Incremental Maintenance of Schema-Restructuring Views*

Andreas Koeller and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{koeller|rundenst}@cs.wpi.edu

Abstract. An important issue in data integration is the integration of semantically equivalent but schematically heterogeneous data sources. Declarative mechanisms supporting powerful source restructuring for such databases have been proposed in the literature, such as the SQL extension *SchemaSQL*. However, the issue of incremental maintenance of views defined in such languages remains an open problem.
We present an incremental view maintenance algorithm for schema-restructuring views. Our algorithm transforms a source update into an incremental view update, by propagating updates through the operators of a *SchemaSQL* algebra tree. We observe that schema-restructuring view maintenance requires transformation of data into schema changes and vice versa. Our maintenance algorithm handles any combination of data updates or schema changes and produces a correct sequence of data updates, schema changes, or both as output. In experiments performed on our prototype implementation, we find that incremental view maintenance in *SchemaSQL* is significantly faster than recomputation in many cases.

1 Introduction

Information sources, especially on the Web, are increasingly independent from each other, being designed, administered and maintained by a multitude of autonomous data providers. Nevertheless, it becomes more and more important to integrate data from such sources [13, 11]. Issues in data integration include the heterogeneity of data and query models across different sources, called model heterogeneity [3] and incompatibilities in schematic representations of different sources even when using the same data model, called schema heterogeneity [13, 11]. Much work on these problems has dealt with the integration of schematically different sources under the assumption that all "data" is stored in tuples and all "schema" is stored in attribute and relation names. We now relax

* This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 99–88776.

this assumption and focus on the integration of heterogeneous sources under the assumption that schema elements may express data and vice versa.

One recent promising approach at overcoming such schematic heterogeneity are *schema-restructuring query languages*, such as *SchemaSQL*, an SQL-extension devised by Lakshmanan et al. [11, 12]. Other proposals include IDL by Krishnamurthy et al. [9] and HiLog [2]. These languages, in particular *SchemaSQL*, support querying schema (such as lists of attribute or relation names) in SQL-like queries and also to use sets of values obtained from *data tuples* as *schema* in the output relation. This extension leads to more powerful query languages, effectively achieving a transformation of semantically equivalent but syntactically different schemas [11] into each other.

Previous work on integration used either SQL-views, if the underlying schema agreed with what was needed in the view schema [14], or translation programs written in a programming language to reorganize source data [3]. We propose to use *views* defined in schema-restructuring languages in a way analogous to SQL-views. This makes it possible to include a larger class of information sources into an information system using a query language as the integration mechanism. This concept is much simpler and more flexible than ad-hoc "wrappers" that would have to be implemented for each data source. It is also possible to use or adapt query optimization techniques for such an architecture.

However, such an integration strategy raises the issue of maintaining schema-restructuring views, which is an open problem. As updates occur frequently in any database system, view maintenance is an important topic [1]. View maintenance in a restructuring view is different from SQL view maintenance, due to the disappearance of the distinction between data and schema, leading to new classes of updates and update transformations. In this paper, we present the first incremental maintenance strategy for a schema-restructuring view language, using *SchemaSQL* as an example.

1.1 Motivating Example

Consider the two relational schemas in Fig. 1 that are able to hold the same information and can be mapped into each other using *SchemaSQL* queries. The view query restructures the input relations on the left side representing airlines into attributes of the output relations on the right side representing destinations. The *arrow*-operator (->) attached to an element in the FROM-clause of a *SchemaSQL*-query allows to query schema elements, giving *SchemaSQL* its meta-data restructuring power. Standing by itself, it refers to "all relation names in that database", while attached to a relation name it means "all attribute names in that relation".

SchemaSQL is also able to transform data into schema. For example, *data* from the attribute Destination in the input schema is transformed into *relation names* in the output schema, and vice versa *attribute names* in the input (Business and Economy) are restructured into *data*.

Now consider an update to one of the base relations in our example. Let a tuple $t(\mathsf{Destination} \Rightarrow \mathsf{Berlin}, \mathsf{Business} \Rightarrow 1400, \mathsf{Economy} \Rightarrow 610)$ be added to

BA

Destination	Business	Economy
Paris	1200	600
London	1100	475

LH

Destination	Business	Economy
Paris	1220	700
London	1180	500

⇒

```
create view
  CITY(Class, AIRLINE) AS
select CLASS,FLIGHT.CLASS
from
  -> AIRLINE,
  AIRLINE FLIGHT,
  AIRLINE-> CLASS,
  FLIGHT.Destination CITY
where
CLASS<>'Destination' and
FLIGHT.CLASS <= 1100;
```

⇒

LONDON

Class	BA	LH
Business	1100	*null*
Economy	475	500

PARIS

Class	BA	LH
Economy	600	700

Fig. 1. A Schema-Restructuring Query in *SchemaSQL*.

the base table LH (a **data update**). The change to the output would be the addition of a new relation Berlin (a **schema change**) with the same schema as the other two relations. This new relation would contain one tuple t(Type $\Rightarrow$ Economy, BA $\Rightarrow$ *null*, LH $\Rightarrow$ 610). In this example, a data update is transformed into a schema change, but all other combinations are also possible. The effect of the propagation of an update in such a query depends on numerous factors, such as the input schema, the view definition, the set of unique values in the attribute Destination across *all* input relations (city names), and the set of input relations (airline codes). For example, the propagation would also depend on whether other airlines offer a flight to Berlin in the Economy-class, since in that case the desired view relation already exists.

