Data Integration using Self-Maintainable Views

Ashish Gupta*
Stanford University
agupta@cs.stanford.edu

H. V. Jagadish
AT&T Bell Laboratories
jag@research.att.com

Inderpal S. Mumick
AT&T Bell Laboratories
mumick@research.att.com

Abstract

Integration of data from multiple databases is an important problem. We consider a model for data integration wherein data from multiple databases is combined into an integrated view that is materialized and stored in a database. All queries on the view are then answered directly from the view, without having to go to the different databases. Such a model is the preferred one for applications where fast response and high availability are important. A problem that must be solved to realize this model is that the view has to be kept current in response to changes occurring in the underlying databases. The databases can supply the integrating system with a log of their changes, but little more can be expected. Thus, incremental view maintenance for the integrated view is very important. Further, the view maintenance should be able to proceed without accessing the underlying databases every time one of the databases reports a change.

We identify that the outer-join operator is very important in defining integrated views. We give incremental view maintenance algorithms for the outer-join and its variants. We define the concept of a *self-maintainable* views – these are views that can be maintained using only the contents of the view and the database modifications, but without accessing any of the underlying databases. We derive tight conditions under which a view is self-maintainable, and give algorithms for self-maintenance. For example, we show how to self-maintain several types of select-project-join and outer-join views upon insertions, deletions and updates.

1 Introduction

Most large organizations have related data in distinct databases. For example, a university has data about students in several databases. A company such as IBM or AT&T has information about the same customers in several databases. Many of these databases may be legacy systems that have evolved separately over time. Others may even be new systems, but that are being implemented separately for organizational reasons, such as funding and ownership. Integrating data from such distinct databases is a pressing business need. Companies want to analyze the behavior of their customers as a whole, and cannot do so unless the information is first integrated.

A common approach for integration is to define an integrated view, and then map queries on the integrated view onto queries on the individual systems. This model is not good for applications where response time is critical. Further, in this model, the availability of the integrated view is the lowest denominator of the availabilities of all databases touched by a query.

We propose an alternative model, good for applications where fast response to queries and high availability are important: The integrated view is materialized and stored in a target database. All queries on the view are then answered directly from the view. A copy of the underlying tables is *not* stored in the target database. This approach solves the difficulties with the query translation approach. However, it does involve the additional storage expense for the materialized integrated view. It has

*Current address: IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099

been our experience that, in many applications, query performance needs alone are enough to justify this additional expense. Further, it is not necessary to store This model of integration is being called the *data warehousing* model [Z+94].

Provisioning continues in the underlying database, and a problem that must be solved is how to maintain the integrated view current as the underlying databases change. It may be expensive or impractical to obtain much more than just a periodic log of local updates from each underlying database. In fact, the change log may not be supplied at the same frequency by each component, and all of the components may not even be connected at the same time. However, a user of the integrated view would like a guaranty that the integrated view correctly reflects all changes supplied by a component database.

Maintaining the integrated database current in such a world requires one to have efficient incremental view maintenance techniques. In deed, it is important to look one step further when defining the integrated views, and define views that are maintainable in an efficient manner. Efficient maintenance means, at the very least, that the view be maintainable when a log of changes arrives from a component database, regardless of whether the component databases themselves are accessible or not. Under such circumstances, it becomes crucial that the integrated view be *self-maintainable*, meaning that view maintenance should be possible without requiring access to any underlying database, and without access to any information beyond the view itself and the log of the changes.

This paper looks at the problem of view maintenance in this framework. There are two specific problems we look at:

1. Outer-joins [Dat89, ISO92], and variants of outer-join operations are very useful for data integration. How does one incrementally maintain outer join views?

2. How can one tell whether a given view is self-maintainable? What are the syntactic restrictions for a view to be self-maintainable, so that the data integrator may write views that are self-maintainable. What are the algorithms for self-maintaining views?

Results

- We present algorithms to do incremental maintenance for views defined using the outer-join operator. We consider the full outer-join, the left and right outer-joins, the natural full outer-join operators, and a *match* operation that we define.

- We define the self-maintainability problem as the problem of maintaining a view in response to a modification (insert, delete, or update) using only the view and the set of changes to the referenced relations (and without access to the full referenced relations). We obtain complete conditions and syntactic restrictions under which SPJ views (which are views defined using the select-from-where clause of SQL) are self-maintainable with respect to insertions, deletions, and updates. For example, we derive that most SPJ views are not self-maintainable with respect to insertions, but they are often self-maintainable with respect to deletions and updates. We present algorithms for the self-maintenance of such self-maintainable views.

- We model updates directly for the purpose of view maintenance. Previous view maintenance work has treated an update to a tuple as an insertion of a new tuple plus the deletion of the old tuple. We show that the view maintenance problem for updates is easier when updates are modeled directly rather than as a set of deletions and insertions; in particular updates can be self-maintained even when insertions cannot be self-maintained.

- We identify classes of outer-join views that are self-maintainable and derive algorithms for the self-maintenance of outer-join and match views. We show that outer-join views are usually self-maintainable, and match views are always self-maintainable. This is a fortunate co-incidence, since our empirical observation indicates that the outer join is an important operator in many integrated view definitions.

Paper Outline The two intertwined concepts on which this paper is based are self-maintenance and outer-joins. In Section 2 we argue, by example, that these two concepts are important for data integration.

Section 3 gives a few definitions and notation that we use in the paper. View maintenance for outer join views is discussed in Section 4. The idea of self-maintenance is described in Section 5, and then in Section 6 we discuss self-maintenance for regular SPJ views. Outer-join maintenance and self-maintenance are brought together in Section 7, where we discuss self-maintenance for outer join views. In Section 8, we discuss related work and then we conclude with Section 9.

2 Motivating Example

The primary motivation for this work is data integration, and in particular some specific data integration problems faced by AT&T. The example we consider comes from a similar application in a university.

EXAMPLE 2.1 Consider three databases about students maintained by different parts of a university. The computer science department maintains a relation `cs_phd` containing, for each student in the department, his student identifier (Stu_id), the name of his advisor, number of years in the PhD program, whether the student has finished his course requirements, and the qualifying exam taken, if any. The registrar's office maintains the relation `registrar` that contains, for each student, their name, address, phone extension, the number of units the student is registered for, and the tuition paid. We assume that if a student is not registered then `registrar` does not have a tuple for that student. The relation `library` states the borrowing privilege for each student. Each relation has a key, which we underline in the schema below.

`cs_phd`$(\underline{Stu\_id}, Advisor, Year, Courses, Qual)$
`registrar`$(\underline{Stu\_id}, Name, Address, Phone, Units, Tuition)$.
`library`$(\underline{Stu\_id}, Borrowing\_Privilege)$.

The three relations may be maintained in separate databases by different offices in the university. The university graduate office may want to look at all attributes of all student, regardless of what databases these appear in. For this purpose, the university defines an integrated view by joining the three relations on the attribute *Stu_id* as follows (we use a syntactic variant of SQL, and we use a subset of attributes from each relation so as to make the view readable):

(A): `CREATE VIEW univ_student AS`
`SELECT` $Stu\_id, Name, Address, Phone, Borrowing\_Privilege, Year$
`FROM cs_phd`$(Stu\_id, Year)$,
`registrar`$(Stu\_id, Name, Address, Phone)$,
`library`$(Stu\_id, Borrowing\_Privilege)$.

We use a syntactic shorthand in SQL to represent equijoin predicates and selected attributes. The attributes of a relation are identified with variables after the relation name in the `FROM` clause. An equijoin predicate is then represented by using the same variable name for two attributes. For example, in the view A, we use the variable *Stu_id* for the attributes `cs_phd`.Stu_id, `registrar`.Stu_id, and `library`.Stu_id, implying an equijoin between these three relations. The set of attributes in the `SELECT` clause identifies the attributes of the view and are called *distinguished* attributes.

