Updates Through Views: A New Hope

Yannis Kotidis
AT&T Labs - Research
kotidis@research.att.com

Divesh Srivastava
AT&T Labs - Research
divesh@research.att.com

Yannis Velegrakis
AT&T Labs - Research
velgias@research.att.com

Abstract

Database views are extensively used to represent unmaterialized tables. Applications rarely distinguish between a materialized base table and a virtual view, thus, they may issue update requests on the views. Since views are virtual, update requests on them need to be translated to updates on the base tables. Existing literature has shown the difficulty of translating view updates in a side-effect free manner. To address this problem, we propose a novel approach for separating the data instance into a logical and a physical level. This separation allows us to achieve side-effect free translations of any kind of update on the view. Furthermore, deletes on a view can be translated without affecting the base tables. We describe the implementation of the framework and present our experimental results

1. Introduction

Views are an important asset of database management systems. They limit access to only the portions of the data that are relevant to an application. They achieve schema independence [14] since certain physical database schema changes can be handled by modifying the view query while keeping the logical view interface unchanged. Applications deal with views the same way they deal with base tables. In fact, applications are rarely aware of whether a relation they access is a base table or a view. Views are usually virtual. Their instance data is completely defined by applying the view query on the base tables. Due to this virtual nature, view updates must be translated to updates on the base tables in a way that the view state after the update is the same we would have gotten if the update had been applied to a materialized view instance. This translation is referred to as *updates through views* [2, 9, 10]. Figure 1 provides a graphical representation of the problem statement. The user specifies an update U, which may be an *insertion*, *deletion* or *update* on the view instance $V(I)$. The goal is to find an update W on the base tables (instance I) that results in a new instance $W(I)$, with a view $V(W(I))$ that implements the view update. The update is said to be implemented *without side-effects* in the view if $U(V(I))=V(W(I))$, that is, no other view tuple should be affected by the base tables modification apart from the one specified in the view update command, and no additional tuple should appear in the view after the base tables have been modified.

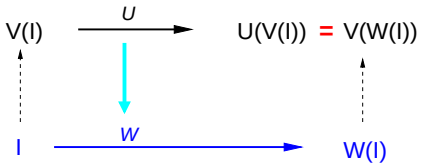

Figure 1. The view update problem

Unfortunately, existing literature on *updates through views* has shown that for many common cases there may be no side-effect free translation [8, 16, 10]. This led researchers to permit side-effects, to develop algorithms to detect them [5], or to restrict the kind of updates that can be performed on a view [10]. For many applications where views need to be handled the same way base tables are handled, accepting side-effects or having such restrictions is unacceptable. Consider, for instance, a view that joins a table `Person` and a table `Car` and a delete request for a person-car pair from the view. The deletion can be achieved either by removing the association between the person and the car through modification of the join attribute, or by deleting the car from the `Car` table, or by deleting the person from the `Person` table. If the car tuple joins with many persons or the person tuple with many cars, any of these three actions will have the side-effect of additionally deleting the tuples from the view that are formed by a join using the specific car or the person, respectively. Furthermore, the latter two actions make the additional assumption that the deletion of the car-person pair is due to the deletion of the car or the person. The semantics of the view deletion do not necessarily imply either one. We claim that the translation should make the minimum assumptions and should not make the car or the person disappear from the base tables `Car` or `Person`, respectively, neither should affect any other tuple in the view. If a car or a person are to be deleted from the database, they should be deleted through a delete command on the table `Car` or `Person`.

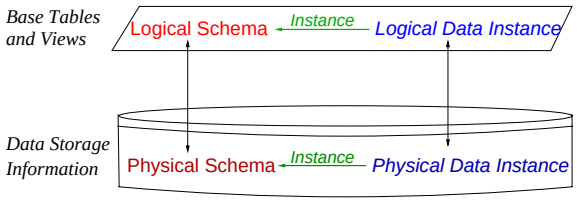

Figure 2. Abstraction levels in a DBMS

Due to the above results, in order to achieve side-effect free translations, we are forced to relax the requirement that the instance of a view is equal to the execution of its view query on the base tables. However, since the view is virtual, the view instance should be governed by the well-founded relationship between a view and its base tables. That is, every tuple that appears in the view must be justified by the appearance of certain tuples in the base tables, but the absence of a tuple from the view does not imply the absence of a tuple in the base tables. In other words, we require that for a view V with a view query Q_v, and an instance I, the instance of the view $V(I) \subseteq Q_v(I)$. This means that an insertion of a tuple in the view requires and may cause the insertion of certain tuples in its base tables. For the same reason, a deletion of a tuple from a base table will cause the deletion of every tuple in the view that is formed using the specific tuple. Since the tuple in a base table does not depend on the existence of any tuple in the view, a deletion of a view tuple should not modify the base table instances. Such modifications will lead to side-effects in the view.

In short, our goal in this work is to develop a mechanism that will allow every view update to be implemented: (1) without side-effects; (2) without affecting the instances of the base tables of the view in the case the update is a view deletion; and (3) without violating the well-founded relationship between the view and its base tables.

We advocate here that if we consider a clear separation between the physical and the logical level, for a user requested view update at the logical level (see Figure 2), one can *always* find an update at the physical level that performs the view update with no side-effects. The separation between the physical and the logical level is not new. In fact, it is one of the 12 rules of Codd [6] that characterize a relational database. Unfortunately, existing database systems do not provide true physical data independence, since every construct of the logical level corresponds directly to a primary physical structure. We have identified this lack of separation as one of the reasons for the difficulty of the view update problem. Following the example of the GMAP [17] system, but with a different goal in mind, we propose a physical representation of the data that is different from its logical representation. This separation does not affect the way users and applications interact with the views and the base tables, since all this interaction takes plase at the logical level.

Personnel

t_p:

dep	emp
CS	Fox
EE	Fox
Phil	Jones

Teaching

t_t:

prof	equip	sem
Fox	proj	PL
Fox	proj	OS
Fox	proj	DB
Fox	lp	OS
Jones	mic	DB

Schedule

t_s:

cour	rm	day
PL	10	Mon
DB	10	Tue
DB	23	Wed
DB	45	Fri

$\mathbf{V_b} = Personnel \bowtie_{emp=prof} Teaching \bowtie_{sem=cour} Schedule$

t_d:

dep	emp	prof	equip	sem	cour	rm	day
CS	Fox	Fox	proj	PL	PL	10	Mon
CS	Fox	Fox	proj	DB	DB	10	Tue
CS	Fox	Fox	proj	DB	DB	23	Wed
CS	Fox	Fox	proj	DB	DB	45	Fri
EE	Fox	Fox	proj	PL	PL	10	Mon
EE	Fox	Fox	proj	DB	DB	10	Tue
EE	Fox	Fox	proj	DB	DB	23	Wed
EE	Fox	Fox	proj	DB	DB	45	Fri
Phil	Jones	Jones	mic	DB	DB	10	Tue
Phil	Jones	Jones	mic	DB	DB	23	Wed
Phil	Jones	Jones	mic	DB	DB	45	Fri

Figure 3. Base tables and view instance

The current work makes the following contributions: (i) We provide a framework that allows views and base tables to be treated identically with respect to queries, insertions, deletions and updates; (ii) We achieve view updates with no side-effects by extending the relational model with identifiers on the values and using it as a physical data model; (iii) We guarantee that all the three previously mentioned desiderata for view updates are satisfied; (iv) We describe and implement the proposed framework using existing database technology, and we provide experimental results using our prototype on real and synthetic data.