1.2 Contributions

We propose to use schema-restructuring *query* languages to define *views* over relational sources and we solve several new problems that arise, using *SchemaSQL* as an example. We observe that, due to the possible transformation of "schema" into "data" and vice-versa, we must not only consider data updates (DUs) for *SchemaSQL*, but also schema changes (SCs). A consequence is that, as shown in this paper, using the standard approach of generating query expressions that compute some kind of "delta" relation Δ between the old and the new view after an update is not sufficient, since the schema of Δ would not be defined. Our algorithm in fact transforms an incoming (schema or data) update into a sequence of schema changes and/or data updates on the view extent.

The contributions of this work are as follows: (1) we identified the new problem of schema-restructuring view maintenance, (2) we gave an algebra-based solution to the problem, (3) we proved this approach correct, (4) we implemented a prototype and assessed performance experimentally.

This work is different from previous approaches in view maintenance since the problem of view maintenance of schema-restructuring views is fundamentally different from the traditional view maintenance problem, as we argue in Sec. 3.2.

1.3 Outline of Paper

Section 2 reviews some background on *SchemaSQL*, in particular the algebra operators used in *SchemaSQL* evaluation. Section 3 explains our view maintenance strategy and Section 4 gives an outline of a proof for our approach. Finally, Sections 5 and 6 give related work and conclusions, respectively.

2 Background

2.1 *SchemaSQL*

In relational databases it is possible to store equivalent data in different schemas that are incompatible when queried in SQL [13]. However, for information integration purposes it is desirable to combine data from such heterogeneous schemas. *SchemaSQL* is an SQL derivative designed by Lakshmanan et al. [11] which can be used to achieve schema-restructuring transformations of relational databases. In [12], Lakshmanan et al. describe an extended algebra and algebra execution strategies to implement a *SchemaSQL* query evaluation system. It extends the standard SQL algebra which uses operators such as $\sigma(R)$, $\pi(R)$, and $R \bowtie S$ by adding four operators named UNITE, FOLD, UNFOLD, and SPLIT originally introduced by Gyssens et al. [6] as part of their "Tabular Algebra". Lakshmanan et al. show that any *SchemaSQL* query can be translated into this extended algebra.

***SchemaSQL* Algebra Operators** We will give an overview over the four operators introduced in [12]. Due to space consideration, we will not give precise mathematical definitions but rather refer to our Technical Report [8]. Additionally, Lakshmanan's original definition has a slight ambiguity in the FOLD/UNFOLD-operator pair that we clarified below. The original SchemaSQL proposal can be supported as well, with slight changes in the update propagation scheme.

Examples for the four operators defined in this section can be found in Fig. 2. We will refer to the input relation of each operator as R and to the output relation as Q.

The Unite-Operator is defined on a set of k relations $R^* = \{R_1, \ldots, R_k\}$ with attribute name a_p as an argument. The operator assumes input relations with identical schema and has as output one new relation Q. Q is constructed by taking the union of all input relations and adding a new attribute A_p whose values are the *relation names* of the input relations. In Fig. 2, the UNITE-operator is defined over the set of relations BA, LH and has the attribute name Airline as its argument.

The Fold-Operator works on a relation R in which a set of attributes must have the same domain. We denote the set of names of these attributes as $A^* = \{a_1, \ldots, a_n\}$. The operator takes as arguments the *names* of the pivot and data attributes a_p and a_d in its output relation. Furthermore, we require the attribute set A^* to satisfy a uniqueness constraint in order to avoid ambiguities in the operator (this requirement is not explicit in [12]).

BA

Destination	Business	Economy
Paris	1200	600
London	1100	475

LH

Destination	Business	Economy
Paris	1220	700

⇓

UNITE Airline

⇓

TMP_REL_0001

Airline	Destination	Business	Economy
BA	Paris	1200	600
BA	London	1100	475
LH	Paris	1220	700

⇓

FOLD Type, Price,{Business,Economy}

⇓

TMP_REL_0002

Airline	Type	Destination	Price
BA	Business	Paris	1200
BA	Business	London	1100
BA	Economy	Paris	600
BA	Economy	London	475
LH	Business	Paris	1220
LH	Economy	Paris	700

LONDON

Type	BA	LH
Business	1100	*null*
Economy	475	500

PARIS

Type	BA	LH
Economy	600	700

⇑

SPLIT Destination

⇑

TMP_REL_0004

Type	Destination	BA	LH
Business	London	1100	*null*
Economy	Paris	600	700
Economy	London	475	500

⇑

UNFOLD Airline, Price

⇑

TMP_REL_0003

Airline	Type	Destination	Price
BA	Business	London	1100
BA	Economy	Paris	600
BA	Economy	London	475
LH	Economy	Paris	700

STANDARD-SQL-Operator

⟹

```
select *
from tmp_rel_0002
where price <= 1100;
```

⟹

Fig. 2. An Example Using All Four *SchemaSQL* Operators UNITE, FOLD, UNFOLD, SPLIT.