Now consider how changes in the individual databases might be propagated into the integrated view. Consider an update to a student's phone extension, say `registrar`$(s1, john, holly\ street, 1234)$

becomes library$(s1, john, holly\ street, 5678)$. This update can be propagated to the view univ_student by simply changing the phone extension for the univ_student tuple, if any, that has $Stu\_id = s1$. Similarly, if student $s1$ is deleted from relation cs_phd then we can update the view by removing the univ_student tuple, if any, with $Stu\_id = s1$. However, if a new student is inserted into the relation library then we cannot modify view univ_student without accessing the relations cs_phd and registrar. Since, we could maintain the view in response to updates and deletions using only the modification and the view, but we couldn't do the same for insertions, we say that the view univ_student is self-maintainable with respect to updates and deletions, but not with respect to insertions.

View definition (A) is not well suited for many data integration applications because it loses information. Consider a student who is on leave of absence and thus does not appear in the registrar's database. This student may still have borrowing privileges and a valid status in the computer science department. However, the integrated view would lose this student due to the join with relation registrar.

If it is desired to keep information about students that do not appear in all databases, then the *outer join* operator should be used to integrate the relations. A variant of view A can be written using a natural full outer-join:

```
(B):  CREATE VIEW univ_student AS
          SELECT Stu_id, Name, Address, Phone, Borrowing_Privilege, Year
          FROM  cs_phd(Stu_id, Year) |&| registrar(Stu_id, Name, Address, Phone) |&|
                library(Stu_id, Borrowing_Privilege).
```

The symbol |&| is used as shorthand for the keyword "full outer-join." As before, repeated variable names are used to represent equijoins. Intuitively, the above outer join view contains the tuples in the SPJ view (A) shown earlier and also contains all the tuples of the three relations that do not contribute to (A). The latter tuples have NULL values for the attributes of the relation that did not contribute a joining tuple. As an example, consider the following database (considering only the relevant attributes) cs_phd $= \{(s1, 1^{st}), (s2, 2^{nd})\}$, registrar $= \{(s1, john, holly\ ave, 1234)\}$, library $= \{(s1, yes), (s3, no)\}$. The integrated view univ_student is:

Stu_id	*Name*	*Address*	*Phone*	*Borrowing_Privilege*	*Year*
$s1$	*john*	*holly ave*	1234	*yes*	1^{st}
$s2$	NULL	NULL	NULL	NULL	2^{nd}
$s3$	NULL	NULL	NULL	no	NULL

A full outer-join operation is defined as a binary operation as it is usually not associative. However, in this example, the outer-join predicate is an equijoin, and we are combining the joined attributes (Stu_id in the three relations) into one attribute in the view (regardless of which relation the student id comes from, it is placed in the first attribute of the view). Such an outer-join is called a natural outer-join, and it is associative. So we have taken the liberty of writing it as a muti-way outer-join. Further, in this example, the natural full outer-join is on the key attributes. A natural outer-equijoin on the key attributes is a very commonly occurring operator in data integration applications and we refer to it as the match operator. Like a regular join, a match operator is associative. □

Outer joins are of importance in other domains besides data integration, such as extended relational systems [MPP+93], as illustrated in the Appendix A.

3 Background and Notation

We consider the simple SELECT-PROJECT-JOIN views written using SQL SELECT-FROM-WHERE clauses. Such views are called SPJ views. If such a view has only one relation occurrence in the FROM clause, it will be denoted as an SP (select and project) view. We will also consider views using outer-joins as discussed in Example 2.1.

We use the term modification to refer collectively to insertions, deletions, and updates. We will use keys to define updates and to derive self-maintenance algorithms.

Definition 3.1 (Key Attributes key(R):) The key attributes for relation R is a subset of the attributes of R such that no two tuples of R can have the same values for these attributes. □

A relation may have several keys, and any one of these could be used for any of the results we derive. We will underline the key attributes of every relation. For instance, if the attributes A and C together form the key of relation $r(A, B, C, D)$ then we will write relation R as $r(\underline{A}, B, \underline{C}, D)$.

Definition 3.2 (Updates:) For a relation R that has a key, a tuple in R is said to be updated if one or more attributes of the tuple are assigned a value different from its original value. An update to tuple r that results in tuple r' is represented as $\mu(r, r)$. □

Definition 3.3 (Local and Join Predicates:) Consider a predicate $p(\bar{X})$ in the `WHERE` clause of a view. $p(\bar{X})$ is said to be a local predicate if all variables in $\bar{X}$ appear in a single relation in the `FROM` clause. Otherwise, $p(\bar{X})$ is said to be a join predicate .
□

Definition 3.4 (Derivation tree [RSUV89, Mum91]:)(Informal) A derivation tree for a tuple t is a tree representation of the manner in which the tuple is derived from base relations. For a tuple t in a base relation, the derivation tree consists of a single node labeled with tuple t. For a tuple t in a view defined as a join between relations $R_1, \ldots, R_k$, a derivation tree consists of a root node labeled with the tuple t, and one child node for each of the tuples $(r_1 \in R_1, r_2 \in R_2, \ldots, r_k \in R_k)$ that join to derive tuple t. □

Definition 3.5 (Tuple Derivation:) Consider view V defined using relation R, and possibly other relations. A tuple r in R derives a tuple v in V if tuple r appears in some derivation tree for tuple v.
□

4 View Maintenance for Outer Joins

In this section we develop an algorithm to incrementally maintain outer-join views. We use the following syntax to define a view V as an outer-join of relations R and S. Recall that the outer-join operator, $|\&|$, is a binary operator.

```
CREATE VIEW V  AS
        SELECT X̄
        FROM R(Ȳ) |&| S(Z̄)
        WHERE g(Ȳ, Z̄).
```

$\bar{Y}$ and $\bar{Z}$ are distinct sets of variables that represent the attributes of relations R and S respectively. $\bar{X}$ is the set of distinguished attributes. $g(\bar{Y}, \bar{Z})$ is a conjunction of predicates that represent selection and join predicates. Note, equijoins are represented by explicit equality predicates in $g(\bar{Y}, \bar{Z})$. We discuss how to incrementally maintain the full outer-join view V defined as above, when the relations R and S may be modified using delete and insert operations.

To begin with, we restrict ourselves to views in which the keys of both relations R and S are distinguished, *i.e.*, in the set of attributes $\bar{X}$. We refer to the set of changes to relation R as $\Delta(R)$, that consists of insertions $\Delta^+(R)$ and deletions $\Delta^-(R)$. Similarly for relation S. We want to compute $\Delta(V)$ given $\Delta(R)$ and $\Delta(S)$.

We use the differencing techniques used to maintain regular SPJ views [GMS93], to rewrite the original view definition to obtain queries that compute $\Delta(V)$ which is the set of changes to view V. The following two queries together compute $\Delta(V)$.

(a): SELECT $\bar{X}$
FROM $\Delta(R(\bar{Y}))$ |& $S(\bar{Z})$
WHERE $g(\bar{Y}, \bar{Z})$

(b): SELECT $\bar{X}$
FROM $R^\nu(\bar{Y})$ &| $\Delta(S(\bar{Z}))$
WHERE $g(\bar{Y}, \bar{Z})$.

Here R^ν represents the new value of R after modification. All other values in these equations refer either to the pre-modification values or to the deltas themselves. These two queries do not compute the entire change to the view. First, we explain the intuition for why this is the case.

Query (a) computes the effect on V of inserts and deletes to relation R. Consider a tuple r^+ inserted into R and the effect the insertion may have on the view. If r^+ does not join with any tuple in s, then $r^+.NULL$ has to be inserted into view V. If instead, r^+ does join with some tuple s in S, then $r^+.s$ is inserted into the view. Both these consequences are captured by Query (a) by use of the left-outer-join. However, if r^+ does join with some tuple s then there is a possible side effect that is not mentioned in query (a). The tuple $NULL.s$ may have to be deleted from the view V if $NULL.s$ is in the view. This will be the case if previously tuple s did not join with any tuple in R.

Similarly, a deletion r^- from R not only removes a tuple from the view, as captured by Query (a), but may also precipitate the insertion of a tuple $NULL.s$. This will be the case if r^- is the only tuple that joined with tuple s before the deletion. Similarly, Query (b) handles the modifications to table S. In the table below we list the possible effects and side-effects of evaluating Queries (a) and (b). Subsequently, we state Algorithm 4.1 for doing incremental outer-join view maintenance. The algorithm has two blocks, one each for the two queries. Within each block, the side effects are handled as discussed above.