This document is organized as follows. Section 2 presents motivational examples and explains the main idea of our solution. Section 3 formally describes our solution and Section 4 presents the theoretical properties of the proposed algorithms. Section 5 describes related work. Section 6 explains how our framework has been implemented over an existing relational database system, while Section 7 presents experimental results demonstrating the feasibility of the framework.

2. Motivation and Our Solution

To demonstrate the problem, let us consider the three relations `Personnel`, `Teaching` and `Schedule` of Figure 3. The first specifies for each employee (`emp`) the department (`dep`) where she is employed. The second describes seminars (`sem`) that are taught by professors (`prof`) and the teaching equipment (`equip`) they use. The third determines the room number (`rm`) and the day (`day`) the seminar course (`cour`) takes place. Assume a view V_b that joins these three tables as shown in Figure 3. We are deliberately using an example without primary/foreign key constraints for illustrative purposes. We will show what issues arise when we try to update the view. Assume that an application requests the

deletion of the shaded view tuple t_d from the view. Tuple t_d is formed by joining tuples t_p:[EE, Fox], t_t:[Fox, proj, DB] and t_s:[DB, 10, Tue] of relations Personnel, Teaching and Schedule, respectively. Deletion of tuple t_p or modification of its value Fox will achieve the deletion of tuple t_d from the view. However, since t_p also joins with three other tuples of table Teaching, this base table modification will have the unanticipated effect of removing three additional tuples from the view (the one above t_d, and the two below). Similar observations can be made for tuples t_t and t_s. In fact, there is no change that can be done on the base tables to make tuple t_d disappear from the logical instance of view V_b without causing side-effects in the view.

Assume for the moment that tuple t_p could join with only tuple t_t from Teaching. Then, deleting it, or changing its Fox value to null, would have achieved the requested view update. However, such a change implies that the reason for the deletion of tuple t_d is that Fox stops being affiliated with the EE department. If this was the case, then the delete request should have been issued on table Personnel and not on the view V_b. A query on Personnel or on any other base table of V_b, such as Personnel, Teaching or Schedule, should return the same result before and after the deletion of the view tuple t_d. Unfortunately, it can be shown that there is no update on the base tables that can achieve this desired result.

To tackle this issue, we propose a novel technique that considers values at the physical level as a pair of a display form and an identifier. We refer to these values as *id-values*, and we represent them by putting the identifier as a subscript to the display form. The values at the logical level, with which users and applications interact, remain unchanged. Two id-values can be used to form a join if they agree on their identifier. When an id-value is mapped to the logical level, only its display form appears. This allows us to have different id-values that appear the same at the logical level but have different identifiers and therefor participate in different joins. We will also call a *clone* of an id-value v another id-value v' that has the same display form but different identifier. A clone of a tuple is a duplication of the tuple where at least one of its attribute values is a clone of the respective attribute value of the original tuple.

We present first a naive approach that illustrates the technique but has large space complexity. Later on, we show how the same idea can be used in a much more sophisticated and space efficient way. Using id-values, we can have a physical representation of the tables which have instead of one, four different copies of tuple t_p, one for each view-tuple-forming join it participates. This idea is illustrated in Figure 4 where the identifiers are mentioned as subscripts next to the display form of each id-value. For certain id-values the identifier is not critical for the example and has been omitted for better readability. (For the moment, ignore the lower part of the figure and the fact that two tuples in Schedule are strike-through.) A few computations can verify that for these tables the instance of view V_b with the identifiers of the id-values suppressed, is exactly the same as the one in Figure 3.

Personnel

dep	emp
CS	Fox_1
CS	Fox_2
CS	Fox_3
CS	Fox_4
EE	Fox_5
EE	Fox_6
EE	Fox_7
EE	Fox_8
Phil	$Jones_9$
Phil	$Jones_{10}$
Phil	$Jones_{11}$
CS	Fox
EE	Fox

Teaching

prof	equip	sem
Fox_1	proj	PL_{21}
Fox_2	proj	DB_{22}
Fox_3	proj	DB_{23}
Fox_4	proj	DB_{24}
Fox_5	proj	PL_{25}
Fox_6	proj	DB_{26}
Fox_7	proj	DB_{27}
Fox_8	proj	DB_{28}
$Jones_9$	mic	DB_{29}
$Jones_{10}$	mic	DB_{30}
$Jones_{11}$	mic	DB_{31}
Fox	proj	OS
Fox	lp	OS

Schedule

cour	rm	day
PL_{21}	10	Mon
~~DB_{22}~~	~~10~~	~~Tue~~
DB_{23}	23	Wed
DB_{24}	45	Fri
PL_{25}	10	Mon
~~DB_{26}~~	~~10~~	~~Tue~~
DB_{27}	23	Wed
DB_{28}	45	Fri
~~DB_{29}~~	~~10~~	~~Tue~~
DB_{30}	23	Wed
DB_{31}	45	Fri

~~CS~~	~~Fox_{12}~~
~~EE~~	~~Fox_{13}~~
~~Phil~~	~~$Jones_{13}$~~

Fox_{12}	proj	DB_{32}
Fox_{13}	proj	DB_{33}
$Jones_{14}$	mic	DB_{34}

DB_{32}	10	Tue
DB_{33}	10	Tue
DB_{34}	10	Tue

Figure 4. Base tables at the physical level

Deletion of the view tuple t_d can now be done with no side-effects by deleting/modifying the dark gray shaded tuples. If we further wanted to delete view tuples [CS,Fox,Fox,proj,DB,DB,10,Tue] and [Phil,Jones,Jones,mic,DB,DB,10,Tue], we could remove the light gray shaded tuples. Unfortunately, after the last two deletions, tuple [DB,10,Tue] will disappear from the instance of the base table Schedule. To avoid this and to satisfy the second view update desiderata of Section 1, we clone every shaded tuple and set new identifiers to the join attribute id-values in the clone. Subsequently, the clone of the shaded tuple of the first table (Personnel) and the shaded tuple of the last table (Schedule) are removed. That way, the deletion of the view tuples is achieved but the instances of the base tables remain unchanged. The lower part of Figure 4 illustrates the new tuples that were inserted in the previous step. The strike-through tuples of the figure are the tuples that were removed.

In that new instance, query select * from Teaching on the instance of Figure 4 will return tuple [Fox,proj,DB] multiple times while on the instance of Figure 3, it would have returned it only once. This kind of duplication can be easily removed from query results by adding an extra attribute at the physical level that determines whether two tuples are clones of the same original tuple.