The operator then takes all data values from the set A^* of related attributes, and sorts them into *one* new attribute a_d, introducing another new attribute a_p that holds the former attribute names. To motivate the above uniqueness constraint, note that its violation would require us to introduce multiple tuples in the output relation that differ only in their attribute a_d. The semantics of such tuples are not clear in a real-world application.

In Fig. 2, the FOLD-operator is defined on relation TMP_REL_0001 and has the arguments $a_p = \mathsf{Type}, a_d = \mathsf{Price}, A^* = \{\mathsf{Business}, \mathsf{Economy}\}$.

The Unfold-Operator is the inverse of FOLD. It is defined on a relation R and takes two attribute names a_p, a_d from R as arguments. We call A_p the *pivot attribute* and A_d the *data attribute*. We also define A^* as the set of distinct values in A_p.

The schema of Q then consists of all attributes in R except the data and pivot attribute, plus one attribute for each distinct data value in the pivot attribute. Each tuple t' in Q is constructed by taking a tuple t in R and filling each new attribute A_i with the value from attribute A_d in a tuple from R that has the *name* a_i as *value* in A_p (assuming an implicit conversion between names and values as required above). The new attributes all have the domain D_d of the old attribute A_d.

In Fig. 2, the UNFOLD-operator is defined over relation TMP_REL_0003 and takes as its arguments $a_p = \mathsf{Airline}$ and $a_d = \mathsf{Price}$. The operator produces output by taking tuples from TMP_REL_0003, and filling the attributes representing airlines with values from the data attribute Price in TMP_REL_0003, matching attribute names in the output relation with the values of the pivot attribute Airline in the input relation.

The Split-Operator is the inverse of the UNITE-operator. It takes as its argument the attribute name a_p. We define A^* as the set of unique values in A_p, similar to the UNFOLD-case. SPLIT then transforms a single relation R into a *set* of $k = |A_p|$ relations with the schema of R except for the pivot attribute A_p. We require that A_p does not have NULL-values. SPLIT then breaks relation R into k relations with the same schema, with the new relation names the k distinct values from R's attribute A_p.

In Fig. 2, the SPLIT-operator is defined over relation TMP_REL_0004, takes as its only argument $a_p = \mathsf{Destination}$, and produces 2 tables names LONDON and PARIS.

***SchemaSQL* Query Evaluation** Similar to traditional SQL evaluation, [12] proposes a strategy for *SchemaSQL* query evaluation that first constructs and then processes an algebra query tree, leading to an efficient implementation of *SchemaSQL* query evaluation over an SQL database system. In order to evaluate a *SchemaSQL* query, an algebra expression using standard relational algebra plus the four operators introduced above is constructed. This expression is of the following form [12]:

$$V = \textsc{Split}_a(\textsc{Unfold}_{b,c}(\pi_{\bar{d}}(\sigma_{cond}(\textsc{Fold}_{e_1,f_1,\bar{g_1}}(\textsc{Unite}_h(R_1)) \times \ldots \times \textsc{Fold}_{e_m,f_m,\bar{g_m}}(\textsc{Unite}_h(R_m)))))) \quad (1)$$

with attribute names a, b, c, e_i, f_i, h_i, the sets of attribute names $\bar{d}$ and $\bar{g}_i$, and selection predicates *cond* determined by the query. Any of the four *SchemaSQL* operators may not be needed for a particular query and would then be omitted from the expression. $R_1 \ldots R_m$ are base relations or, in the case that the expression contains a UNITE-operator, sets of relations with equal schema.

The algebraic expression for our running example (Fig. 1) is:

$$V = \textsc{Split}_{\mathsf{Destination}}(\textsc{Unfold}_{\mathsf{Airline,Price}}(\sigma_{\mathsf{Price}<1100}(\textsc{Fold}_{\mathsf{Type,\ Price,\ \{Business,Economy\}}}(\textsc{Unite}_{\mathsf{Airline}}(\mathsf{BA,LH}))))) \quad (2)$$

This algebraic expression is then used to construct an algebra tree whose nodes are any of the four *SchemaSQL* operators or a "Standard-SQL"-operator (including the π, σ, and $\times$-operators of the algebra expression) with standard relations "traveling" along its edges. The query is then evaluated by traversing the algebra tree and executing a query processing strategy for each operator, analogous to traditional SQL query evaluation.

3 The *SchemaSQL* Update Propagation Strategy

3.1 Classes of Updates and Transformations

The updates that can be propagated through *SchemaSQL* views can be grouped into two categories: *Schema Changes (SC)* and *Data Updates (DU)*. Schema changes are: *add-relation*(n, S), *delete-relation*(n), *rename-relation*(n, n') with relation names n, n' and schema S and *add-attribute*(r, a), *delete-attribute*(r, a), *rename-attribute*(r, a, a') with r the name of the relation R that the attribute named a belongs to, a' the new attribute name in the rename-case, and the notation otherwise as above. Data updates are any changes affecting a tuple (and not the schema of the relation), i.e., *add-tuple(r,t), delete-tuple(r,t), update-tuple(r,t,t'))*, with t and t' tuples in relation R with name r. Note that we consider *update-tuple* as a basic update type, instead of breaking it down into a *delete-tuple* and an *add-tuple*. An *update-tuple* update consists of two tuples, one representing an existing tuple in R and the other representing the values of that tuple after the update. This allows to keep relational integrity constraints valid that would otherwise be violated temporarily.