The following table summarizes the possible effects of evaluating Queries (a) and (b) that translate insertions and deletions from relations R and S to changes in the view.

Delta Query	Possible effect on view V	Possible side effect
query (a)	insert $r^+.s$	delete NULL.s
$\Delta(R)$ |& S	insert $r^+.NULL$	
	delete $r^-.s$	insert NULL.s
	delete $r^-.NULL$	
query (b)	insert $r^\nu.s^+$	delete $r.NULL$
R^ν &| $\Delta(S)$	insert $NULL.s^+$	
	delete $r.s^-$	insert $r.NULL$
	delete $NULL.s^-$	

Before we explain the algorithm, we illustrate outer-join view maintenance via an example.

EXAMPLE 4.1 Consider vertical fragments of two relations from the university-database of Example 2.1 (vertical fragments are used to avoid having a very wide view).

cs_phd($\underline{Stu\_id}$, $Advisor$).
registrar($\underline{Stu\_id}$, $Name$).

Consider the following full outer-join view on the two relations. Attribute Stu_id appears twice, once each for relations cs_phd and registrar.

```
CREATE VIEW student AS
    SELECT Stu_id1, Stu_id2, Name, Advisor
    FROM cs_phd(Stu_id1, Advisor) |&| registrar(Stu_id2, Name).
```

Consider an initial database and the resulting view `student`.

$\texttt{cs\_phd}= \{(s1, adv1), (s3, adv3)\}$.
$\texttt{registrar}= \{(s1, john), (s2, mary)\}$.
$\texttt{student}= \{(s1, s1, john, adv1), (NULL, s2, mary, NULL), (s3, NULL, NULL, adv3)\}$.

Now let us modify the base relations and see how the view changes. The changes to the base relations are:

$\Delta^+(\texttt{cs\_phd}) = \{(s2, adv2)\}$, $\Delta^-(\texttt{cs\_phd}) = \{(s1, adv1)\}$, $\texttt{cs\_phd}^\nu = \{(s2, adv2), (s3, adv3)\}$.
$\Delta^+(\texttt{registrar}) = \{(s3, sara)\}$, $\Delta^-(\texttt{registrar}) = \{(s1, john)\}$, $\texttt{registrar}^\nu = \{(s2, mary), (s3, sara)\}$.

First we apply query (a) to compute the change to view `student`. The query is computed by left-outer-joining $\Delta(\texttt{cs\_phd})$ with `registrar`. The result is:

$\Delta^+(\texttt{student}) = \{(s2, s2, mary, adv2)\}$, $\Delta^-(\texttt{student}) = \{(s1, s1, john, adv1)\}$.
$\texttt{student}^\nu = \{(NULL, s2, mary, NULL), (s2, s2, mary, adv2), (s3, NULL, NULL, adv3)\}$.

$\texttt{student}^\nu$ does not account for the side effects of joining tuples of `registrar` with inserted `cs_phd` tuples, or of joining tuples of `registrar` with deleted tuples of `cs_phd`. Thus, the side effects need to be taken into account. First, consider $\texttt{registrar}(s2, mary)$ that joins with the inserted tuple $\texttt{cs\_phd}(s2, adv2)$. Thus, we delete $\texttt{student}(NULL, s2, mary, NULL)$. Second, $\texttt{registrar}(s1, john)$ joins with deleted tuple $\texttt{cs\_phd}(s1, adv1)$. We need to check if $\texttt{registrar}(s1, john)$ joins with any other `cs_phd` tuple to contribute a tuple into `student`. After the deletion there is no tuple in $\texttt{student}^\nu$ with key value $s1$, the key for `registrar`. Thus, we infer that $\texttt{registrar}(s1, john)$ does not join with any `cs_phd` tuple and we insert $(NULL, s1, john, NULL)$ into the view. After these changes, the value of view `student` is:

$\texttt{student}^\nu = \{(NULL, s1, john, NULL), (s2, s2, mary, adv2), (s3, NULL, NULL, adv3)\}$.

Now, let us consider the changes computed by query (b) that right-outer-joins relation $\texttt{cs\_phd}^\nu$ with $\Delta(\texttt{registrar})$.

$\texttt{cs\_phd}^\nu = \{(s2, adv2), (s3, adv3)\}$.
$\Delta^+(\texttt{registrar}) = \{(s3, sara)\}$, $\Delta^-(\texttt{cs\_phd}) = \{(s1, john)\}$.
$\Delta^+(\texttt{student}) = \{(s3, s3, sara, adv3)\}$, $\Delta^-(\texttt{student}) = \{(NULL, s1, john, NULL)\}$.
$\texttt{student}^\nu = \{(s2, s2, mary, adv2), (s3, NULL, NULL, adv3), (s3, s3, sara, adv3)\}$.

As before, we need to consider the side effects of joining tuples in `cs_phd` with inserted/deleted tuples in `registrar`. Tuple $\texttt{cs\_phd}(s3, adv3)$ joined with inserted tuple $\texttt{registrar}(s3, sara)$. Thus we remove $\texttt{student}(s3, NULL, NULL, adv3)$ if it appears in the view, which it does. No `cs_phd` tuple joined with the deleted tuple, so is no side effect to take care of. The resulting view is:

$\texttt{student}^\nu = \{(s2, s2, mary, adv2), (s3, s3, sara, adv3)\}$.

Indeed, this is the outer join of the relations $\texttt{cs\_phd}^\nu = \{(s2, adv2), (s3, adv3)\}$ and $\texttt{registrar}^\nu = \{(s2, mary), (s3, sara)\}$. □

Note, the example shown above illustrates the need for using keys in order to take care of side effects in the outer-join-view maintenance process. Now we state the actual algorithm for maintaining outer-join views incrementally. The algorithm assumes that the view includes the keys of both the participating relations.

We use the operator $\uplus$ to take a union of two sets U and V where each element of one of the sets, say U, has a sign associated with it. The sign is either "+" (to indicate insertions) or "−" (to indicate deletions). The operation $\uplus$ produces a third set W that has all elements of sets U and V except those that occur in V and that occur in U with a sign "−."

Algorithm 4.1

Input: A 2-relation outer-join view definition C
The old value of the view V
Set of changes $\Delta(R)$ and $\Delta(S)$, with signs.

Output: Set of insertions made to V
Set of deletions from V

Method

```
% First use query (a)
Initialize sets potential_deletions, potential_additions, Δ(V) to be empty
compute Δ(V) using query (a)
    IF Δ(V) ≠ ∅                                          % Query (a) changes value of View V
        V^ν = V ⊎ Δ(V)                                   % ⊎ is as defined above
        FOR each tuple s in S that joined with some tuple r in Δ(R)                     @
            IF s joined with r+ add NULL.s to the set of potential_deletions            @
            IF s joined with r- add NULL.s to the set of potential_additions            @
        FOR each tuple NULL.s in the set potential_additions                            @
            IF s appears in any tuple in the view V^ν THEN                              @
                Remove NULL.s from potential_additions                                  @
        V^ν = V^ν ⊎ potential_additions                                                 @
        FOR each tuple t in the set potential_deletions                                 @
            IF t is in view V^ν then delete t from V^ν                                  @
            % The above step is correct only when the key for the updated relations are in V
% Next use query (b)
Initialize sets potential_deletions, potential_additions, Δ(V) to be empty
compute Δ(V) using query (b)
    IF Δ(V) ≠ ∅                                          % Query (b) changes the value of view V
        V^ν = V ⊎ Δ(V)
        FOR each tuple r in R^ν that joined with some tuple in Δ(S)
            IF r joined with s+ add r.NULL to the set of potential_deletions
            IF r joined with s- add r.NULL to the set of potential_additions
        FOR each tuple r.NULL in the set potential_additions
            IF r appears in any tuple in the view V^ν THEN
                Remove r.NULL from potential_additions
        V^ν = V^ν ⊎ potential_additions
        FOR each tuple t in the set potential_deletions
            IF t is in view V^ν then delete t from V^ν
V+ = V^ν - V                                             % Set of insertions into V
V- = V - V^ν                                             % Set of deletions from V
```

◇

The above algorithm assumes that the view includes the keys of both the participating relations. If the view does not include the keys of the relations, then a tuple in view V may occur multiple times. We use a count with each tuple to indicate the multiplicity of the tuple in a view. counts can be used to maintain correctly outer-join views with duplicates by a straightforward extension of the above algorithm using the techniques described in [GMS93]. We omit the details from this paper.

