemaSQL` query in Figure 1(c) (see Example 1). The resulting expression is depicted as a tree in Figure 6(a).

Now, several execution strategies are possible. One option is to combine Cartesian product with selection and projection (as is done in classical relational optimization), and then apply unfold, followed by split. Clearly, any strategy for join may be used, and any of the proposed strategies for unfold and split may

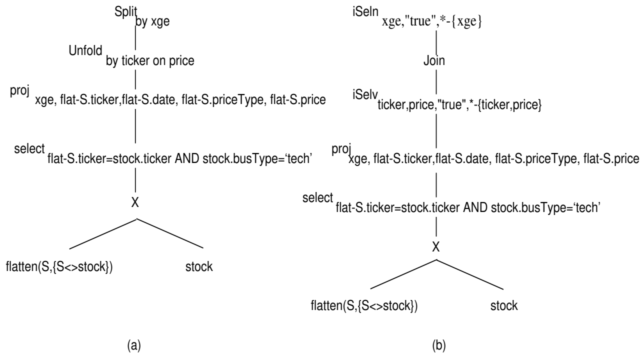

Figure 6: *(a) Operator Tree corresponding to the* `SchemaSQL` *query in Example 1; (b) Equivalent operator tree employing physical operators, where we use abbreviations: "*-$\{A\}$" means all attributes in the scheme except A.*

be used as well. Since the sets of parameters for unfold and split are disjoint, as argued above, the order of these operations can be swapped, leading to a slightly different execution strategy. So far, we have not utilized the physical operators. To do that, we must translate the restructuring operators in the tree into appropriate physical operators. Doing so leads to the tree shown in Figure 6(b).

Now, any of the strategies proposed for the physical operators can be used for computing them. In addition, examining the operators $iSel_v$ and $iSel_n$ in the tree, we notice that the steps needed to perform these operators can be combined. E.g., consider using strategy II for both $iSel_v$ and $iSel_n$. Then instead of first computing $iSel_v$ in full and then (after the join) computing $iSel_n$, the sorts required for these operations can be combined, and we can sort the table generated by projection on the columns `ticker, xge`. However, the join operation is done on the attributes `xge, date`, so it makes sense to sort the table on the columns `ticker, xge, date`, in that order. Now, conceptually, after $iSel_v$, the resulting tables (for different `ticker`-values) are already sorted on `xge, date`, and can be joined efficiently. As we join them, we can write the output tuples into different relations corresponding to different `xge`-values. In fact, even the intermediate output tables corresponding to the application of $iSel_v$ need not be explicitly created and stored. Thus, we can perform the three operations of $iSel_v$, $\bowtie$, and $iSel_n$ in one shot, on the fly.

6 Experiments

We conducted experiments using TPC-D benchmark data [TPC93] on NT workstation running DB2[4]. Our experiments were designed with the following objectives

- To demonstrate the feasibility of non-intrusive implementation of `SchemaSQL` on an SQL engine.
- To assess relative performance of different non-intrusive strategies.

---

[4] DB2 is a trademark of IBM.

As a possible indication of the efficiency that may be afforded by an *intrusive* implementation, we also implemented the various operations directly in C++, externally to the DBMS.

6.1 The iterated selection and unite operations

We implemented two non-intrusive strategies for the iterative select, viz., *iSel I* and *iSel II* (Section 5). The non-intrusive implementation of *iSel II* uses `SQL ORDER BY` facility for the sorting, and a cursor to scan the sorted tuples. We also implemented *unite I* and *unite II* strategies, both non-intrusively. We used the `lineItem` table in the TPC-D benchmark as a basis for our experiments, which we conducted for varying sizes, while keeping the number of distinct values in the "By" column almost constant. The following table summarizes the timing results. 'Size of table' indicates the number of tuples in the input table (for iSel operations) or output table (for unite operations). 'Number of split tables' indicates the number of tables generated (for iSel operations) or the number of tables united (for unite operations). The experiments show a linear performance for all operations in the range of table sizes we tested. Performance of *iSel I* and *iSel II* are somewhat similar, with *iSel II* becoming less efficient as size of table increases. This may be the result of sorting overhead. Performance of *unite II* is better than that of *unite I* since *unite II* avoids the generation of intermediate tables. Execution times are in seconds.

