Survey Paper

for

CS748T
Distributed Database Management
Lecturer: **Prof. M. Tamer Özsu**

Semantic Data Control in Distributed Database Environment

by

Lubomir Stanchev
#99800760

February 2001
University of Waterloo

Abstract

Semantic data control is responsible for keeping a database valid relative to a specified set of constraints. According to [ÖzVa99], it includes view maintenance, semantic integrity control and security control. In this survey paper we will examine the first two characteristics relative to a distributed database environment.

View maintenance deals with synchronizing relevant data in a database environment. For example the results from often executed queries may be stored at one or more sites in order to reduce the query response time. We will refer to such stored query results as *materialized views* and to the data on which the queries are posed as the *underlying* or *base data*. The synchronization problem is to insure that, when the underlying data is updated the updates are propagated to the materialized view (this problem is called *materialized view maintenance*) and that the reverse also holds - i.e. when the materialized view is updated the updates are propagated to the underlying data (this problem is called *materialized view update)*. Note that, while there is always a unique solution to the first problem, there may be more than one ways to update the underlying data relative to a materialized view update.

Semantic integrity control relates to the problem of keeping the distributed data valid relative to a pre-specified set of integrity constraints. For example when the data is updated, we must make sure that the specified integrity constraints continue to hold on the data after the update. One type of algorithms for performing integrity control, validates that updates will preserve the integrity constraints before executing them. The other type of algorithms executes any updates and then checks to see if the integrity constraints are violated, and if they are, the system tries to perform a minimal modification on the database, such that the integrity constraints will be restored.

In this survey paper we will introduce a formal structure to the problem of semantic data control and will examine existing algorithms for materialized view maintenance, materialized view update and for the two types of semantic integrity controls approaches.

1. Introduction

1.1 Background

1.1.1 Constraints in Distributed Databases

A distributed database is a database distributed between several sites. The reasons for the data distribution may include the inherent distributed nature of the data or performance reasons. In a distributed database the data at each site is not necessarily an independent entity, but can be rather related to the data stored on the other sites. This relationship together with the integrity assertions on the data are expressed in the form of data constraints. Figure 1 shows a classification of the possible data constraints that can hold on a distributed database. We can think of those constraints as an invariant, which must

hold at any given time instance (static constraints) or during any time interval of pre-specified length (dynamic constraints).

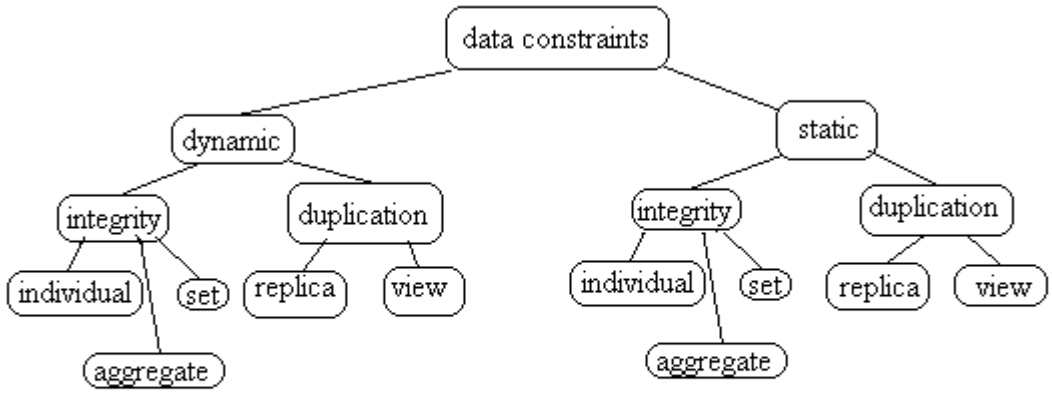

Figure 1. Distributed database data constraint classification

At the top level there are two type of constraints dynamic and static. To check whether a *static constraint* holds, at any given time point, we need a static snapshot of the database instance. On the other hand, to verify whether a *dynamic constraint* holds we need to have information about how the database instance evolves over time. An example of a dynamic constraint is: within every 5 minutes of the life of the database instance something must happen, e.g. an update must occur.

On an orthogonal plane, data constraints can be integrity or duplication. *Integrity* constraints are independent of how the data is distributed or duplicated If an integrity constraint is based on a single collection of data objects and a single variable it is called *individual*. *Set constraints* are those that are based on more than one data collections or on more than one variables. For example, in the distributed database shown in Figure 2, individual integrity constraints may include the primary key constraints on the tables R_1 and R_2, whiles a set integrity constraint may include a referential foreign key constraint between the two tables. *Aggregate constraints* are integrity constraints involving aggregate operators such as min, max, sum, count and average. The cash present at every supermarket, at any time instance, should be less than $10,000 is an example of a static individual aggregate data constraint on the distributed database of a chain-store supermarket.