We also discuss how to incrementally maintain three variants of outer-join views that are important for data integration: left-outer-join, right-outer-join, and match. We discuss left and right-outer-join view maintenance algorithms in Appendix B, and the algorithm for maintaining match views in Section 7.2.

5 The Self-Maintainability Problem

EXAMPLE 5.1 Consider the university-students database introduced in Example 2.1. The three relations `cs_phd`, `registrar`, and `library`, may all be owned and administered by different parts of the university. Thus, they may all reside on different databases as illustrated by Figure 1. The integrated materialized view that is used to provide fast query answering, needs to be changed as the component databases change. The view may be difficult to maintain because not all the underlying databases may be available always, or the databases may be expensive to access. In addition, a distributed update involves concurrency control problems, interface translations, `etc.` Therefore, it is desirable that the view be maintained using solely the contents of the view and the modification. Multiple remote database accesses, and all its associated problems, can be avoided if such maintenance is possible. □

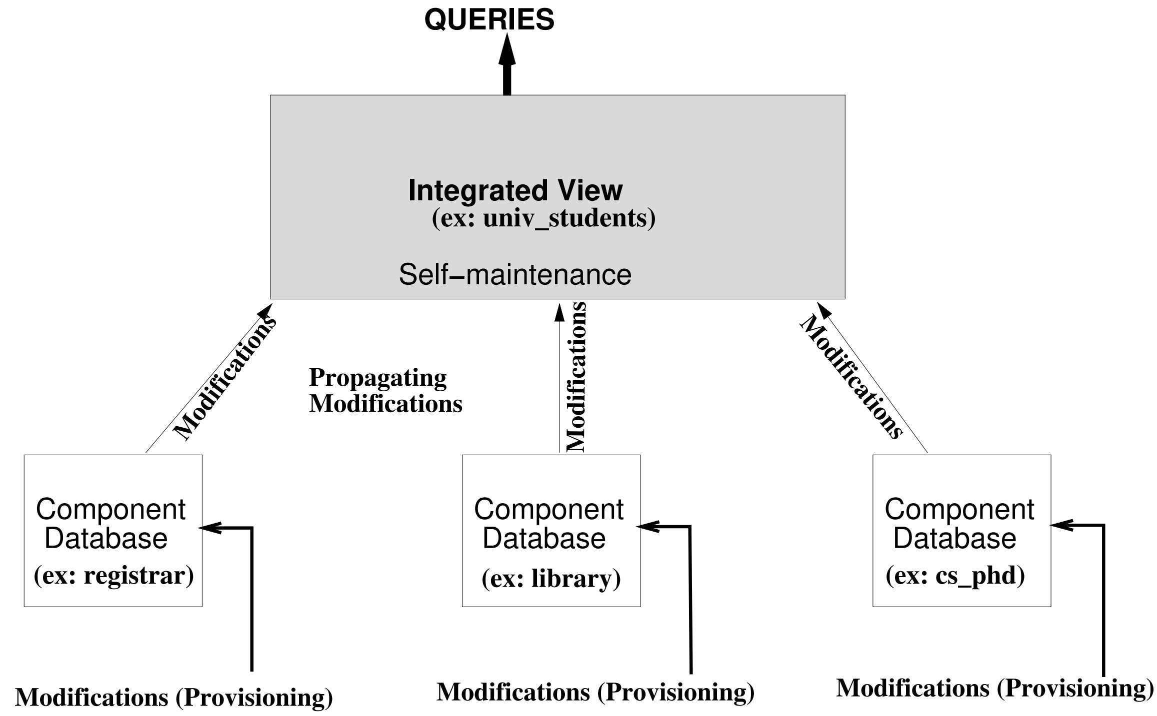

Figure 1: Integrating Data from Multiple Databases

Whether such maintenance is possible depends on both the definition of the view and on the type of modification. If for a given view and modification we can indeed determine the effect of the modification using only the contents of the view, then we say that the view is *self-maintainable* for that modification. For a given view, a modification has at least two components, namely the particular relation that is modified and whether the modification is an insertion, deletion, or update. There are also other finer distinctions such as which attribute of a relation is updated, or the actual value of the modified attribute. However, we restrict ourselves to classifying self-maintainability with respect to the relation being modified, and the type of modification.

We consider the three types of modifications and for each we identify classes of views that are self-maintainable with respect to the modified relation. We omit mentioning "with respect to modified relation R" when the context makes it clear that relation R is modified. Also, henceforth the phrase "with respect to R" implicitly assumes that R is used in the view under consideration.

Definition 5.1 (Self Maintainability with respect to a Modification:) A view V is said to be self-maintainable with respect to a modification type (insertion, deletion, or update) if for all database states, the view can be self-maintained in response to a modification of the indicated type to the base relations. □

In the following sections we will consider three types of views. Views defined using the selection and projection operations alone are called SP views. Views defined using the selection, projection, and join operations are called SPJ views. Views defined using outer-joins are called OJ views. As an especially useful type of OJ views, we also discuss self-maintenance of match views.

A view is said to be satisfiable [Shm87] if there exists a database for which the view is non-empty. Satisfiability of SPJ views can be checked easily. An unsatisfiable view is always empty, and hence trivially self-maintainable. In the analysis and theorems of the following sections, we will assume that the SP and SPJ views are satisfiable. The notion of satisfiability does not apply to OJ views. Also, we will assume that the views do not use predicates, or sets of predicates, that form a tautology (are always true).

6 Self-Maintenance for SPJ views

In this section we syntactically characterize classes of SPJ views that are self-maintainable in response to insertions, deletions, and updates.

6.1 Insertions

Views are self-maintainable for insertions only under very limited circumstances. We will show that it is not possible to self-maintain an SPJ (select-project-join) view joining at least two distinct relations upon an insertion into a component relation. Even a view involving a self-join of a relation with itself may not be self-maintainable. For instance, consider the following example:

EXAMPLE 6.1

(V): `CREATE VIEW` Grandparent `AS`
`SELECT` X, Y, Z
`FROM` $Parent(X, Y)$, $Parent(Y, Z)$.

When a new tuple is added to the *Parent* relation, it could join with many previously existing tuples in the *Parent* relation, and there is no way to identify these given only the tuples in the *Grandparent* relation, even though the intermediate Y attribute has been retained in the *Grandparent* relation. For instance, the *Grandparent* relation could be empty, even though the *Parent* relation is not. Therefore, the *Grandparent* relation is not self-maintainable with respect to insertions. □

Theorem 6.1 *The self-maintainability of an SPJ view in response to insertions can be determined by constructing two new SPJ views and testing if these two views are equivalent. The test can be done in time exponential in the number of self joins in the SPJ view.* □

Proof: (outline) Two new SPJ views are constructed, such that the view is self-maintainable if and only if the two queries are equivalent. One of the two views tries to get the relevant information about the base relation from the given SPJ view, and the other goes to the base relation itself. Query containment [Ull89, CV92, GSUW94] techniques are used to check if these two view are equivalent. □

Corollary 6.2 *An SPJ view that takes the join of two or more distinct relations is not self-maintainable with respect to insertions.* □

Corollary 6.3 *All SP views are self-maintainable with respect to insertions. An SPJ view defined using self-joins over a single relation R is self-maintainable if every join is based on* key(R). □

Algorithm for self-maintenance on Insertions: For the self-maintainable cases identified in Corollary 6.3 self-maintenance is easy to do. For SP views self-maintenance can be done by doing a selection and a projection on the newly inserted tuples. For an SPJ view with self-joins on a key, the critical observation is that the newly inserted tuples can only join with themselves. So self-maintenance involves evaluating the view definition on the inserted set of tuples.