size	splits	*iSel I*	*iSel II*	*unite I*	*unite II*
2770	326	108	100	176	31
5335	329	132	145	198	36
6712	330	158	180	224	37
14042	333	236	312	334	71
20702	334	281	429	389	101

We also coded, in C++, a direct implementation based on *iSel II* strategy. It differs from *iSel II* in that using a (external) sort-merge sorting algorithm, the generation of output tables is built into the merge phase of the sort-merge algorithm. Hence the C++ implementation avoids storing the sorted table. The performance was surprising: about an oder of magnitude faster than the non-intrusive implementations. Note that the direct C++ implementation avoids significant database overheads such as logging, maintaining the catalog tables, and other similar foot prints. Nevertheless, these experiments suggest that an efficient *intrusive* implementation of `SchemaSQL` might yield significant benefits. We are currently investigating this hypothesis.

6.2 The unfold and fold operations

We implemented *unfold I* and *unfold III* strategies (Section 5) both non-intrusively. The *unfold II* strategy differs from *unfold I* only in the underlying iSel strategy and its performance can be derived by adding the difference in *iSel I* and *iSel II* to that of *unfold I*. We also implemented two strategies for the fold operation. The following table summarizes the timing results. We used the projection of the TPC-D `lineitem` table on `orderkey, linenumber, extendedprice` for these experiments. Since the FD $\{$`orderkey, linenumber`$\} \rightarrow$ `extendedprice` holds in this table, the number of tuples in the projection is unchanged.

The experiments show a linear performance for all operations in the range of table sizes we tested. In the table below, the columns u-size, uI, and uIII represent the size of unfolded table, unfold I, and unfold III algorithms, respectively. Number of unfolded columns is 7. The relative inefficiency of *unfold III* is attributed to the tuple-at-a-time nature of this non-intrusive implementation. Our C++ implementation of *unfold III* shows a performance improvement similar to that of the iterated select operation: more than an order of magnitude faster. The comments we made regarding the C++ implementation of iterated select are valid here too. In addition, the C++ implementation uses the `fstrem` library class that performs block I/O. The strategies *fold I* and *fold II* differ in that *fold I* explicitly generates intermediate tables that are UNIONed to produce the folded table, while *fold II* avoids the generation of these intermediate tables (in this respect, they are similar to *unite I* and *unite II*, discussed earlier). Their performance figures reflect the more efficient nature of *fold II*. The parameter 'no. of unfolded columns' refers to the number of column labels which are values that belong to the domain of some name.

size	u-size	*uI*	*uIII*	*fold I*	*fold II*
24146	6000	40	104	41	17
48214	12000	79	210	74	33
60175	15000	89	263	88	48
120515	30000	193	520	182	87
180566	45000	307	801	336	149

7 Summary and Future Work

Efficient implementation of `SchemaSQL` on a single RDBMS is the main focus of this paper. As mentioned in Section 1, there are several important applications which motivate this. We developed a logical algebra and a physical algebra. We gave an algorithm for translating `SchemaSQL` queries into equivalent expressions in the logical algebra and established equivalences between logical operators and expressions in the physical algebra. We illustrated several opportunities for optimization at the logical and physical level. We proposed several implementation strategies for the various operators, all of which are non-intrusive in the sense that they require no additions to the plan operators used by existing `SQL` systems. We conducted a series of experiments based on the TPC-D bench mark data and showed: (i) the feasibility of the various proposed strategies and (b) their relative performances.

In this paper, we confined ourselves to single block SchemaSQL queries without aggregation. In the full paper [LSS99], we address aggregation and nested queries. Our experiments showed the possible promise of intrusive implementations. We are currently investigating this hypothesis. For lack of space, we could only sketch the optimization opportunities available. A comprehensive study of cost-based query optimization as well as development of schema independent indexes for SchemaSQL query processing is a promising direction of work. Many "classical" optimization problems like containment and query answerability using views, acquire a new twist in the SchemaSQL context because of data/schema interplay. We are investigating some of these issues.

Acknowledgements: We would like to thank Keir B. Davis who implemented the operations in C++ and helped with the creation of TPC-D benchmark tables. Lakshmanan's work was supported in part by NSERC (Canada) and Sadri's work was supported by NSF (USA).