The presented technique, referred to as *eager*, has the drawback of requiring too many tuples at the physical schema. For example, there is no reason to have three different copies of tuple [Phil,Jones] since it is not anyway affected by our intended delete of view tuple t_d. An improved technique, called *on-demand*, makes copies of tuples only when needed. Consider again the example of deleting the view tuple t_d. Starting from table Personnel, it is noticed

Personnel

dep	emp
CS	Fox
~~EE~~	~~Fox~~
Phil	Jones
EE	$\texttt{Fox}_p$
~~EE~~	~~$\texttt{Fox}_d$~~

Teaching

prof	equip	sem
Fox	proj	PL
Fox	proj	OS
Fox	proj	DB
Fox	lp	OS
Jones	mic	DB
$\texttt{Fox}_p$	proj	PL
$\texttt{Fox}_p$	proj	OS
~~$\texttt{Fox}_d$~~	~~proj~~	~~DB~~
$\texttt{Fox}_p$	lp	OS
$\texttt{Fox}_p$	proj	$\texttt{DB}_p$
~~$\texttt{Fox}_d$~~	~~proj~~	~~$\texttt{DB}_d$~~

Schedule

cour	rm	day
PL	10	Mon
DB	10	Tue
DB	23	Wed
DB	45	Fri
~~$\texttt{DB}_d$~~	~~10~~	~~Tue~~
$\texttt{DB}_p$	23	Wed
$\texttt{DB}_p$	45	Fri

Figure 5. Modified base tables

that t_d is formed by a join through tuple t_p:[EE, Fox]. That tuple is replaced by tuples [EE, $\texttt{Fox}_d$] and [EE, $\texttt{Fox}_p$], called the *delete* and the *preserve* tuple, respectively. The goal is to make every join in which tuple t_p participates to use [EE, $\texttt{Fox}_p$] unless the result of the join is the view tuple t_d in which case the join should use [EE, $\texttt{Fox}_d$]. All the joins that were formed between [EE, Fox] and a tuple in Teaching with id-value Fox in the join attribute prof cannot be formed after the last replacement. Their Fox id-value cannot be changed to $\texttt{Fox}_p$ or $\texttt{Fox}_d$ because this will make them unable to join with other tuples of Personnel such as [CS, Fox]. Instead, they are cloned and in the clone, the id-value Fox is replaced by $\texttt{Fox}_p$, unless the tuple that is cloned is tuple t_t, in which case it is replaced by $\texttt{Fox}_d$. These new tuples are indicated in Figure 5 with the light gray color. A similar process is repeated, this time for tuple [$\texttt{Fox}_d$, proj, DB] which is replaced by [$\texttt{Fox}_p$, proj, $\texttt{DB}_p$] and [$\texttt{Fox}_d$, proj, $\texttt{DB}_d$]. Notice that the copy with id-value $\texttt{DB}_p$ has id-value $\texttt{Fox}_p$ instead of $\texttt{Fox}_d$.[1] Finally, in relation Schedule, every tuple with id-value DB is cloned and in each clone, the id-value DB is replaced by $\texttt{DB}_p$, unless the tuple that is cloned is the t_s in which case it is replaced by $\texttt{DB}_d$. The tuples generated in this second step are illustrated with the dark gray color in Figure 5. In the resulting instance, shown in Figure 5, deletion of the view tuple t_d can be achieved by deleting the double strike-through tuples. Such a deletion satisfies the view update translation requirements of Section 1. Furthermore, it has no redundancy and is minimal, in the sense that removal of any other tuple will result in unanticipated changes in the logical view instance.

Notice that due to the replacement of tuple t_p, four new tuples were inserted in table Teaching, where four was the cardinality of the id-value Fox in the join attribute prof. Similarly, three tuples were inserted in table Schedule, where three was the cardinality of id-value DB in the join attribute cour. On the other hand, id-value DB has cardinality two in attribute sem and id-value Fox has cardinality two in attribute emp. Thus, had the same process started first from table Schedule, then proceeded to table Teaching and finally to table Personnel, the final instance would have fewer tuples. This means that the order in which the relations are processed is critical for the on-demand technique. This is not the case for the eager technique.

The requirement for view updates with no view side-effects extends to insertions as well. Assume that the base tables are as shown in Figure 3 and tuple [ME, Fox, Fox, mic, PL, PL, 12, Fri] needs to be inserted in view V_b. This insertion requires a tuple [ME, Fox] be inserted in table Personnel, [Fox, mic, PL] in Teaching and [PL, 12, Fri] in Schedule. Naturally, tuple [ME, Fox] will join not only with [Fox, mic, PL], but with all the tuples in Teaching having id-value Fox in attribute prof, causing additional tuples to appear in view V_b. To avoid this, the new tuples are inserted in the physical base tables with new id-values. For example, tuple [ME, $\texttt{Fox}_n$] is inserted in Personnel, [$\texttt{Fox}_n$, mic, $\texttt{PL}_{n'}$] in Teaching and [$\texttt{PL}_{n'}$, 12, Fri] in Schedule. Their join will create the logical view tuple [ME, Fox, Fox, mic, PL, PL, 12, Fri], but no other, since due to the new identifiers they have, they will join with none of the other existing tuples.

3 Supporting View Updates

This section provides the algorithms for updating the physical instance in order to implement a view update request issued at the logical level, in a way that satisfies the three desiderata set in Section 1. Updates are declarative statements of the form insert into V values (...), delete from V where <conditions> or update V set attr_1=expr_1, attr_2=expr_2,... where <conditions>. The current study is restricted to a single view. Generalization of the results for the case where there are multiple views, or views that are defined at any time during lifetime of the database system is out of the scope of this work and is part of future research.

3.1. Handling Insert Statements

For an insertion of a new tuple t_v in a view, the right tuples are created in the base tables so that their join is tuple t_v. In particular, a new tuple t_R is created for every relation R that appears in the from clause of the view query. If an attribute A of a relation R is used in the select clause of the view query, then a new id-value v_o is created for the attribute A of the tuple t_R. The identifier o of that id-value differs from any other identifier of an id-value in the domain of A that is already in the database. The display form v is the one specified in the insert statement for the attribute A. Finally, for every two or more attributes that the where clause of the view query specifies or logically implies to be equal, e.g., the join attributes, their identifiers are set the same. This ensures that the new tuple t_R forms the right joins that generate tuple t_v.

If the values in the insert statement violate the conditions

[1] For simplicity of notation, we have used subscript d and p for both Fox and DB, but it should be understood that they refer to distinct identifiers.

of the view query, following the approach of Keller [10] and Dayal and Bernstein [9], the insertion statement is rejected.

Example 3.1 *Consider the base tables of Figure 3 and view V with view query:* select * from Personnel, Teaching where emp=prof. *If tuple [*CS, Berry, Berry, PC, HW*] is to be inserted in the view, tuples [*$\texttt{CS}_{n_1}$, $\texttt{Berry}_{n_2}$*] and [*$\texttt{Berry}_{n_2}$, $\texttt{PC}_{n_3}$, $\texttt{HW}_{n_4}$*] will be created in the relations* Personnel *and* Teaching*, respectively. The identifiers n_i are all new identifiers that do not exist in the database. Notice how the join between the two tuples is preserved by having the id-value in the attribute* emp *and* prof *use the same identifier n_2. If instead, tuple [*CS, John, Berry, PC, HW*] was to be inserted in the same view, the statement would have been rejected since it violates the condition that attributes* emp *and* prof *should be equal.*

In a sense, the conditions set by the view query are handled as constraints on the view instance.

When the view query projects out certain attributes, one may need to introduce id-values with null display forms on the projected-out attributes whose value cannot be inferred from the join or the equality conditions in the view.

Example 3.2 *If the view in Example 3.1 did not have attributes* emp *and* equip *in its* select *clause, then insertion of tuple [*CS, Berry, HW*] in the view would have been translated to physical level insertions of tuple [*$\texttt{CS}_{n_1}$, $\texttt{Berry}_{n_2}$*] in* Personnel *and [*$\texttt{Berry}_{n_2}$, $\texttt{null}_{n_3}$, $\texttt{HW}_{n_4}$*] in* Teaching.