6.2 Deletions

Theorem 6.4 *An SPJ view V that takes the join of one or more relations $R_1, \ldots, R_n$ is self-maintainable with respect to deletions to R_1 if and only if all derivation trees for a tuple t in view V use the same tuple r from R_1 for any individual occurrence of R_1 in the view definition, and it is possible to identify this tuple r from the tuple t alone, without referring to the contents of any R_i.* $\Box$

A sufficient condition for the conditions of Thm 6.4 to be true is when, for some key of relation R_1, the key attributes are either retained in the view, or are equated to a constant in the view definition. Then, given a tuple t in the view, we can identify the tuple r of R_1 from the key attributes.

Corollary 6.5 *A SPJ view is self-maintainable with respect to deletions to R_1 if the key attributes from each occurrence of R_1 in the join are either included in the view, or are equated to a constant in the view definition.* $\Box$

The lesson learned from the theorems above is that self-maintenance of SPJ views is possible in the face of deletions, if some moderately restrictive conditions are satisfied.

Algorithm for self-maintenance on deletions: Given a view that is self-maintainable due to Corollary 6.5, we can self-maintain the view as follows: Given a deleted tuple r in relation R_1, check if the key of the deleted tuple satisfies any predicates that equate key attributes to constants in the view definition. Then, look for tuples in the view that have the same values on the remaining key attribute. Delete all such tuples from the view.
We illustrate the use of the above algorithm with two examples.

EXAMPLE 6.2 Consider the schema

`dept`$(\underline{D}, L)$ — *% Department D is at Location L.*
`emp`$(E, \underline{Id}, D, S)$ — *% Employee E with identifier Id works in department D has salary S.*

The above relations are used to define the view `work_at` as:

(V): `CREATE VIEW work_at AS` — *% List of employees that work at each location*
`SELECT` L, E, Id
`FROM dept`$(\underline{D}, L)$, `emp`$(E, \underline{Id}, D, S)$.

The view includes the key for relation `emp` but not for relation `dept`. Consider an initial view `work_at` $= \{(tokyo, e1, john), (tokyo, e2, tom), (ithaca, e3, sara)\}$. Now delete tuple $(john, e1, shoe, 100)$ from relation `emp`. It is simple to observe that tuple `work_at`$(tokyo, e1, john)$ needs to be deleted.

Consider, instead the deletion of tuple `dept`$(hat, tokyo)$. Now it is not possible to determine which tuples to delete from view `work_at`. The reason is that for a tuple t in view `work_at`, it is not possible to identify which tuple in `dept` derives t. Thus, deleted tuple `dept`$(hat, tokyo)$ may require that t = `work_at`$(tokyo, e1, john)$ be deleted given that "*john*" is in department "*hat*." Alternatively, deleting `dept`$(hat, tokyo)$ may have no effect on the view because "*john*" may be in some other department located in "*tokyo*." $\Box$

Consider another example where the key is not distinguished but the key can be inferred from the definition of the view.

EXAMPLE 6.3 In the schema of the above example, lets assume that the attributes E, S together form an alternate key for relation `emp`. That is, employees with the same name do not have the same salaries. On the relations `dept` and `emp`, with this new key, consider the following view:

(V): `CREATE VIEW work_at AS` *% List of employees that work at each location*
`SELECT` L, E *% and whose salary is* $= 100$
`FROM dept`($\underline{D}, L$) , `emp`($\underline{E}, Id, D, \underline{S}$)
`WHERE` $S = 100$.

Consider the initial view `work_at` $= \{(tokyo, john), (ithaca, tom)\}$ and let tuple `emp`$(john, e1, shoe, 100)$ be deleted. Given that there is only one "john" with salary "100" in relation `emp`, we infer that tuple `work_at`$(tokyo, john)$ is deleted. □

6.3 Updates

Recall from Definition 3.2 that a tuple r in relation R is said to be updated if one or more attributes of r are assigned values different from their original value. An update to tuple r that results in tuple r' is represented as $\mu(r, r')$.

Updates have been modeled as deletions followed by insertions in previous view maintenance work. While logically correct, representing an update in this manner may lose information that could otherwise be utilized for incremental view maintenance. Also, such a representation of an update means that a view is self-maintainable with respect to updates only if the view is self-maintainable with respect to both inserts and deletes.

In fact, the following lemma may suggest that the above conclusion regarding self-maintenance in response to updates, is the best one can do.

Lemma 6.1 *Let V be an SPJ view, and R one of the relations in the* `FROM` *clause, such that there is at least one predicate on R. Then, the view maintenance problem for V in response to any insertion or a deletion into R can be reduced to the view maintenance problem for V in response to an update to R.* □

However, from Lemma 6.1 we should not conclude that self-maintenance with respect to updates is harder than self-maintenance with respect to either inserts or deletes. By modeling an update as a delete plus an insert, we are losing the link between the deleted and inserted tuples; often most of the attributes of the tuple that needs to be inserted into the view can be obtained from the deleted tuple, enabling self-maintenance to be done.

By modeling an update as a third type of modification, we retain information about the tuple that is deleted from the view. This information then allows insertions to be handled easier in two respects: the conditions for self-maintenance becomes less onerous than in the case of simple insertions; and the computational effort required to perform the insertion is decreased, whether or not the view is self-maintainable. Thereby, we can often do view maintenance more efficiently than previously considered, and we can show that several views that are not self-maintainable with respect to insertions are self-maintainable with respect to updates.

EXAMPLE 6.4 Consider view `return_before` defined using (a vertical fragment of) relation `cs_phd` introduced in Example 2.1, and relation `borrowed` that has information about books borrowed from the library.

`CREATE VIEW return_before AS`
`SELECT` $Stu\_id, Book\_ID, Grad\_Dt, Due\_Dt$
`FROM cs_phd`($\underline{Stu\_ID}, Advisor, Grad\_Dt$) , `borrowed`($\underline{Book\_ID}, \underline{Stu\_id}, Due\_Dt$).

Let view `return_before` contain only tuple $(s1, QA76, 23\_June, 20\_Mar)$. Let tuple $(QA76, s1, 20\_Mar)$ in `borrowed` be updated to `borrowed`$(QA76, s1, 15\_June)$, *i.e.*, student "s1" extends the due date on book $QA76$ to 15_June.

If we treat this update as a deletion followed by an insertion then we can infer that tuple $(s1, QA76, 23\_June, 20\_Mar)$ is deleted from the original view. However, we are unable to infer the tuple that needs to be inserted, because there is no way to infer the graduation date of student "s1" without accessing the `cs_phd` relation.

On the other hand, the new view tuple can be obtained from the deleted one by merely changing the Due_Dt attribute from 20_Mar to 15_June. No reference to any underlying relation should be required. This section considers how to capture this intuition formally. □

We present algorithms for incremental maintenance of views in response to updates in the next two subsections. There are two components to any such algorithm, roughly corresponding, respectively, to deletion and insertion.

1. Which tuples, if any, in the current view are potentially affected by the update.
2. What is the effect of the update on these tuples?

6.3.1 Self-Maintenance

The ability to self-maintain a view depends upon the attributes being updated.

EXAMPLE 6.5 Consider a variant of view `return_before` that concerns itself only with books borrowed by students who will graduate before 15_Jun.

`CREATE VIEW return_before AS`
    `SELECT` $Stu\_id, Book\_ID, Grad\_Dt, Due\_Dt$
    `FROM cs_phd`$(\underline{Stu\_id}, Advisor, Grad\_Dt)$, `borrowed`$(\underline{Book\_ID}, \underline{Stu\_id}, Due\_Dt)$
    `WHERE` $Grad\_Dt < 15\_Jun.$

Consider an update to attribute Due_Dt of a tuple in `borrowed`. The updated attribute is not involved in any selection condition and therefore it is possible to self-maintain the view in response to the update just as in Example 6.4. Thus, we use the key of the updated tuple in `borrowed` to identify the tuple in `return_before` that needs to be changed, and update the Due_Dt attribute. If there is no tuple with the key value of the updated tuple, then the update has no effect on the view.