3.2 *SchemaSQL* Update Propagation vs. Relational View Maintenance

Update propagation in *SchemaSQL*-views, as in any other view environment, consists in recording updates that occur in the input data and translating them into updates to the view extent. In incremental view maintenance for SQL [16, 5], many update propagation mechanisms have been proposed. Their common feature is that the new view extent is obtained by first computing *extent differences* between the old view V and the new view V' and then adding them to or subtracting them from the view, i.e., $V' = (V \backslash \nabla V) \cup \Delta V$, with ∇V denoting some

set of tuples computed from the base relations that needs to be deleted from the view and ΔV some set that needs to be added to the view [16].

In *SchemaSQL*, this mechanism leads to difficulties. If *SchemaSQL* views must propagate both schema *and* data updates, the schema of ΔV or ∇V does not necessarily agree with the schema of the output relation V. But even when considering only data updates to the base relations, the new view V' may have a different schema than V. That means the concept of set difference between the tuples of V' and V is not even meaningful. Thus, we must find a way to incorporate the concept of schema changes. For this purpose, we now introduce a data structure ∂ which represents a sequence of n data updates DU and schema changes SC.

Definition 1 (defined update). *Assume two sets DU and SC which represent all possible data updates and schema changes, respectively. A change $c \in DU \cup SC$ is* ***defined (or valid) on a given relation*** *R if one of the following conditions holds:*

- *if $c \in DU$, the schema of the tuple added or deleted must be equal to the schema of R.*
- *if $c \in SC$, the object c is applied to (an attribute or relation) must exist (for delete- and update-changes) or must not exist (for add-changes) in R.*

Definition 2 (valid update sequence). *A sequence of updates $(c_1, \ldots, c_n)$ with $c_i \in DU \cup SC$, denoted by ∂R, is called* ***valid for*** *R if for all i $(1 < i \leq n)$, c_i is defined on the relation $R^{(i-1)}$ obtained by applying $c_1, \ldots, c_{i-1}$ to R.*

For simplicity, we will also use the notation $\partial\omega$ to refer to a valid update sequence to the output table of an algebra operator ω. Note that these definitions naturally extend to views, since views can also be seen as relational schemas. For an example, consider propagation of update `add-tuple('Berlin',1400,610) to` LH in Fig. 4 (p. 11). Having the value Berlin in the update tuple will lead to the addition of a new relation BERLIN in the output schema of the view—forming a sequence ∂V which contains both a schema change and a data update:

$$\partial V = (\textit{add-relation}(\mathsf{BERLIN}, (\mathsf{Type}, \mathsf{Destination}, \mathsf{BA}, \mathsf{LH})),$$
$$\textit{add-tuple}(\mathsf{BERLIN}, (\text{'Economy'}, \mathsf{null}, 610)))$$

The *add-relation*-update is valid since the relation BERLIN did not exist in the output schema before, and the *add-tuple*-update is valid since its schema agrees with the schema of relation BERLIN defined by the previous update.

3.3 Overall Propagation Strategy

Given an *update sequence* implemented by a `List` data structure, our update propagation strategy works according to the algorithm in Fig. 3. Each node in the algebra tree has knowledge about the operator it represents. This operator is able to accept *one* input update and will generate a sequence of updates as

```
function propagateUpdate(Node n, Update u)
    List r ← ∅, s ← ∅
    if (n is leaf)
        if (n.operator is affected by u)
            r.append(n.operator.operatorPropagate(u))
    else
        for(all children c_i of n)
            /* s will change exactly once, see text */
            s.append(propagateUpdate(c_i, u))
        for(all updates u_i in s)
            r.append(n.operator.operatorPropagate(u_i))
    return r
```

Fig. 3. The *SchemaSQL* View Maintenance Algorithm

output. Each (leaf node) operator can also recognize whether it is affected by an update (by comparing the relation(s) on which the update is defined with its own input relation(s)). If it is not affected, it simply returns an empty update sequence.

After all the updates for the children of a node n are computed and collected in a list (variable s in the algorithm in Fig. 3), they are propagated one-by-one through n. Each output update generated by the operator of n when processing an input update will be placed into one update sequence, all of which are concatenated into the final return sequence r (see Fig. 3, $\leftarrow$ is the assignment operator).

The algorithm performs a postorder traversal of the algebra tree. This ensures that each operator processes input updates after all its children have already computed their output[1]. At each node n, an incoming update is translated into an output sequence ∂n of length greater than or equal to 0 which is then propagated to n's parent node. Since the algebra tree is connected and cycle-free (not considering joins of relations with themselves) all nodes will be visited exactly once. Also note that since updates occur only in one leaf at a time, only exactly one child of any node will have a non-empty update sequence to be propagated. That is, the first **for**-loop will find a non-empty addition to s only once per function call. After all nodes have been visited, the output of the algorithm will be an update sequence ∂V to the view V that we will prove to have an effect on V equivalent to recomputation.

3.4 Propagation of Updates through Individual *SchemaSQL* Operators

Since update propagation in our algorithm occurs at each operator in the algebra tree, we have to design a propagation strategy for each type of operator.