The situation is different if the insert command is for a base table instead of a view. If a tuple t_i is to be inserted in the logical schema relation R, then a new tuple t_n is inserted in the physical table R. Every attribute of t_n is an id-value v_o where identifier o is new and the display form v is the one specified in the respective attribute of the logical level tuple t_i. The expected behavior of tuple t_n is to join with all tuples of the other tables that agree on the respective join attributes. This cannot happen since the identifiers of the id-value in t_n are new. To cope with this issue, for every relation R' that joins with R through attributes A and B, respectively, every tuple with a display form in attribute A equal to the display form of attribute B in t_n, is cloned, and the identifier of the id-value of attribute A in the clone is set to be the same as the identifier of the id-value of attribute B in t_n. This way, the behavior expected by the insertion of tuple t_i in R is achieved.[2]

Example 3.3 *Consider the base tables of Figure 5, and assume that tuple [*Fox, mic, DB*] is to be inserted in relation* Teaching*. Let tuple [*$\texttt{Fox}_m$, $\texttt{mic}_p$, $\texttt{DB}_n$*] be its physical level representation. Notice that identifiers m, p and n do not exist in the instance of Figure 5. The tuple is expected to join with every tuple in relation* Personnel *with display form* Fox *on attribute* emp*. To ensure this, every tuple of* Personnel *with that property is cloned and in the clone, the identifier of the id-value* Fox *in* emp *is set to be m. Similar actions are performed for relation* Schedule.

The above steps are for the on-demand technique. For the eager technique, one extra step is needed to generate one clone of the newly inserted tuple for every join it participates in.

3.2. Handling Delete Statements

Deletions from regular tables are handled as usual. Of course a delete operation on a base table will have side-effects on any view defined over this table, but this is a natural consequence. When a view tuple is to be deleted, it is not clear which of the component tuples are to be modified. If the eager technique is used (e.g. Figure 4), deletions are handled by cloning the tuples that participate in the join forming the view tuple under deletion, as we did in Section 2. Deletions from a view are more challenging if the on-demand technique is to be used. There, we try to do the minimum number of changes in the base tables that achieve the view deletion without side-effects in the view and without affecting the instances of the base tables. We again consider views that involve natural joins: $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ where each table R_i refers to a base table. If a R_i is a view, it can be replaced by the view definition by applying query unfolding [12, 15]. Let $A_{i,j}$ denote the (common) join attribute of tables R_i and R_j. When R_i and R_j join on multiple attributes then $A_{i,j}$ refers to their composite attribute. Notice that the definition allows cyclic joins as well as self-joins, i.e., when R_i and R_j are the same relation. The process presented, here with some minor modifications, works even in these cases.

We first consider the case of a single tuple delete. Let t_d denote the single tuple that we would like to remove from view V. The algorithm (Algorithm 1) utilizes two main procedures: $processIn()$ and $processOut()$. At an abstract level, it will visit every table R_i and modify its contents. Let t_{d_i} be the tuple of table R_i that is used in the join forming the view tuple t_d. For each t_{d_i}, during $processIn()$ and $processOut()$, the algorithm generates a single special tuple, referred to as the *delete* tuple, and some special tuples, referred to as *preserve* tuples. The preserve and delete tuples are clones of existing base table tuples with different (new) identifiers at the join attributes. The delete tuples only join among themselves to form the view tuple t_d. Every other view tuple that was formed through a join using t_{d_i} keeps being formed through a join using the preserve tuples; thus any modification on the preserve tuples will have undesired effects on the view. Removal of the delete tuples, on the other hand, results in the deletion of only tuple t_d

[2] There is also an alternative technique that achieves the same result by modifying only relation R but the description is omitted for brevity.

Algorithm 1 Main Loop

Input: $G_e(V_e, E_e))$ {The join-execution graph}

1: **while** $\exists R_i \in V_e$: R_i is not visited $\wedge$
$\forall R_j \in predecessors(R_i)$: R_j is visited
do
2: Call $processIn(R_i)$ if $predecessors(R_i) \neq \emptyset$
3: Call $processOut(R_i)$ if $successors(R_i) \neq \emptyset$
4: Mark R_i visited
5: **end while**

from the view, i.e., achieves the desired deletion with no side-effects. The order in which the relations R_i are visited is based on the notion of the *join execution graph*.

Definition 3.4 *Given a view query Q_v, the join graph $G(V, E)$ is an undirected graph whose set of nodes is the relations in Q_v: $V = \{R_1, \ldots, R_n\}$ and set of edges $E = \{(R_i, R_j) | R_i$ joins with R_j through $A_{i,j}\}$. The join execution graph $G_e(V_e, E_e)$ is a connected DAG, obtained from join-graph $G(V, E)$, by considering the nodes in V_e to be those in V, making the edges in E directional and removing one or more of them to make the graph acyclic in the case of cyclic joins in the view query.*

Multiple join execution graphs can be obtained from a join graph. Intuitively, given a view query Q_v, each join execution graph represents a different "execution plan" for the on-demand view deletion algorithm that we will describe next. Each such plan correctly performs the delete command, i.e., satisfies the three requirements of Section 1. However, the final instance differs depending on the join execution graph that was used. For example, in Section 2 we saw that if we process the relations in the order `Schedule`, `Teaching`, `Personnel`, we generated fewer tuples than if we had done it in the order `Personnel`, `Teaching`, `Schedule`. We would like to be able to find and use the execution graph that minimizes the size of the final instance. For the moment, we assume that the join execution graph G_e is given and is `Personnel` $\rightarrow$ `Teaching` $\rightarrow$ `Schedule`.

Procedure processIn(R_i) is invoked for a relation R_i that is chosen to be processed when the set $predecessors(R_i)$ is not empty, i.e. node R_i has one or more incoming edges in G_e. It assures that changes made in adjacent nodes of R_i in G_e result in no tuples disappearing from the view. It consists of the following two steps.

Step I.1: Create special delete tuple A clone of t_{d_i} is inserted in R_i. The clone differs from t_{d_i} only on the id-value of the join attribute corresponding to an incoming edge (R_j, R_i). The new id-value is v_d, i.e., has the same display form as in t_{d_i} but a different identifier d. Application of this step on the tables of Figure 5 creates no tuple in `Personnel` (no incoming edge), tuple [`Fox`$_d$, `proj`, `DB`] in table `Teaching` and tuple [`DB`$_d$, `10`, `Tue`] in table `Schedule`.

Step I.2: Create join-preserve tuples for incoming edges If (R_{j_0}, R_i), (R_{j_1}, R_i),..., (R_{j_k}, R_i) are incoming edges of R_i in G_e, let t be a tuple in R_i that joins with tuples $t_{d_{j_0}}$, $t_{d_{j_1}}$,..., $t_{d_{j_k}}$ of relations R_{j_0}, R_{j_2}, ..., R_{j_k}, respectively. Tuple t is cloned in R_i exactly $2^k - 2$ times. (In case table R_i joins with multiple tables using the same join attribute, k refers to the number of join-attributes that have an incoming edge. In the join execution graph, no R can have both in-coming and out-going edges on the same attribute.) Let v^{j_l} be the id-value of join attribute $A_{j_l,i}$, $0 \leq j_l \leq k-1$. If we enumerate the copied tuples using index value h in range $1 \ldots 2^k - 2$, then the id-value of join attribute $A_{j_l,i}$ in the h^{th}-clone obtains a new id-value $v_p^{j_l}$ with special identifier p, if the bit in position j_l of the binary representation of h is 1. When t is not the tuple t_{d_i}, for any value of k, a clone of t is added in R_i. That clone has all join-attributes (for all incoming edges) having the special preserve identifier. When t is the tuple t_{d_i}, no action is performed (the process took place during Step I.1). Table `Teaching` in our join execution graph has only one incoming edge emanating from table `Personnel`; thus, in Figure 5 tuples [`Fox`, `proj`, `PL`], [`Fox`, `proj`, `OS`] and [`Fox`, `lp`, `OS`] are cloned and the respective tuples [`Fox`$_p$, `proj`, `PL`], [`Fox`$_p$, `proj`, `OS`] and [`Fox`$_p$, `lp`, `OS`] are introduced. Table `Schedule` also has one incoming edge from table `Teaching`. Application of Steps I.1 and I.2 results to its bottom three shadowed tuples.