Instead, let us consider an update to a tuple in `cs_phd` on attribute $Grad\_Dt$. Let tuple `cs_phd`$(s1, John, 20\_Jun)$ be updated to `cs_phd`$(s1, John, 10\_Jun)$. We show that it is not possible to self-maintain this view. Say there is no tuple in view `return_before` that has $Stu\_id = s1$ as its first attribute. That is, before the update this student does not appear in any tuple in `return_before`. Now, using only the updated tuple and the view, it is not possible to update the view. Why? It may be that this student had not checked out any book and thus will not be in the view after the update. Alternatively, may be that the student had checked out a book and will now have to in the view because the graduation date has become $< 15\_Jun$. It is not possible to resolve the issue using only the update and the view. □

The following definition captures the property that enables self-maintenance.

Definition 6.1 (Variable Private to a Relation (VaPR):) Given a view definition V, a variable, or equivalently, an attribute, A, of a relation used in the view definition is said to be VaPR if it is not involved in any predicate. A variable that is not VaPR is called an *exposed* variable. □

A VaPR attribute does not affect the combinations of tuples that contribute to the view.

Theorem 6.6 *A SPJ view V with a join of two or more distinct relations is self-maintainable with respect to updates to a relation R_1 if and only if the updated attributes are VaPR with respect to view V, and either:*

- *updated attributes are not distinguished, or*
- *V is self-maintainable with respect to deletions to the relation R_1.*

$\Box$

Algorithm for self-maintenance on updates: The statement and proof of Theorem 6.6 yield the following algorithm for self-maintaining views in response to updates.

Algorithm 6.1
Input: View Definition
Update $\mu(r, r')$ for relation R_1
View V
Output Updated view V'
Method
If updated attributes are not distinguished **or**
If tuple r does not satisfy the local predicates
then do nothing.
Else
Let $\mathcal{T}$ be the subset of view V that equijoins with tuple r on the distinguished part of $\mathsf{key}(R)$.
FOR each tuple $u \in \mathcal{T}$ **DO**
Replace updated attributes of r in u by the new values in r'
$\diamond$

We illustrate the use of the above algorithm via Example 6.4.

EXAMPLE 6.6 The view relation `return_before` contains tuple $(s1, QA76, 23\_June, 20\_Mar)$. Then tuple `borrowed`$(QA76, s1, 20\_Mar)$ is updated to `borrowed`$(QA76, s1, 15\_June)$. The view and the update satisfy the conditions of Theorem 6.6: (a) The key of the updated tuple in relation `borrowed` can be used to identify the updated tuple in the view, and (b) the updated attribute *Due_Dt* is VaPR. Thus the view is self-maintainable.

There are no local predicates on `borrowed` and attribute *Due_Dt* is distinguished. Thus, the algorithm proceeds to the phase where the effect of the update is reflected on the view.

The result of joining `return_before` with `borrowed`$(QA76, s1, 15\_June)$ on attribute *Stu_id* yields only tuple $(s1, QA76, 23\_June, 20\_Mar)$. In this tuple, the last attribute *Due_Dt* is changed to reflect the updated value yielding tuple $(s1, QA76, 23\_June, 15\_June)$. $\Box$

6.3.2 Updates to Exposed Variables

Even if the conditions of Theorems 6.6 are not satisfied, it is advantageous to model updates in the way we have done. In this section we discuss how to self-maintain views when exposed variables are updated. Since these variables are not VaPR they cannot be handled by Algorithm 6.1.

First, we distinguish between exposed variables that are used in local predicates and variables that are used in join predicates.

Definition 6.2 (Locally and Globally Exposed Variable:) An exposed variable, or attribute, A, of a relation used in the view definition V is said to be *locally exposed* if A is not involved in any join predicate. Otherwise, it is said to be *globally exposed.* $\Box$

Intuitively, a locally exposed variable participates in a selection condition local to a particular relation, and does not affect the combinations of tuples joined with from other relations.

Theorem 6.7 *An SPJ view V with a join of two or more distinct relations is self-maintainable with respect to updates to a locally exposed attribute of relation R_1 if and only if*

- *V is self-maintainable with respect to deletions to the relation R_1.*
- *There exists at least one tuple t in V that is derived using each updated tuple in V.*

□

The algorithm for self-maintenance for views that satisfy the conditions of Theorem 6.7 is similar to Algorithm 6.1.

When the updated attribute is involved in a join predicate, the view is not self-maintainable. We have derived an algorithm to maintain views in response to such updates that is more efficient than algorithms that treat an update as a delete followed by an insert. The savings arise by repeating only the joins in which the updated attribute is involved.

6.3.3 Complex Updates

Formally, an update is applied to a single tuple, and causes a change in the value of some attributes. However, from the preceding discussion, it is clear that the important property of an update is not so much that there is change to a single tuple as that there is relevant information available regarding the "newly inserted" updated tuple. Having made this observation, there is no reason to require that only one tuple be so "newly inserted" at a time. Thus one could have a *complex update* applied to a relation in which one tuple is deleted and instead a number of tuples inserted that match the deleted tuple, and one another, on all attributes except the ones being updated. Complex updates can be self-maintained in a manner similar to single updates.

EXAMPLE 6.7 Consider the same schema and initial state as in Example 6.4, *i.e.*, the original view contains tuple $(john, QA76, 23\_June, 20\_Mar)$. Now suppose that `borrowed`$(QA76, john, 20\_Mar)$ is deleted and replaced by `borrowed`$(QB77, john, 15\_June)$ and `borrowed`$(QB78, john, 15\_June)$, *i.e.*, John returned book $QA76$, and instead borrowed books $QB77$ and $QB78$. As before, if we treat this as a deletion followed by two insertions, we do not have enough information to self-maintain the view. However, if we consider this a complex update, then we are able to update the view. □

6.4 Summary

An SPJ view is seldom self-maintainable in response to insertions. For the view to be self-maintainable in response to deletions and updates, it is useful to retain the key of the modified relation in the original view. Even then, we may need to go to base relations when exposed variables are updated and the updated tuple does not already contribute to the view.

7 Self-Maintainable Outer-Join Views

Outer-Join (OJ) views contain more information about the underlying base relations than views defined using regular joins. In particular, OJ views contain information about tuples that do not contribute to the answer in a regular view. This property makes OJ views more self-maintainable than SPJ views.

We first consider self-maintenance for general outer-join views, and then give a specialized self-maintenance algorithm for `match` views.

7.1 Self-maintaining full outer-join views

The attempt of the subsequent discussion is to show how we can replace relations R, S, R^ν, and S^ν in Algorithm 4.1 by functions of view V so that we do not need to go to the base relation but can instead derive the relevant information from the view itself. Thus, we avoid accessing the actual relations R and S.

Theorem 7.1 *An outer-join view V defined using two relations R and S, such that:*

- *The keys of R and S are distinguished, and*
- *All exposed attributes of R are distinguished.*

is self-maintainable with respect to all types of modifications to relation S. □

Algorithm for self-maintenance: Algorithm 4.1 is used in this context directly with the following alterations. Query (a) does not derive anything because $\Delta(R)$ is empty. Query (b) is computed by generating R^ν from V^ν instead of accessing relation R. R^ν is defined as $\pi_{R_v}(V)$ where R_v is the set of those attributes of R that are part of view V. Note, because all the non-VaPR attributes of R are distinguished, the projection has all the attributes required for computing the outer-join predicate. In addition, all tuples that have any NULL attribute in $\pi_{R_v}(V)$ are not a part of R^ν.

EXAMPLE 7.1 We use the view `student` from Example 4.1 to illustrate how an OJ view is self-maintained. We modify only relation `registrar`. Thus, in the language of Theorem 7.1, R is `registrar` and S is `cs_phd`. The requirements of the theorem are satisfied by view `student`. Namely, the keys of both relations are distinguished, and the non-VaPR attribute of `registrar`, *Stu_id*, is distinguished. We use the same initial database as in Example 4.1, and the same set of changes to relation `registrar`. Thus:

`cs_phd`$= \{(s1, adv1), (s3, adv3)\}$,   `registrar`$= \{(s1, john), (s2, mary)\}$.
`student`$= \{(s1, s1, john, adv1), (NULL, s2, mary, NULL), (s3, NULL, NULL, adv3)\}$.
$\Delta^+(\texttt{registrar}) = \{(s3, sara)\}$, $\Delta^-(\texttt{registrar}) = \{(s1, john)\}$.