[1] We are not considering concurrent updates in this paper.

Propagation of Schema Changes through SQL Algebra Operators The propagation of updates through standard SQL algebra nodes is simple. Deriving the update propagation for data updates is discussed in the literature on view maintenance [16, 5]. It remains to define update propagation for selection, projection, and cross-product operators under schema changes, as these are the only operators necessary for the types of queries discussed in this paper. In short, *delete-relation*-updates will make the output invalid, while other *relation*-updates do not affect the output. *Attribute*-updates are propagated by appropriate changes of update parameters or ignored if they do not affect the output. For example, a change *delete-attribute*(r, a) would not be propagated through a projection operator $\pi_{\bar{A}}$ if $a \notin \bar{A}$, and would be propagated as *delete-attribute*(q, a) otherwise, with q the name of the output relation of $\pi_{\bar{A}}$. We refer to our technical report [8] for further details, as they are not important for the comprehension of this paper.

***SchemaSQL* Operators** In the appendix (Figs. 5–8), we give the update propagation tables for the four *SchemaSQL* operators. In order to avoid repetitions in the notation, the cases for each update type are to be read in an "if-else"-manner, i.e., the first case that matches a given update will be used for the update generation (and no other). Also, NULL-values are like other data values, except where stated otherwise.

Inspection of the update propagation tables shows several properties of our algorithm. For example, the view becomes invalid under some schema changes or data updates, mainly if an attribute or relation that was necessary to determine the output schema of the operator is deleted (e.g., when deleting the pivot or data attribute in UNFOLD). In the case of *rename*-schema changes (e.g., under *rename-relation* in FOLD), some operators change their parameters. Those are simple renames that do not affect operators otherwise. The operator will then produce a zero-element output sequence. In those cases we denote renaming by $\Rightarrow$.

3.5 Update Propagation Example

Continuing our running example, Fig. 4 gives an example for an update that is propagated through the *SchemaSQL*-algebra-tree in Fig. 2. All updates are computed by means of the propagation tables in the appendix.

The operators in Fig. 4 appear in boxes with their output attached below each box (SQL-statements according to our update tables in [8]). The actual tuples added by these SQL-statements are shown in tabular form. The sending of updates to another operator is denoted by double arrows ($\Uparrow$), while single arrows ($\uparrow$) symbolize the transformation of SQL-statements into updates. We are propagating an *add-tuple*-update to base relation LH. Algorithm *propagateUpdate* will perform a postorder tree traversal, i.e., process the deepest node (UNITE) first, and the root node (SPLIT) last. The operators are denoted by ω_1 through ω_5, in order of their processing. First, the UNITE operator propagates

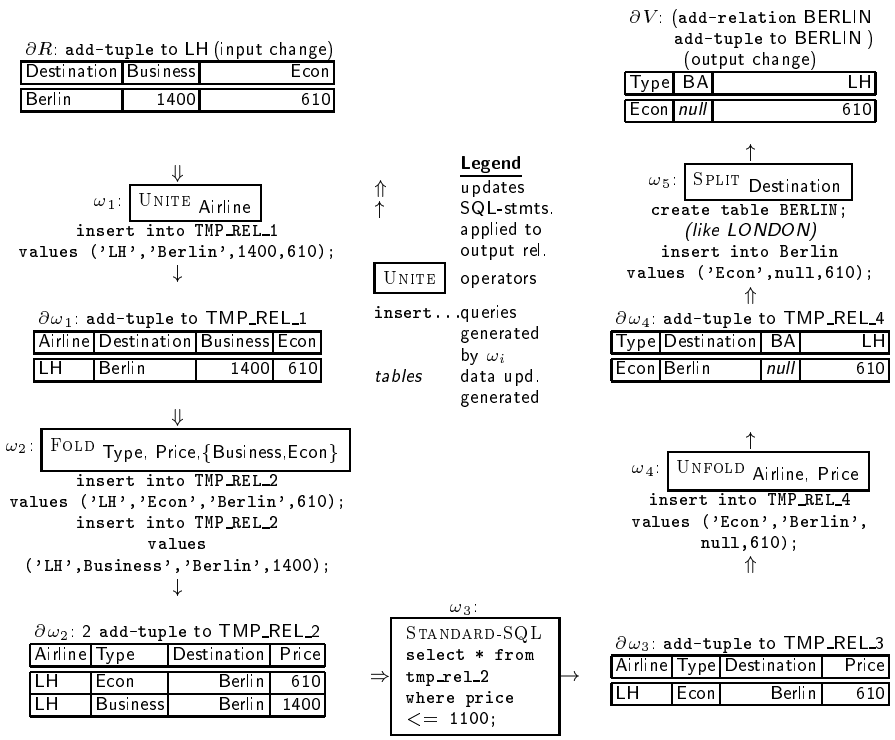

Fig. 4. Update Propagation in the View from Figure 2.

the incoming update into a one-element sequence $\partial\omega_1$ of updates which is then used as input to the FOLD-operator. The FOLD-operator propagates its input into a two-element sequence $\partial\omega_2$, sent to the StandardSQL-operator. This operator then propagates each of the two updates separately, creating two sequences $\partial\omega_{3_1}$ and $\partial\omega_{3_2}$, with 1 and 0 elements, respectively. Those sequences can simply be concatenated before the next operator's propagation is executed (Sec. 3.3), yielding $\partial\omega_3$. Since one update is not propagated due to the WHERE-condition in the StandardSQL-node, we have $\partial\omega_3 = \partial\omega_{3_1}$. UNFOLD now transforms its incoming one-element update sequence $\partial\omega_3$ into another one-element sequence $\partial\omega_4$ which becomes the input for the SPLIT-operator. This operator then creates the two-element final update sequence ∂V, consisting of an *add-relation* schema change followed by an *add-tuple* data update.