Note that after $processIn$ has been executed for a relation, the tuple generated in Step I.1 will take the role of tuple t_{d_i}. For example, for table `Teaching` and for the procedure $processOut()$ that follows, the tuple t_{d_i} will be considered the tuple [`Fox`$_d$, `proj`, `DB`].

Procedure processOut(R_i) is invoked when the set $successors(R_i)$ is not empty, i.e. node R_i has one or more outgoing edges in G_e. It modifies R_i so that tuple t_{d_i} does not interfere with other joins apart from the one creating the view tuple t_d. It consists of the following three steps.

Step O.1: Create special delete tuple A clone of tuple t_{d_i} is inserted in R_i. In the clone, every join attribute $A_{i,j}$ for which there is outgoing edge (R_i, R_j) keeps the same display form but gets a new identifier d. Tuples [`EE`, `Fox`$_d$] of relation `Personnel` and [`Fox`$_d$, `proj`, `DB`$_d$] of relation `Teaching` in Figure 5 are created by cloning the t_{d_i} tuples [`EE`, `Fox`] and [`Fox`$_d$, `proj`, `DB`], respectively.

Step O.2: Create join-preserve tuples between R_i and adjacent nodes in G_e A clone of tuple t_{d_i} is inserted in R_i. In the clone, the join attribute $A_{i,j}$ for which there is outgoing edge (R_i, R_j) in G_e keeps the same display form but gets a new identifier p. The clone preserves all the view tuples which were formed through a join using t_{d_i} and which should remain in the view after the deletion of t_d. In Figure 5, tuples [`EE`, `Fox`$_p$] in table `Personnel` and [`Fox`$_p$, `proj`, `DB`$_p$] in table `Teaching` were created by cloning

tuples [EE, Fox] and [$\texttt{Fox}_d$, proj, DB] respectively, due to their join with tuples in relations Teaching and Schedule.

Step O.3: Remove original tuple Tuple t_{d_i} is removed from R_i. Single-strike-through tuples in Figure 5 are deleted during this step.

After Algorithm 1 has terminated, the special delete tuple that has been created in each table plays the role of t_{d_i} and can be removed without side-effects. In Figure 5, these tuples are the double-strike-through.

When the delete command is declarative, i.e., multiple view tuples are to be deleted, the processing is similar to the case of a single tuple delete. The main difference is that t_{d_i} refers to a bag of tuples. In that case, one special delete tuple is created (in $processOut()$ and $processIn()$) for the whole bag, instead of one for each of its tuples.

3.3. Handling Update Statements

Updates on regular tables are performed as in the relational model. They have, though, the same problem we had for insertions on regular tables when these tables were participating in view joins (see Section 3.1). We deal with this issue in the same way we did for the case of insertions.

An update on the view can be modeled as a deletion followed by an insertion. Although seeing the delete this way generates a correct translation of the view update, it generates more base table tuples than it is needed. So, instead, what we do is to issue a virtual delete followed by a base table value update. The virtual delete is similar to the delete described in Section 3.2. The only difference is that at the end the delete tuples (the double strike-through tuples in the Figures) are not removed. Recall that these tuples generate the view tuple that needs to be deleted/modified and only that. Hence, we issue an update that modifies the display forms of their id-values appropriately, and this update has no side-effect in the view.

Example 3.5 *Consider the update command* update V_b set prof='Nick' where dep='EE' and sem='DB' and rm=10, *which sets the* prof *attribute of tuple* t_d *of Figure 3 to value* Nick. *After the virtual delete, the instance will be like the one in Figure 5 but with the double strike-through tuples being kept in the base tables. The join of these tuples forms the view tuple* t_d *and nothing else; thus, they can be modified with no side effects in the view. The performed modification is to change id-value* $\texttt{Fox}_d$ *in* emp *to* $\texttt{Nick}_d$. *Note that the identifier remained the same. Only the display form changed. Due to this, the new id-value can still join with the unchanged id-value* $\texttt{Fox}_d$ *in* prof *generating the correct updated view tuple.*

Special care needs to be taken when the update statement uses functions or values obtained from other attributes to specify the value of an attribute instead of a constant.

Personnel

dep	emp
CS	Fox
~~EE~~	~~Fox~~
Phil	Jones
EE	Fox_p
EE	$10_{d_{10}}$
EE	$23_{d_{23}}$
EE	$45_{d_{45}}$

Teaching

prof	equip	sem
Fox	proj	PL
Fox	proj	OS
Fox	proj	DB
Fox	lp	OS
Jones	mic	DB
Fox_p	proj	PL
Fox_p	proj	OS
~~$\text{Fox}_{d_{10}}$~~	~~proj~~	~~DB~~
~~$\text{Fox}_{d_{23}}$~~	~~proj~~	~~DB~~
~~$\text{Fox}_{d_{45}}$~~	~~proj~~	~~DB~~
Fox_p	lp	OS
Fox_p	proj	DB_p
$\text{Fox}_{d_{10}}$	proj	$\text{DB}_{d_{10}}$
$\text{Fox}_{d_{23}}$	proj	$\text{DB}_{d_{23}}$
$\text{Fox}_{d_{45}}$	proj	$\text{DB}_{d_{45}}$

Schedule

cour	rm	day
PL	10	Mon
DB	10	Tue
DB	23	Wed
DB	45	Fri
$\text{DB}_{d_{10}}$	10	Tue
$\text{DB}_{d_{23}}$	23	Wed
$\text{DB}_{d_{45}}$	45	Fri

Figure 6. Update using an attribute value

In such cases, multiple clones of the delete tuple t_d are required to achieve the desired functionality. We give an example but we omit the full details on this issue due to lack of space.

Example 3.6 *Consider the update command* update V_b set emp=rm where dep='EE' and sem='DB' *which sets the* emp *attribute of tuple* t_d *in Figure 3 and its subsequent two tuples to the value of their* rm *attribute, which is* 10, 23 *and* 45, *respectively. The result table that our on-demand algorithm will give is indicated in Figure 6. The difference from Figure 5 is that the double strike-through tuples are not deleted but instead are replicated with identifiers* d_{10}, d_{23} *and* d_{45}.

If the update violates any of the conditions of the view, similarly to the case of insertions, the update statement is rejected.

The eager approach is similar but requires an additional step at the end that clones the inserted tuple as many times as the number of view generating joins it participates in.

4. Correctness, Completeness, Complexity

A translation of a view update to updates on the base tables that satisfies the conditions of Section 1 is referred to as a *correct translation.* Given a database instance I, a view V on I and a declarative update U on V, we can generate a correct translation as follows. We materialize view V. We apply the update to the materialized view. In case the update is a delete, the deleted tuple is cloned, as discussed in Section 2. We assign a different identifier to every id-value of every tuple, unless two id-values are join values in which case they are assigned the same id. The display forms remain the same. We now project these id-enhanced views on each base table. The result instance generates the updated view and every original base table.