Query (a) does computes an empty answer because $\Delta(\texttt{cs\_phd}) = \{\}$. Thus, only query (b) needs to be computed. Query (b) right-outer-joins $\texttt{cs\_phd}^\nu$ with $\Delta(\texttt{registrar})$. $\texttt{cs\_phd}^\nu$ is the same as `cs_phd`. The vertical fragment of $\texttt{cs\_phd}^\nu$ used in query (b) is obtained from view `student` as follows:

$\texttt{cs\_phd}^\nu = \pi_{Stu\_id1,Adv}(\texttt{student}) = \{(s1, adv1), (s3, adv3)\}$.

Using relation $\texttt{cs\_phd}^\nu$ thus obtained, we proceed as in Example 4.1.

$\Delta^+(\texttt{student}) = \{(s3, s3, sara, adv3)\}$, $\Delta^-(\texttt{student}) = \{(s1, s1, john, adv1)\}$.
$\texttt{student}^\nu = \{(NULL, s2, mary, NULL), (s3, NULL, NULL, adv3), (s3, s3, sara, adv3)\}$.

As a side effect of inserting $(s3, s3, sara, adv3)$ into `student` the tuple $(s3, NULL, NULL, adv3)$ is removed from the view. Similarly, as a side effect of deleting tuple $(s1, s1, john, adv1)$, we need to check if the pair $(s1, adv1)$ appears in any other tuple in the view `student`. It does and thus we need to insert $(s1, NULL, NULL, adv1)$ into the view. Thus, the final view is:

$\texttt{student}^\nu = \{(s1, NULL, NULL, adv1), (NULL, s2, mary, NULL), (s3, s3, sara, adv3)\}$.
□

7.2 Self-Maintaining match Views

A match view is a "natural full outer join" view that may involve multiple relations that are equijoined over their keys, and the join attributes are distinguished. Therefore, it would seem that due to Theorem 7.1 a two-way match operator is self-maintainable. However, the following example illustrates that this is not always the case because a match is a "natural" full outer join, *i.e.*, only the non-null values of the join attributes are retained in the view.

EXAMPLE 7.2 Consider the following match view:

CREATE VIEW P AS
    SELECT B
    FROM $R(\underline{B})$ $|\&|$ $S(\underline{B})$.

Consider the following two alternative values for relations R and S and the resulting value for view P:

$(DB1)$: $R = \{(b)\}$, $S = \{(b)\}$, $P = \{(b)\}$.
$(DB2)$: $R = \{(b)\}$, $S = \{\}$, $P = \{(b)\}$.

Looking at only the view P, it is not possible to distinguish database $DB1$ from $DB2$. Thus, if tuple $R(b)$ were to be deleted, then it is not possible to self-maintain P because if the underlying database is $DB1$ then after the deletion the final view should be $P = \{(b)\}$, and if the underlying database is $DB2$ then after the deletion the final view should be $P = \{\}$. □

The problem in the above example arises because after naturalization, it may not be possible to identify the relations that contribute non-NULL attributes to tuples in the view. If the view is such that for each tuple in the view it is possible to identify if an underlying relations contributes a joining tuple, then self-maintenance becomes possible. It is possible to identify joining tuples if a non-key, and hence nonjoin, attribute is present from each relation. In particular, we state the following theorem.

Theorem 7.2 *A n-way* match *view V is self maintainable with respect to all modifications if and only if each underlying relation has at least one distinguished non-key attribute.* □

Thus, a two-way match view that satisfies the condition of the above theorem, is self-maintainable. Henceforth, we consider only match views that satisfy the condition of Theorem 7.2. Now, consider a multi-way match. Since the match operation is associative, we can write this as a two-way match between the modified relation, and a match of all the other relations. (It is not useful for a relation to appear in a match more than once). By Theorem 7.2, this two-way match is self-maintainable. Therefore, a multi-way match operation is self-maintainable. Below we discuss how to self-maintain multi-way match views when only one of the underlying base relations is modified. The extension to multiple base relation updates is straightforward.

Note, each tuple in an underlying base relation joins with exactly one other tuple in any other relation, and derives exactly one tuple in the match view. The identity of the base relation tuple that derives a tuple in the view, can be determined from the distinguished attribute(s) of the view. Updates to non-key attributes of a tuple now become particularly easy to maintain. Simply find the corresponding match view tuple and change the updated attributes. Updates to key attributes, insertions, and deletions are a little more involved to maintain. The algorithm given below discusses how to self-maintain match views for these modifications.

Algorithm 7.1
Input: A n-relation match view definition C that uses relation R
Old value of view V.
Set of insertions, deletions, updates to R

Output: New view V^ν
Method
% We describe how to handle updates, insertions, and deletions one *tuple at a time.*
% Handle Updates to Key attributes.

Let tuple r in R be modified to r' by changing key K from k to k'.
Let tuple v in view V have key value k
Let tuple v' in view V have key value k'. *% There may not be such a v'*
Let $D(R)$ be the set of distinguished non-key attributes of R.
In tuple v, set the value of attributes $D(R)$ to NULL. (d)
IF all non-key attributes of v are NULL, **THEN** remove v from view V. (d)
IF tuple v' exists, **THEN** (i)
Set the value of attributes $D(R)$ in v' to the corresponding values in r'. (i)
ELSE (i)
insert into V a tuple formed by padding r' with NULL values for all distinguished (i)
attributes of relations other than R (i)

% Handle Insertions by treating them like updates to Key attributes

Let inserted tuple be r' with key value k'.
Let tuple v' in view V have key value k'. *% There may not be such a v'*
Execute all lines marked "(i)" from update section.

% Handle Deletions by treating them like updates to Key attributes

Let deleted tuple be r with key value k.
Let tuple v in view V have key value k.
Execute all lines marked "(d)" from update section.

$\diamond$

EXAMPLE 7.3 Consider the `match` view `student` described in Example 4.1.

CREATE VIEW student AS
SELECT $Stu\_id, Name, Advisor$
FROM cs_phd$(Stu_id, Advisor)$ $|\&|$ registrar$(Stu_id, Name)$.

We illustrate updates to non-key and to key attributes. Consider an initial database and the resulting value for view `student`.

cs_phd= $\{(s1, adv1), (s3, adv3)\}$.
registrar= $\{(s1, john), (s2, mary)\}$.
student= $\{(s1, john, adv1), (s2, mary, NULL), (s3, NULL, adv3)\}$.

First we consider an update to a non-key attribute. Say tuple r = cs_phd$(s1, adv1)$ is updated to r' = cs_phd$(s1, adv5)$. The view is updated by identifying the view tuple with $Stu\_id = s1$, and then setting the value of its attribute $Advisor$ to $adv5$. Thus, the resulting view is:

student= $\{(s1, john, adv5), (s2, mary, NULL), (s3, NULL, adv3)\}$.

Now, consider an update to a key attribute, *i.e.*, update tuple r = registrar$(s2, mary)$ to r' = registrar$(s3, mary)$. Note, the set of non-key distinguished attributes, D(registrar), is $\{Name\}$. First, we find the `student` tuple v that has $Stu\_id = s2$. For this tuple, we set attribute $Name$ to NULL. The resulting tuple is $(s2, NULL, NULL)$. This tuple is deleted because all its non-key attributes are NULL. Next, we look for tuple v' that has $Stu\_id = s3$, and find $v' = (s3, NULL, adv3)$. To obtain the new value of tuple v' we set the value of attribute $Name$ to that from tuple r' namely $mary$. The resulting view is:

student= $\{(s1, john, adv5), (s3, mary, adv3)\}$.

$\square$

We note that the ability to update the key value of a tuple is very important in data integration applications. For example, consider an organization that has information about customers dealing with two parts of the company in two separate databases. When integrating the database, two customers in separate databases may not be identifiable as the same customers, and are given distinct customer identifiers. However they may get identified as the same customer later, leading to a change in the customer identifier attribute in one database, while retaining all the other attributes.

8 Related Work

The problem of incremental view maintenance has received a lot of attention in the literature [BLT86, CW91, GMS93, HD92, Kuc91, UO92]. The view maintenance work has typically looked at select, project, and join operations. In deductive databases, recursion has been the focus. Recently, aggregation and negation have also been discussed [GMS93]. There are several differences between the current work and the above papers.