4 Correctness

Our update propagation strategy is equivalent to a stepwise evaluation of the algebraic expression constructed for a query. Each operator transforms its input changes into a set of semantically equivalent output changes, eventually leading

to a set of changes that must be applied to the view to synchronize it with the base relation change.

The structure of the algebra tree for a view depends only on the query, not on the base data [12]. The only changes to operators under base relation updates are possible changes of parameters (schema element names) inside the operators. An algebra operator cannot disappear or appear as the result of a base update. However, the entire view query may be rendered invalid, for example under some *delete-relation*-updates.

Theorem 1 (Correctness of *SchemaSQL* View Maintenance). *Let V be a view defined over the set of base relations $R_1, \ldots, R_p$, and $\Delta R_u \in \{DU, SC\}$ an update applied to one relation R_u $(1 \leq u \leq p)$. Let R'_u be the relation R_u after the application of ΔR_u and V'_{REC} be the view after recomputation. Furthermore, let the* SchemaSQL *View Maintenance Algorithm as defined in Section 3.3 produce a change sequence ∂V that transforms view V into view V'_{INC}. Then, $V'_{\mathsf{REC}} = V'_{\mathsf{INC}}$.*

Proof. (Sketch) We only give the proof idea, the full proof can be found in [8]. We prove by first showing that each operator by itself propagates updates correctly, i.e., produces results equivalent to recomputation. We then prove overall correctness by induction over the unique path in the algebra tree from the algebra node (leaf) in which the update occured to the root of the tree. □

5 Related Work

The integration of data stored in heterogeneous schemas has long been an object of intensive studies. The problem of schematic heterogeneity or different source capabilities is repeatedly encountered when attempting to integrate data. Some more recent examples are Garlic [17] and TSIMMIS [3]. Several logic-based languages have been developed to integrate heterogeneous data sources (e.g., SchemaLog [4]). Some SQL-extensions have also been proposed, in particular, *SchemaSQL* [11] (see below).

Those approaches overcome different classes of schematic heterogeneities. However, the important class of schematic heterogeneities in semantically equivalent relational databases is often excluded from integration language proposals, and even if it is covered, the problem of incremental view maintenance in a view over an integrated schema is rarely discussed. Krishnamurthy et al. [9] were the first to recognize the importance of schematic discrepancies and developed a logic-based language called IDL to deal with such problems. Miller et al. [13] show that relational databases may contain equivalent information in different schemas and give a formal model (Schema Intension Graphs) to study such "semantic equivalence" of heterogeneous schemas. An overview over object-oriented approaches can be found in [15]. Pitoura *et al.* also discuss a number of OODBMS implementation that support views. However, none of the projects listed has a comprehensive incremental view maintenance strategy.

An important approach at integrating semantically equivalent schemas has been done by Gyssens et al. [6] and later by Lakshmanan, Sadri, and Subramanian [11, 12]. In [11], the authors present *SchemaSQL*, which is used as the

basis for our work. *SchemaSQL* builds upon earlier work in SchemaLog [4]. It is a direct extension of SQL, with the added capability of querying and restructuring not only data, but also schema in relational databases, and transforming data into schema and vice-versa. Thus, using *SchemaSQL* as a query language makes it possible to overcome schematic heterogeneities between relational data sources.

A second foundation of our work is the large body of work on incremental view maintenance. Many algorithms for efficient and correct view maintenance for SQL-type queries have been proposed. One result, taking concurrency into account, is SWEEP [1]. Most of those approaches follow an *algorithmic* approach in that they propose algorithms to compute changes to a view.

Related to our work are also performance studies on incremental view maintenance algorithms. An early paper on measuring the performance of incremental view maintenance strategies is Hanson [7]. More recently, there are performance studies on some OO view maintenance algorithms for example by Kuno et al. [10].

6 Conclusions

In this paper, we have proposed the first incremental view maintenance algorithm for schema-restructuring views. We have shown that the traditional approach at incremental view maintenance—rewriting view queries and executing them against the source data—is not easy to adapt for such views, and that in addition it is necessary to include schema changes into the picture. We have solved this problem by defining an algebra-based update propagation scheme in which updates are propagated from the leaves to the root in the algebra tree corresponding to the query. We have also proved the correctness of the algorithm.

The update propagation strategy described in this paper has been implemented in Java on top of a *SchemaSQL* query evaluation module also written by us [8]. The algebra tree builder was constructed along the lines of [12]. Our experiments showed that for most queries and schemas, incremental maintenance performs significantly better than recomputation. A case in which incremental maintenance does not outperform recomputation occurs when a base update such as delete-relation is translated into a long sequence of single-tuple updates by one of the *SchemaSQL*-operators (up to one update per tuple in the deleted base relation). We plan to address this issue by introducing *update batches* as a new class of updates in addition to individual data updates and schema changes.