References

[AIS93] Agrawal, R., Imielinski, T., and Swami, A. Database Mining: A Performance Perspective. *IEEE TKDE*, 5(6):pp 914-925, 1993.

[BLT86] J.A. Blakeley, P.A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. *Proc. ACM SIGMOD*, 61–71, May 1986.

[CKPS95] S. Chaudhuri et al. Optimizing Queries with Materialized Views. *ICDE*, March 1995.

[CKW93] Chen W., Kifer M., and Warren D.S. Hilog: A foundation for higher-order logic programming. *Jl. of Logic Prog.*, 15(3):187–230, 1993.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate Query Processing in Data Warehousing Environments. *VLDB*, 358–369, Sept. 1995.

[GHRU97] H. Gupta et al. Index Selection for OLAP. *ICDE*, May 1997.

[GL98] Gingras F. and Lakshmanan L.V.S. nD-SQL: a multi-dimensional language for interoperability and olap. *VLDB*, 1998.

[GLRS93] Grant, J. et al. Query Languages for Relational Multidatabases. *VLDB Journal*, 2(2): pp 153-171, 1993.

[GLS96] Gyssens, Marc, Lakshmanan, L.V.S., and Subramanian, S. N. Tables as a paradigm for querying and restructuring. In *Proc. ACM Symp. on PODS*, June 1996.

[GM96] A. Gupta and I.S. Mumick. What is the Data Warehousing Problem? Are Materialized Views the Answer. *VLDB*, 1996.

[GMS93] Gupta, A., Mumick, I.S., and Subrahmanian, V.S. Maintaining views incrementally. *ACM SIGMOD*, 1993.

[HGW+95] Hammer, J. et al. The Stanford Data Warehousing Project. *Data Engg Bulletin*, 18(2), June, 1995.

[HRU96] V. Harinaryanan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. *ACM SIGMOD*, 205–216, May 1996.

[KKS92] Kifer, M., Kim, W., and Sagiv, Y. Querying Object-Oriented Databases. *ACM SIGMOD*, 393-402, 1992.

[KZ95] Krishnamurthy, R., and Zloof, M. RBE: Rendering By Example *ICDE*, 1995.

[KLK91] Krishnamurthy, R., Litwin, W., and Kent, W. Language features for interoperability of databases with schematic discrepancies. *ACM SIGMOD*, 40–49, 1991.

[LMSS95] A.Y. Levy et al. Answering Queries Using Views. *ACM Symp. on PODS*, May 1995.

[LSS93] Lakshmanan L.V.S., Sadri F., and Subramanian I. N. On the logical foundations of schema integration and evolution in heterogeneous database systems. *DOOD '93*, 81–100. LNCS-760, Dec. 1993.

[LSS96] Lakshmanan L.V.S., Sadri F., and Subramanian, I. N. SchemaSQL – a language for querying and restructuring multidatabase systems. *VLDB*, 239–250, Bombay, India, September 1996.

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

[LSS99] Lakshmanan L.V.S., Sadri F., and Subramanian I. N. On An Efficient Implementation of SchemaSQL Technical Report, IIT Bombay, 1999.

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

[MTW97] Miller R.J. et al. DataWeb: Customizable Database Publishing for the Web. *IEEE MultiMedia* 4(4): 14-21 (1997).

[Ros92] Ross, K.. Relations with relation names as arguments: Algebra and calculus. *ACM Symp. on PODS*, 346–353, June 1992.

[SAB+95] Subrahmanian, V.S. et al. HERMES: Heterogeneous Reasoning and Mediator System. Tech. report, University of Maryland, College Park, MD, 1995.

[SDJL96] D. Srivastava et al. Answering Queries with Aggregation Using Views. *VLDB*, Sept 1996.

[Sub97] S.N. Subramanian. *A Foundation for Integrating Heterogeneous Data Sources.* PhD Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1997.

[SV98] S.N. Subramanian and S. Venkataraman. Query Optimization Using Restructuring-Views. IBM Internal Report, 1998. (Submitted for publication.)

[TSY96] O. Tsatalos et al. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB Jl*, 5(2), April 1996.

[TPC93] TPC. TPC Benchmark^{TM} D (Decision Support). Working draft 6.0, Transaction Processing Performance Council, August 1993.

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

[YPAG98] Yerneni, R. et al. Fusion Queries Over Internet Databases *EDBT*, pp 57-71, 1998.