First, view maintenance of outer-join views has not been studied.

Second, view maintenance has been studied as a problem to compute the changes to the view given the changes to the underlying relations, while having access to the view and the underlying relations. The problem of self-maintenance, where the view must be maintainable without accessing the underlying relations has not been discussed much. Tompa and Blakeley [TB88] deduce conditions, that if satisfied by the current extent of an SPJ view, would make the view self-maintainable in response to insertions or deletions. They do not identify classes of views that are self-maintainable independent of the particular instance of the view, and they do not consider updates separately from deletes + inserts. However, their techniques were helpful to us in developing an understanding for the self-maintenance problem.

Third, the previous work has modeled updates as deletions + insertions. In this paper we model updates directly, and show that doing so leads to more efficient view maintenance techniques and better self-maintenance properties.

The view update problem [Kel86] is how to translate an update to a view onto updates to the base relations. It is the dual to the view maintenance problem. Clearly the view update problem is relevant to the data integration applications, as one may want to move some of the provisioning processes onto the integrated database.

9 Conclusions

In this paper we consider the problem of view maintenance in the framework of data integration applications. We propose that materialized views be used to integrate data and thus provide fast access to data that would otherwise have to be remotely obtained. We define self-maintenance as an important property for views used to integrate databases. A view is said to be self-maintainable with respect to a particular type of modification (insert, delete, or update), if the view can be maintained in response to that type of modifications to a base relation without accessing any of the base relations. Having self-maintainable views allows the integrated database to guaranty that any modifications of that type to the base relations can be applied to the integrated view as soon as the modifications are made available. Further, problems relating to remote accesses, locking, distributed two-phase commits, and a variety of concurrency control problems [Z+94] can be avoided.

We observe that the outer join is an important operator in many integrated view definitions. We identify a special type of outer-join, called the *match* operator that appears to occur very commonly in data integration applications. A match is essentially a natural full outer-join on the key attributes of the relations. It is an associative operator, so n-way match joins can be defined.

Having set the framework, we develop solutions to the following problems:

- How to incrementally maintain outer-join views?
  We give incremental maintenance algorithms for full outer-join, and its variants, the left, right, and match join operations.

- What are the tightest conditions under which a view can be self-maintained? What are the syntactic restrictions that ensure self-maintainability?

  - match views are self maintainable if one non-key attribute of each joined relation is included in the view.
  - Full outer-join views are self-maintainable if the keys and the exposed variables of the joined relations are included in the view.
  - SPJ views are self-maintainable with respect to deletions and updates to VaPR attributes if the keys of the joined relations are included in the view.
  - SPJ views are usually not self-maintainable with respect to insertions.

- How to handle updates during view maintenance.
  We show that modeling updates directly rather than as deletions + insertions makes view maintenance easier, and can make some views self-maintainable with respect to updates.

We recommend that integrated views be defined so as to be self-maintainable for as large class of modifications as possible.

References

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

[BW89] T. Barsalou and G. Wiederhold. Knowledge-Based Mapping of Relations into Objects. In *Proceedings of Computer Aided Design*, 1989.

[CV92] S. Chaudhuri and M. Vardi. On the Equivalence of Datalog Programs. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 55–66, San Diego, CA, 1992.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *17th VLDB*, 1991.

[Dat89] C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Massachusetts, 1989.

[GB94] Ashish Gupta and J. A. Blakeley. Maintaining Views using Materialized Views . In *unpublished document*, 1994.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, pages 157–167.

[GSUW94] Ashish Gupta, Shuky Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint Checking with Partial Information. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS)*, 1994, pages 45-55.

[HD92] John V. Harrison and Suzanne Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, 1992.

[Kel86] A. Keller. The Role of Semantics in Translating View Updates. In *IEEE Computer,* 19:1, January 1986, pages 63-73.

[Kuc91] V. Kuchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *DOOD, LNCS 566*, 1991.

[ISO92] ISO. Database Language SQL ISO/IEC. Document ISO/IEC 9075:1992. Also available as ANSI Document ANSI X3.135-1992, 1992.

[LW90] B.S. Lee and G. Wiederhold. Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views. Technical report, CIFE – Stanford University, May 1990.

[MPP+93] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Sudkamp. SQL/XNF - Processing Composite Objects as Abstractions over Relational Data. In *Proceedings of the Ninth IEEE International Conference on Data Engineering*. 1993.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases*. Ph.D. Thesis, Stanford University, Stanford, CA 94305, USA, 1991.

[RGL] Arnon Rosenthal and Cesar Galindo-Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins.

[RSUV89] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Vardi. Proof-tree Transformation Theorems and Their Applications. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, 1989.

[Shm87] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 237–249, San Diego, CA, March 1987. ACM SIGACT-SIGMOD-SIGART.

[TB88] F. W. Tompa and J. A. Blakeley. Maintaining Materialized Views without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume 2. Computer Science Press, New York, 1989.

[UO92] Toni Urpi and Antoni Olive. A Method for Change Computation in Deductive Databases. In *18th VLDB*, pages 225–237, 1992.

[Z+94] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View update anomalies in a warehousing architecture. unpublished manuscript, October 1994.

A Extended Relational Systems

Complex applications, such as design applications and advanced business applications, require complex object support in conjunction with access to traditional relational data. Thus there has been effort to develop tools that provide complex objects either on top of relational systems [BWS89, LW90] or integrated into the relational system [MPP+93]. Outer joins are used in the emulation of objects in relational systems because the outer join uses NULLS to provide a mechanism of preserving tuples that do not join with other tuples. This is an effective way of capturing NULL valued attributes in objects. The following example illustrates this notion. This example is similar to examples in [LW90, MPP+93].

EXAMPLE A.1 Consider a database that has the following relations:

dept($\underline{D}, L, M$) % *Department D is located at L and has manager M.*
emp($\underline{E}, D, S$) % *Employee E is in department D and has salary S.*
proj($\underline{P}, D$) % *Project P is being managed by department D.*
works_in($\underline{E}, P$) % *Employee E works for Project P.*

Let us build an object for each department that has as attributes the employees in that department and the projects being managed by the department. The object can be built as follows:

```
CREATE VIEW dept_obj AS
        SELECT D, L, M, E, S, P
        FROM (dept(D, L, M) |& emp(E, D, S)) |& proj(P, D).
```

Each top level instance of object `dept_obj` "owns" the lower objects as children. There may be departments that have either no employees or no projects. " $|\&$ " stands for the left outer join operator, and it ensures that departments are present in the result even if they do not have any employees or any projects. The parenthesis indicate the order in which the left-outer-joins are done.

Similarly, we could create an object for each project and have it "own" all salaried employees that work for the project and their salaries. Note, the definition of this object uses a regular join in addition to a left-outer-join.

CREATE VIEW proj_emp AS
  SELECT P, D, E, S
  FROM proj(P, D) $|\&$ (emp(E, D, S) , works_in(E, P)).

□

B Maintaining Left and Right Outer-Join Views

We explain the maintenance process using only left outer-join views. Right outer joins are similarly handled. Left outer-join views can be maintained by using a variant of Algorithm 4.1. Consider the following left outer-join view:

(C): CREATE VIEW V AS
  SELECT $\bar{X}$
  FROM $R(\bar{Y})$ $|\&$ $S(\bar{Z})$
  WHERE $g(\bar{Y}, \bar{Z})$.

The delta queries for the above left-outer-join view defined above are as follows:

(a): SELECT $\bar{X}$
  FROM $\Delta(R)(\bar{Y})$ $|\&$ $S(\bar{Z})$
  WHERE $g(\bar{Y}, \bar{Z})$

(b): SELECT $\bar{X}$
  FROM $R^{\nu}(\bar{Y})$, $\Delta(S)(\bar{Z})$
  WHERE $g(\bar{Y}, \bar{Z})$.

Note, query (b) has changed as compared to the delta query for full outer-join views: it uses a regular join instead of an outer-join. The change to the algorithm is that all the rows marked "@" are removed. That is, the potential_deletions and potential_additions sets are not maintained (and therefore not manipulated) when query (a) is applied.