In summary, we believe our work is a significant step towards supporting the integration of large yet schematically heterogeneous data sources into integrated environments such as data warehouses or information gathering applications, while allowing for incremental propagation of updates. One application that comes to mind is in a larger data integration environment such as EVE [14], in which the *SchemaSQL* wrapper would help to integrate a new class of information sources into a view.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[2] W. Chen, M. Kifer, and D. Warren. Hilog as a platform for database languages. *IEEE Data Eng. Bull.*, 12(3):37, September 1989.

[3] H. Garcia Molina, J. Hammer, K. Ireland, et al. Integrating and accessing heterogeneous information sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[4] F. Gingras, L. Lakshmanan, I. N. Subramanian, et al. Languages for multidatabase interoperability. In Joan M. Peckman, editor, *Proceedings of SIGMOD*, pages 536–538, 1997.

[5] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[6] M. Gyssens, L. V. S. Lakshmanan, and I. N. Subramanian. Tables as a paradigm for querying and restructuring (extended abstract). In ACM, editor, *Proceedings of ACM Symposium on Principles of Database Systems*, volume 15, pages 93–103, New York, NY 10036, USA, 1996. ACM Press.

[7] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of SIGMOD*, pages 440–453, 1987.

[8] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. Technical Report WPI-CS-TR-00-25, Worcester Polytechnic Institute, Dept. of Computer Science, January 2001. `http://www.cs.wpi.edu/Resources/techreports`.

[9] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):40–49, June 1991.

[10] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in *MultiView*: Strategies and performance evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sept/Oct. 1998.

[11] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In T. M. Vijayaraman et al., editors, *International Conference on Very Large Data Bases*, pages 239–250, Mumbai, India, Sept. 1996.

[12] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *International Conference on Very Large Data Bases*, pages 471–482, 1999.

[13] R. J. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *International Conference on Very Large Data Bases*, pages 120–133, Dublin, Ireland, August 1993.

[14] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *EDBT'98*, pages 359–373, 1998.

[15] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object orientation in multidatabase systems. *ACM Computing Surveys*, 27(2):141–195, June 1995.

[16] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(3):337–341, September 1991.

[17] M. Tork Roth, M. Arya, L. M. Haas, et al. The Garlic project. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):557 ff., 1996.

Input Change	Conditions	Propagation
add-tuple (r, t)	$t[a_1, \ldots, a_n, a_p] \in R$	invalid view (key violation)
	$t[a_p] \in A^*$ $t[a_1, \ldots, a_n] \in R$	`update Q set [`$t[a_p]$`] =` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_p] \in A^*$ $t[a_1, \ldots, a_n] \notin R$	`insert into Q (`$a_1, \ldots, a_n, a_p$`) values (`$a_1, \ldots, a_n, a_d$`)`
	$t[a_p] \notin A^*$ $t[a_1, \ldots, a_n] \in R$	*add-attribute(q,*$t[a_p]$*)*, `update Q set [`$t[a_p]$`] =` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_p] \notin A^*$ $t[a_1, \ldots, a_n] \notin R$	*add-attribute(q,*$t[a_p]$*)*, `insert into Q (`$a_1, \ldots, a_n, a_p$`) values (`$a_1, \ldots, a_n, a_d$`)`
delete-tuple (r, t)	$t[a_p]$ exists in $R[a_p]$ exactly once	*delete-attribute(q,*$t[a_p]$*)*
	$t[a_p]$ exists in $R[a_p]$ more than once	`update Q set [`$t[a_p]$`] = NULL where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ [2]
update-tuple (r, t, t')	$t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$	`update Q set [`$t[a_p]$`] =` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$	break down into (*delete-tuple*, *add-tuple*)
add-attribute(r, a)		*add-attribute*(q, a)
delete-attribute(r, a)	$a \in \{A_d, A_p\}$	invalid view
	$a \notin \{A_d, A_p\}$	*delete-attribute*(q, a)
rename-attribute(r, a, a')	$a = A_d$	$\textsc{Unfold}_{a_p,a} \Longrightarrow \textsc{Unfold}_{a_p,a'}$
	$a = A_p$	$\textsc{Unfold}_{a,a_d}(R) \Longrightarrow \textsc{Unfold}_{a',a_d}(R)$
	$a \notin \{A_d, A_p\}$	*rename-attribute*(q, a, a')
delete-relation(r)		*delete-relation*(q)
rename-relation(n, n')		$\textsc{Unfold}_{a_p,a_d}(N) \Longrightarrow \textsc{Unfold}_{a_p,a_d}(N')$ (renaming the input relation)

[2] if this update leads to a tuple with all NULL-values, the tuple must be deleted.