Theorem 4.1 *For every declarative view update, there is always a correct translation.*

The procedure described above is the main concept on which the *eager* approach is based. However, for a given view update, one can find infinitely many correct instances, and the issue there is which one to choose. It is desirable to choose the one with the minimum number of required changes in the physical instance. This is what our *on-demand* approach is trying to achieve.

When the view update is an insert, the tuples inserted at the base tables join to form the inserted view tuple but generate no other. If one of them is not inserted in the respective base table, the view tuple will not appear in the view. If the update is a delete, given a join execution graph, as explained in Section 3.2, Algorithm 1 will introduce the minimum required tuples in the base tables to achieve the side-effect free view deletions. By considering all the possible join execution plans, the optimal one can be found. The optimal join execution graph is identified as the one that minimizes the cost function defined below. For the case where each relation joins with at most two other relations in the view query then, as will be explained, the optimal selection can be found in linear time without even enumerating all possible join execution graphs. Similar observations hold for the updates, since updates are handled similar to deletions.

To derive the cost of a delete, we first consider the case of a single view tuple deletion (e.g., example of Figure 3). To derive the cost we study separately processes $processOut()$ and $processIn()$ of Algorithm 1. Let $f_{in}(R_i)$ (resp. $f_{out}(R_i)$) be the number of join-attributes in table R_i with in-coming (resp. outgoing) edges in the join execution graph. Procedure $processOut(R_i)$ performs three tasks: (i) inserts a special delete tuple, (ii) clones all tuples matching t_{d_i} as many times as the fan out of node R_i and (iii) removes the original tuples:

Theorem 4.2 *The size of base table R_i during execution of $processOut(R_i)$ is increased by $Cost_{out}(R_i) = 1 + (f_{out}(R_i) - 1)|\sigma_{t_{d_i}}(R_i)|$, where $|\sigma_{t_{d_i}}(R_i)|$ is the multiplicity of tuple t_{d_i} in R_i*

As an example, for the instance in Figure 5 where the join execution graph is `Personnel` $\rightarrow$ `Teaching` $\rightarrow$ `Schedule`, the cost $Cost_{out}$(`Personnel`) is 1+(1-1)1=1.

For a tuple t in R_i, procedure $processIn(R_i)$ will clone t $2^{comm(t,t_{d_i})} - 1$ times, where $comm(t, t_{d_i})$ is the number of common join-attribute id-values between t and t_{d_i}. This also accounts for the special delete tuple that is created.

Theorem 4.3 *The size of base table R_i during execution of $processIn(R_i)$ is increased by:*

$$Cost_{in}(R_i) = \sum_{t \in R_i} 2^{comm(t,t_{d_i})} - 1$$

The above discussion suggests that, given the join graph G, we can enumerate all possible join execution graphs G_e and select the one that minimizes the sum

$$\sum_{i:successors(R_i)\neq\{\}} Cost_{out}(R_i) + \sum_{i:predecessors(R_i)\neq\{\}} Cost_{in}(R_i)$$

When each base relation joins with at most two other relations, the cost function is decomposable and this process can be expedited by considering the direction of each edge in G_e independently. Consider undirectional edge (R_i, R_j) in G and let (for ease of exposition) $A_{i,j}$ be the common join attribute. Let o be the identifier of the join attribute and n_i, n_j be the multiplicities of o in base tables R_i and R_j respectively. If the direction of the edge in G_e is $R_i \rightarrow R_j$ then the cumulative increase in the size of the base tables R_i and R_j is $1+n_j$, while, for direction $R_i \leftarrow R_j$, the formula changes to $1 + n_i$. Thus, we can pick the direction of each edge independently by considering simple statistics on the multiplicities of the join attribute id-values.

Example 4.4 *Consider our running example of view V_b shown in Figure 3. For edge (*`Personnel`,`Teaching`*) the multiplicity of the join attribute* `Fox` *is 2 in* `Personnel` *and 4 in* `Teaching`*. Thus, the minimum cost plan is* `Personnel` $\leftarrow$ `Teaching`*. Similarly, the multiplicity of* `DB` *in join column* `sem` *of base table* `Teaching` *is 2 and in join column* `cour` *in table* `Schedule` *is 3. Thus, the optimal selection for G_e is* `Personnel` $\leftarrow$ `Teaching` $\leftarrow$ `Schedule`*. The combined increase in the size of the base tables is (1+2)+(1+2)=6. (Notice that we count here the special delete tuples that, in practice are not created since they are removed at the end of the process).*

The existence of key/foreign key constraints in the schema can further expedite our algorithm. In particular, the selection of the optimal join execution graph is done by considering without any further calculation the edge direction from the foreign key to the key.

For multiple tuple deletes through a declarative delete statement, the analysis is similar. We break the problem into a set of sub-problems by considering each id-value of the join attributes independently. This means that the same edge on the join graph can be instantiated with different directions in each sub-problem. The join execution graph may not be a DAG anymore but will be decomposable into a set of DAGs. Again, for join graphs with maximum fan-out that is less than or equal to two, we can optimize the direction of edges independently.

Example 4.5 *Consider the delete command* delete *from V_b* where `emp='Fox'` *for the view of Figure 3. For edge (*`Personnel`,`Teaching`*) there is a single id-value of the join attribute (*`Fox`*) and the optimal direction of the edge in the*

view execution graph is Personnel ← Teaching *(as in the previous example). For edge (*Teaching,Schedule*) the join attributes (*sem=cour*) in the deleted tuples obtain two id-values for this delete statement:* PL *and* DB*. The multiplicity of the* PL *id-value is one in both tables; thus we arbitrarily chose direction* Teaching → Schedule *for the instance of the edge in the sub-problem (when* sem=cour=PL*). For* sem=cour=DB *the optimal direction is* Teaching ← Schedule*. Thus, the join execution graph has two instances for edge (*Teaching,Schedule*) of opposite direction that are processed separately.*

Updates to the view are a generalization of deletes and their analysis is similar. The only difference is that multiple "delete" tuples may get created during an update statement as discussed in section 3.3. It is easy to see that this only affects the constants in the discussed cost functions and the same arguments still apply.

5. Related Work

View update is an old problem [9]. Researchers have realized very early that a database update that reflects a view update may not always exist, and even if it does, it may not be unique. Dayal and Bernstein [8] formalized the notion of correct translation and defined constraints that guarantee it. They generate one translation for each view update but their views are restricted to those without join attributes in the view interface. Along the same lines, Bancilhon and Spyratos [2] used the concept of view complement to check the existence of unique translations, but computation of the view complement has been shown to be NP-Complete [7]. Tomasic [16] uses query containment to check the correctness of a translation but assumes that the translation is provided. For the same purpose, Medeiros and Tompa [5] use the chase [13] but do not provide a method for computing the translation.

Buneman et al. [4] studied ways to propagate view deletions to the sources based on provenance. Keller [10] studied the ambiguity of the translation. He enumerates all translations of a given update based on a number of criteria that guarantee the correctness of the translation, and chooses the one through an interaction with the user at view definition time [11]. Barsalou et al. [3] built on that work to update in a similar fashion object-based views defined over relational data. Keller's work is the one closest to ours. However, it does not support all the kinds of updates we support, it requires the schemas to be in BCNF, and for deletions it does not always preserve the base tables.