Note, that integrity constraints can be both dynamic and static. In particular a *dynamic integrity constraint* may state that if an update, which violates a constraint, is performed, a compensating update which restores the constraint should be performed within two minutes. On the other hand, static integrity constraints require the integrity constraint to be true at any given time instance.

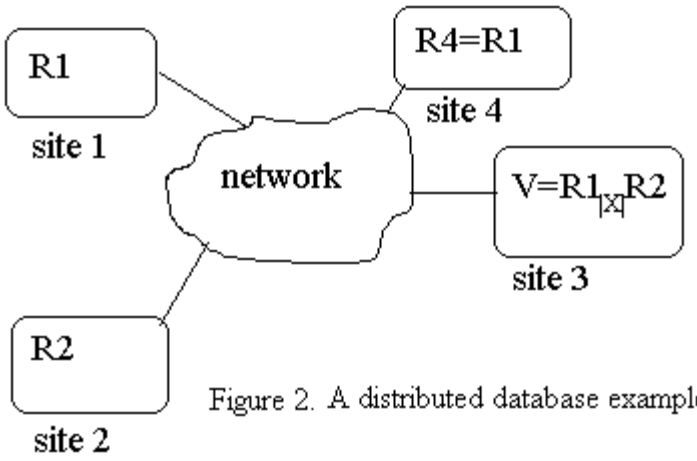

Figure 2. A distributed database example

.

On a orthogonal plain, a *duplication constraint* specifies at what sites *replicated* or *derived* data is to be stored. We will refer to the first type of duplication constraints as *replica constraints* and to the second type as *view constraints*. In the example of Figure 2, the constraint that the table R_1 is to be replicated at *site 4* as R_4 is a replica constraint, while the constraint that *site 3* contains a table with derived data from tables R_1 and R_2 is a view constraint. From now on, we will refer to the stored query result over existing data sets as a *materialized view*. In the example from Figure 2, *V* is a materialized view because it stores the result of the query which performs a join between the tables R_1 and R_2. We will refer to the tables R_1 and R_2 as the *underlying tables* of the materialized view *V*.

Note that replica constraints are a special kind of view constraints. This is the case because a replica constraint can be expressed as a view constraint in which the query producing the view is the identity query. As well, note that duplication constraints can be both dynamic and static. In the example of Figure 2, a dynamic constraint may state that if the table R_1 is updated, the view *V* has to be updated accordingly within five minutes to reflect the changes.

There is another type of constraints related to distributed databases and databases in general - *update constraints*. Those constraints specify what kind of updates are allowed on the data. More specifically an update constraint consists of a triple (P^+,P^-,P^{+-}) associated with each collection of data items, where each of the P^s is a predicate formula with at most one free variable, specifying what type of add, delete and modify operations respectively are allowed on the collection. Note that, in particular the P^s may involve part of the database instance as a parameter. For example, suppose we have the tables $R_1(\underline{A},B)$ and $R_2(\underline{B},C)$ where $R_1.B$ is a foreign key pointing to the primary key R_2.B. We can insure that referential integrity constraint: $\forall t_1 \in R_1$ there $\exists t_2 \in R_2$, s.t. $t_1.B=t_2.A$ (this is a static integrity set constraint) holds by imposing the update constraint $(\exists t_2(t_1.B=t_2.B)$, TRUE, t_1(new).B = t_1(old).B $\vee$ $\exists t_2(t_1(\text{new}).B=t_2.B))$ on the table R_1 and the update constraint (TRUE, $\neg\exists t_1(t_1.B=t_2.B)$, t_2(new).B = t_2(old).B $\vee$ $\neg\exists t_1(t_1.B=t_2(\text{new}).B))$ on table

R. Note that the free variable in the predicate expressions is bound to the tuple which is inserted/deleted from the tables and *t*(new) and *t*(old) are used to denote respectively the new and the old value of the tuple being updated, where *t* is the free variable. As seen from the above example, a characteristic of update constraints is that they can be used to guarantee that certain data constraints hold after an update of the allowed type is performed, provided that this was the case before the update was executed.

Note that, just like data constraints, update constraints can be dynamic (in the previous paragraph we introduced static update constraints). A *dynamic update constraint* is a quadruple (T, P^+,P^-,P^{+-}), where T is a time constraint saying when the static update constraint (P^+,P^-,P^{+-}) should start holding on the database instance. An example of a time constraint is within 5 minutes of the update. Figure 3 shows a visual classification of the possible update constraints. Note that, as shown in the figure, update constraints may be imposed on a single update or on a series of updates designated in a transaction. In the first case we will talk about a *single update constraint* and in the second about a *multiple updates constraint*.