Fig. 5. Propagation Rules for Q=$\textsc{Unfold}_{a_p,a_d}(R)$

Input Change	Conditions and Variable Binding	Propagation

add-tuple (r, t)	$(A^* = \{a_1^*, \ldots, a_k^*\}, k \leftarrow \lvert A^* \rvert)$	`for` i `:=` $1..k$
		`insert into Q`
		`values (`$a_1, \ldots, a_n$`,`a_i^*`,`$t[a_i^*]$`)`

delete-tuple (r, t)		`delete from Q` `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
update-tuple (r, t, t')	$A \in A^*$; set $t[a]$ to a value c	`update Q set` $a_d = c$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = a$
	$A \notin A^*$; set $t[a]$ from a value b to a value c	`update Q set` $a = c$ `where` $a = b$

add-attribute(r, a)	$A \in A^*$[3]	`foreach tuple` $u \in R$
		`insert into Q (`$a_1, \ldots, a_n, a_p, a_d$`)`
		`values (`$u[a_1, \ldots, a_n], a$`,NULL)`

	$A \notin A^*$	*add-attribute*(q, a)
delete-attribute(r, a)	$A \in A^*$	`delete from Q where` $a_p = a$
	$A \notin A^*$	*delete-attribute*(q, a)
rename-attribute(r, a, a')	$A \in A^*$	`update Q set` $a_p = a'$ `where` $a_p = a$
	$A \notin A^*$	*rename-attribute*(q, a, a')
delete-relation(r)		*delete-relation*(q)
rename-relation(n, n')		$\textsc{Fold}_{a_p, a_d}(N) \Longrightarrow \textsc{Fold}_{a_p, a_d}(N')$

[3]Note that the decision whether a *new* attribute should be a member of $a_1, \ldots, a_n$ can only be made by evaluating the view query.

Fig. 6. Propagation Rules for Q=$\textsc{Fold}_{a_p, a_d, A^*}(R)$

Input Change	Conditions	Propagation
add-tuple (r, t)	$t[a_p] \notin A^*$	`add-relation` $[t[a_p]]$ `with schema` $(S_R \backslash R.A_p)$; `insert into` $[t[a_p]]$ `values` $(t[a_1, \ldots, a_n])$
	$t[a_p] \in A^*$	`insert into` $[t[a_p]]$ `values` $(t[a_1, \ldots, a_n])$
delete-tuple (r, t)	$t[a_p]$ exists in $R[a_p]$ exactly once	`delete-relation` $[t[a_p]]$
	$t[a_p]$ exists in $R[a_p]$ more than once	`delete from` $[t[a_p]]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
update-tuple (r, t, t')	$t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$	`update` $[t[a_p]]$ `set` $[a_d]$ `=` $t[a_d]$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$
	$t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$	break down into (*delete-tuple*,*add-tuple*)
add-attribute(r, a)		$\forall q \in \{q_1 \ldots q_n\}$: *add-attribute*(q, a)
delete-attribute(r, a)	$a = A_p$	invalid view
	$a \neq A_p$	$\forall q \in \{q_1 \ldots q_n\}$: *delete-attribute*(q, a)[4]
rename-attribute(r, a, a')	$a = A_p$	$\textsc{Split}_a(R) \Longrightarrow \textsc{Split}_{a'}(R)$
	$a \neq A_p$	$\forall q \in \{q_1 \ldots q_n\}$: *rename-attribute*(q, a, a')
delete-relation(r)		$\forall q \in \{q_1 \ldots q_n\}$: *delete-relation*(q)
rename-relation(n, n')		$\textsc{Split}_{a_p}(N) \Longrightarrow \textsc{Split}_{a_p}(N')$

[4] If this update leads to a tuple with all NULL-values in an output relation, the tuple must be deleted.

Fig. 7. Propagation Rules for Q=$\textsc{Split}_{a_p}(R)$

Input Change	Conditions and Variable Bindings	Propagation
add-tuple (r_x, t)		`insert into Q` $(a_1, \ldots, a_n, a_p)$ `values` $(t[a_1, \ldots, a_n], r_x)$

delete-tuple (r_x, t)		`delete from Q`
		`where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and`
		$a_p = r_x$

update-tuple (r_x, t, t')	$A = A_d$; set $t[a]$ to a value c	`update Q set` $a = c$ `where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = r_x$
	$A \neq A_d$; set $t[a]$ from a value b to a value c	`update Q set` $a = c$ `where` $a = b$ `and` $a_p = r_x$
add-attribute(r, a)	add simultaneously to all R_i	*add-attribute*(q, a)
	otherwise	invalid view
delete-attribute(r, a)	delete simultaneously from all R_i	*delete-attribute*(q, a)
	otherwise	invalid view
rename-attribute(r, a, a')	rename simultaneously in all R_i	*rename-attribute*(q, a, a')
	otherwise	invalid view
add-relation(r_x, S)		no change (until first *add-tuple* to R_x)
delete-relation(r_x)		`delete from Q where` $a_p = r_x$
rename-relation(n, n')		$\textsc{Unite}_{a_p}(\{R_1, \ldots, N, \ldots, R_n\}) \Longrightarrow$ $\textsc{Unite}_{a_p}(\{R_1, \ldots, N', \ldots, R_n\})$ `update Q set` $a_p = n'$ `where` $a_p = n$

Fig. 8. Propagation Rules for Q=$\textsc{Unite}_{a_p}(R_1, R_2, \ldots, R_n)$[5]

[5]Note that r_x is the name of Relation R_x, which is one of the n relations of equal schema that are united by the UNITE-operator.