Achieving the required functionality for any view update without side effects is clearly not feasible with the existing relational model [10, 2, 9]. For that reason we had to extend the data model used at the physical level. The idea of extending the relational model with identifiers is reminiscent of flexible relations [1]. We go further by having identifiers associated with individual values.

depDOM

vID	display
1	CS
2	EE
3	Phil

Personnel

dep	emp
1	10
2	10
3	11

empDOM

vID	display
10	Fox
11	Jones

profDOM

vID	display
10	Fox
11	Jones

Teaching

prof	equip	sem
10	20	30
10	20	31
10	20	32
10	21	31
11	22	32

semDOM

vID	display
30	PL
31	OS
32	DB

equipDOM

vID	display
20	proj
21	lp
22	mic

Figure 7. Physical data instance

6. System Implementation

This section describes how the proposed functionality can be achieved using existing database technology. We consider relational databases simply because they are mature enough and form the majority of the existing storage systems. One way to simulate identifiers in relational databases is to introduce an additional column for every relational attribute. This additional column will keep the identifier of each id-value in that attribute, leaving the original column to represent the display form. The drawback of this *in-lined* approach is that if a logical value is repeated in multiple tuples in an attribute, its identifier will also be repeated. Alternatively, we can introduce a set of binary tables, referred to as *domain tables*. A domain table stores id-values from the domain of an attribute. The first column keeps the id-value identifier (vID), and the second its display form (display). Each domain table corresponds to one and only one attribute of a relation. The id-value identifier column in a domain table serves as a key. The relational tables need then to store only the id-value identifier. We will refer to these tables as *regular tables* to distinguish them from the domain tables. We follow this second approach. The domain and regular tables of the logical tables Personnel and Teaching of Figure 3 are illustrated in Figure 7.

User queries are expressed on the logical tables, and need to be translated to queries on the physical data tables. For this, first, every view involved in the query is replaced by its view definition. This process is equivalent to query unfolding [15, 12]. The select clause of the query is processed next. As specified by the user, it selects attributes from the regular tables, which means that the query result will be tuples of identifiers and not display forms that the user expects. For that, for each expression in the select clause referencing a regular table attribute a join with its domain table is introduced in the query. The join is based on the identifier attribute of the domain table and the referenced attribute of the regular table. The select clause expression

is then replaced by one using the display form attribute of the domain table that was introduced. The third step is to process the conditions of the where clause. For every expression referring to a table attribute of the logical schema, the same steps as above are performed. An exception to the above rule is the case in which an equality join condition appears in the where clause as a result of the query unfolding. In this case, the join condition has to remain unchanged so that the join will be based on the identifiers and not the display forms.

Example 6.1 *Consider the query* select v.day from V_b v where v.emp='Fox'. *After query unfolding, it becomes*

select s.day
from Personnel p, Teaching t, Schedule s
where p.emp='Fox' and p.emp=t.prof and
t.sem=s.cour

Next, due to the expression s.day *in the* select *clause, a join with the domain table* dayDom *of attribute* day *is introduced in the query and expression* s.day *is replaced by expression* dd.display *that selects the display form attribute from the introduced domain table. Similar steps are applied for the expression* p.emp *of the* where *clause and the query becomes*

select dd.display *AS* day
from Personnel p, Teaching t, Schedule s,
dayDom dd, empDom de
where de.display='Fox' and p.emp=t.prof and
t.sem=s.cour and s.day=dd.vID and
p.emp=de.vID

Notice that the second and third equality conditions of the where clause remained unchanged in order to ensure that the join between Personnel, Teaching *and* Schedule *is based on the identifiers.*

Since joins are expensive operators, in general, we have exploited the fact that id-value identifiers are keys for the domain tables so that regular-domain table joins can be executed more efficiently. Of course, query answering in the domain table approach is expected to be slower than in the approach that has the identifiers in-lined in the regular tables. However, the latter is expected to have a much larger cost in terms of space. In general, schema and value distribution knowledge may suggest a hybrid approach, and is a future research topic.

7. Experimental Results

We implemented the prototype on top of an industrial database system, and we tested it on a machine with a 3.2GHz CPU and 1GB of memory. We would like to experimentally compare our work with other view update algorithms. Unfortunately, this is not feasible since other approaches can not support all the kind of updates we support,

DB	Size (MB)	Tuples in V_b	On-demand (MB)	Eager (MB)
VIP_1	13.4	100K	35.8	118
VIP_2	26.3	200K	70.2	235
VIP_3	66.3	500K	175	588
VIP_4	138	1000K	364	1243
VIP_5	310	2850K	729	3060
TPCH	1200	6001K	3600	16000

Table 1. Test databases characteristics

and many of the update statements we randomly generated to test our code could not be executed in other systems. We did, however, conduct several experiments to evaluate the feasibility and correctness of our system. Our goal was to show that with the introduction of identifiers on values, queries can be executed correctly while any kind of updates can be performed on the views without side-effects on the view, and all these with a reasonable cost in time and space. We also compared our on-demand algorithm with the eager algorithm to show the benefit we get by carefully performing the updates. We tried both real and synthetic data. For the former we used a part of a real system, called VIP (Virtual Integration Prototype) that integrates more than 30 legacy systems used in AT&T. The schema of the data we considered is the following:

Orders(OrderNum, Type, Status)
Customers(Name, OrderNum, PhoneNo)
Biller(PhoneNo, AccountNum, ProvisNum)
Provisioning(ProvisNum, CircuitID)

To investigate how well our method can scale, we have created multiple different size instances of these tables. The sizes are indicated in the second column of Table 1. The third column of the same table provides the number of tuples in the view V_b=select * from Orders o, Customers c, Biller b, Provisioning p where c.OrderNum=o.OrderNum and c.PhoneNo=b.PhoneNo and b.ProvisNum=p.ProvisNum which we use in our experiments.

Loading A first step we performed was to take each (native) database and enhance it with ids on the values. This naturally increased the database size. The fourth and fifth column of Table 1 indicates the id-enhanced database size for the on-demand and eager approach, respectively. It can be noticed that the size increase rate is much larger for the eager case, which suggests that the on-demand strategy will scale better. For identifiers we used 64-bit integers. For small databases, one could use 32-bit identifiers with a significant improvement in space. The time for building these id-enhanced databases has a similar trend to the size. However, the id-enhancement of a native instance is an action performed only once, hence the time spent on it is not a critical factor.

Querying To investigate how the size increase affects query answering, twelve queries were considered. The first performs a selection from a single table with a small se-

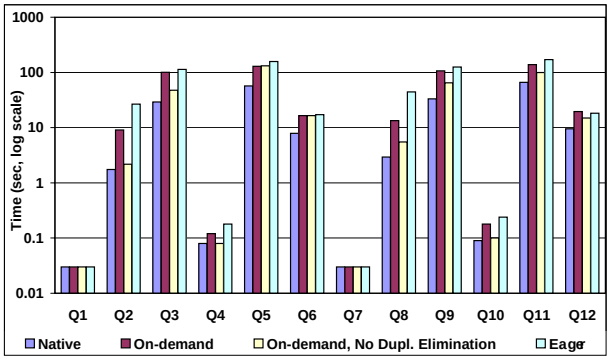

Figure 8. Query execution time on VIP_4

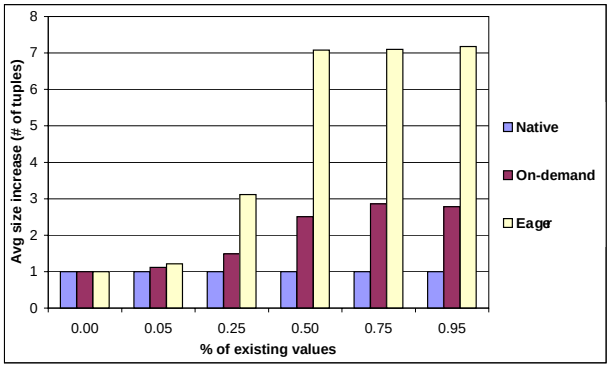

Figure 9. Size increase cause by insertions

lectivity factor. The second is also a selection from the same table but with a much larger selectivity factor. The third selects tuples from a view that joins two tables. The fourth is the same as the third with additional selection conditions in its where clause. The fifth and sixth are similar to the third and fourth but are on the view V_b that joins all four tables. The last six queries are the same as the first six but the number of attributes in their select clause has been doubled. We executed these twelve queries on the native databases and their respective on-demand and eager id-enhanced databases. Their execution time is reported in Figure 8, for only database VIP_4 due to space limitation, but similar observations hold for the rest. The figure has four columns for each query. The first, second and last are the query time for the native, on-demand and eager id-enhanced databases, respectively. As expected, the id-enhanced time is always larger than the time for the native, and this difference gets larger as the number of the attributes in the select clause and the conditions in the where clause increases. The reason is the additional joins with the domain tables that are required to access the display forms of the values. Furthermore, in order to merge the different clones of the same tuple when query results are returned, our algorithm always performs duplicate elimination, based on some auxiliary attribute we keep. This duplicate elimination is a significant factor for this additional time. To illustrate this, we have included in the figure, for each query, the execution time on the on-demand id-enhanced database without the duplicate elimination (third column). The remain difference between the on-demand and the eager case, is due to the larger number of tuples in the latter.

Insertions Insertions on a view table are translated to a tuple insertion in each of the base tables of the view. An insertion on a base table, on the other hand, may require additional tuples to be inserted in other tables as well, as explained in Section 3.1. We generated random insert statements and executed them on the native, on-demand and eager databases. Figure 9 illustrates the average number of tuples that need to be inserted as a result of an insert command. In the first experiment, none of the values in the inserted tuples is already in the database. As such, no additional tuples are required, and the database size increase is the same for the native, the on-demand and the eager case. In the second experiment 5% of the inserted values are already in the database. The remaining 95% are new values. We notice a slightly larger increase in the size of the the on-demand and the eager database. As we keep increasing the percentage of existing values in the insert commands, the database size increase gets larger for the on-demand case and much larger for the eager. In most applications, we expect that insertions will bring new data to the database and will not re-insert data that is already stored which means that the insertion cost will not be large, especially if the on-demand approach is used. In all experiments, the execution time of an insert statement is found to be always in the range of milliseconds, independently of the used strategy.

Deletions Deletion statements on a table are executed similarly to deletions on tables of a native database. Only few extra joins with the domain tables may be required, but our experiments show no noticeable delay due to this. To evaluate our deletion through views, we generated view deletion statements and we executed them on the eager and the on-demand databases until there were no more tuples left in the view. Note that the generated delete statements are declarative commands which means that one single statement may delete multiple tuples from the view. Figure 10 illustrates how the size of one of the databases evolved while we were performing the deletions. Notice the paradox that after each delete command the database size increases. This is justified by the fact that additional tuples are inserted in the database not only to preserve the base tables instances but also to preserve the view on which the deletion takes place from side-effects. We have shown that for a declarative delete our algorithm is optimal. For a sequence of independent declarative deletes, further optimization can be done. This study is left for future work. A second observation is that the rate of increase is similar in the eager and the on-demand approach and such that

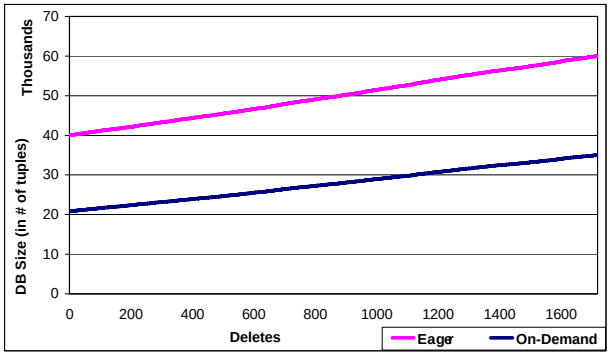

Figure 10. Effect of deletions in space

even after the last (1718th) delete command, the size of the on-demand database is much smaller than the size the eager database had at the beginning of the experiment. This indicates once again the gain we have by following the on-demand approach. The figure also indicates that at the end of the experiment the increase in size is approximately 75%, but this is only after all the view tuples have been removed. We expect that in real scenarios, deletions that remove all the tuples of a view are rare, in which case the size increase will be much less. Furthermore, the experiment was on a generic schema. There are various situations in which the table size increase is much less. For instance, if the join uses key/foreign key attributes, a situation commonly met in practice, the size of the table that has the key attribute remains unchanged during the on-demand view deletions.

We also experimented with synthetic (TPCH) data of various sizes. We performed the same kind of experiments as we did for the real VIP data and we reached similar conclusions. An indication of the space increase we had when loading the TPCH databases is also given in Table 1.

All our experiments were performed using our Java-based research prototype system, where most of the processing was performed outside the database. We expect that porting the system to some other language and integrating it in a DBMS, e.g., as stored procedures, will yield a significant performance improvement.

8. Conclusion

This work presented a novel framework for dealing with the view update problem, which, in contrast to similar approaches, can correctly translate any kind of view updates to base table updates with no side-effects. In addition, for the case of deletions, it guarantees that the virtual instances of the base tables of the affected view will not be affected. The success of the system depends on separating the data level into a logical and a physical, storing additional information in the latter, and hiding it from the users through the right translation mechanism of queries, update statements and results between the two levels. We evaluated the system and showed that the extra cost required does not prohibit the use of the framework for cases where this kind of functionality is needed.

In the future, we plan to extend the framework to include, apart from select-project-join views, views with aggregations and set operators, e.g., union, intersect and set difference. Furthermore, we plan to study how modifications in one view affect other views that are defined using the one that is updated and how the system behaves when new views are introduced.

Acknowledgments: We would like to thank Phil Bernstein and Sergey Melnik for their useful comments.

References

[1] S. Agarwal, A. M. Keller, G. Wiederhold, and K. Saraswat. Flexible Relation: An Approach for integrating Data from Multiple, Possibly Inconsistent Databases. In *ICDE*, pages 495–504, Mar. 1995.

[2] F. B. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM TODS*, 6(4):557–575, Dec. 1981.

[3] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. *SIGMOD Record*, 20(2):248–257, June 1991.

[4] P. Buneman, S. Khanna, and W. C. Tan. On Propagation of Deletions and Annotations Through Views. In *PODS*, pages 150–158, 2002.

[5] C. Medeiros and F. Tompa. Understanding The Implications Of View Update Policies. In *VLDB*, Aug. 1985.

[6] E. F. Codd. Is Your DBMS Really Relational? *ComputerWorld*, 1985.

[7] S. Cosmadakis and C. Papadimitriou. Updates of Relational Views. In *PODS*, page 317, Mar. 1983.

[8] U. Dayal and P. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM TODS*, 8(3):381–416, 1982.

[9] U. Dayal and P. A. Bernstein. On the Updatability of Relational Views. In *VLDB*, pages 368–377, 1978.

[10] A. M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. *SIGMOD*, Mar. 1985.

[11] A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.

[12] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[13] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM TODS*, 4(4):455–469, 1979.

[14] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Boston, 2nd edition, 2000.

[15] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *SIGMOD*, pages 65–78, 1975.

[16] A. Tomasic. Correct View Update Translations via Containment, Dec. 1993.

[17] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB Journal*, 5(2):101–118, 1996.