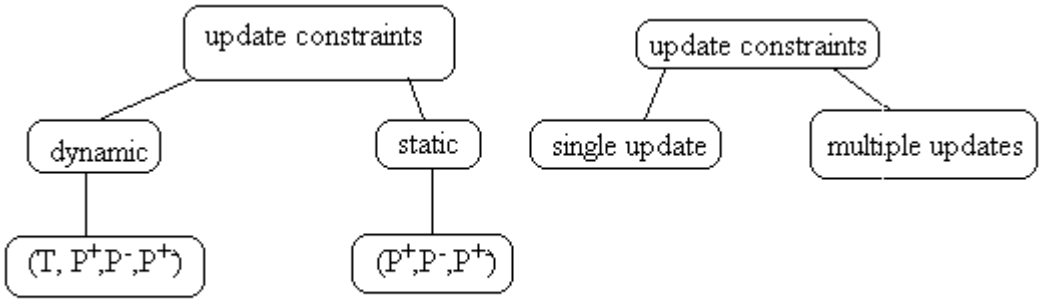

Figure3. Update Constraints Classification

In this paper, we will focus only on how static update constraints can be used to enforce data constraints. There are two reasons on which this decision is based. First, a dynamic constraint of the type: *if (P^+,P^-,P^{+-}) doesn't hold after an update, another update will be perform within a specified interval of time so that after the second update the predicate condition will hold* can be considered a multiple updates constraint on a transaction which contains the above two updates. The second reason, why we avoid exploring dynamic update constraints, is to avoid unnecessarily complications in the reasoning to follow.

So far, we have defined the model we will be working with, i.e. how we interpret a distributed database, what type of constraints we expect to be imposed on its data and what are update constraints. In the next sections, we will give background on the problem of how duplication data constraints and integrity constraints can be imposed through the use of synchronization algorithms and update constraints.

1.1.2 Imposing Constraints

Let's start by introducing some terminology relevant to duplicate constraints. In general there are two type of algorithms for imposing such constraints - *view maintenance algorithms* and *view update* algorithms. The first type of algorithms are used to propagate changes made to the underlying data sources to the corresponding materialized view. The view update algorithms are used to do the reverse, i.e. to propagate changes made to a materialized view to its underlying data sources. Both type of algorithms are called *synchronization algorithms* because they are used to synchronize the data in a distributed database environment. Depending, on whether the synchronization is done immediately during each update or some time in the future, we speak of *immediate* and *deferred* data synchronization algorithms. Table 1 summarizes the different synchronization algorithms for imposing duplication constraints, together with the necessary update constraints.

constraint / operation	static update constraints to be imposed	imposing dynamic duplicate constraints	imposing static duplicate constraints
update to a materialized view	integrity constraint preserving + guaranteeing feasible translation to the underlying data	through deferred materialized view update algorithm	through immediate materialized view update algorithm
update to underlying data sources	integrity constraint preserving	through deferred materialized view maintenance algorithm	through immediate materialized view maintenance algorithm

Table 1. Shows how duplication constraints are imposed.

One thing the table shows that when an update(s) to a materialized view occurs, for this/those update(s) to be accepted, it must be the case that there is a feasible translation of the update(s) to the underlying data sources, which can be guaranteed through a static single/multiple update constraint. Informally, a feasible translation of an update to the underlying data sources is a translation that propagates the changed made to the materialized view to the underlying data in a "well-found" manner relative to a specified condition. We will examine this subject in details in Section 3.

The table also shows that it must also be true that, either the replica constraints specified on a database, after an update is performed, are preserved or that they will be somehow restored shortly. As well, it must be the case that a database is valid relative to the specified integrity constraints before the synchronization algorithms can start working because they rely on the later. Note as well, that we didn't consider replica constraints because they are a special kind of duplicate constraints. Existing synchronization algorithms for materialized view maintenance will be explored in Section 2. The synchronization algorithms for materialized view update will be explored in Section 3.

Table 2 shows how static and dynamic integrity constraints can be imposed. One way static integrity constraints can be imposed is by specifying appropriate static update constraints. This process is called constraint compilation and will be examined closer in Section 4.

imposing static integrity constraints	**imposing dynamic integrity constraints**
through update constraints	by applying a deferred integrity constraint restoring algorithm
by applying an immediate integrity constraint restoring algorithm	

Table 2. Shows how integrity constraints are imposed

Another way of imposing static integrity constraints is to allow for any kind of updates on the database instance and try to restore the integrity constraints when they become violated using *integrity constraint restoring algorithms*. Such algorithms usually work by trying to find a "minimal", relative to some criteria, update to the database instance that will restore the integrity constraints and then perform it. Integrity constraint restoring algorithms will be examined in Section 5.

Imposing dynamic integrity constraints can be done by applying deferred integrity constraint restoring algorithms or through dynamic update constraints. In this paper we will examine only the first method and this will be done in Section 5.

To finish this introductory discussion of constraint enforcing, let's look at the example shown in Figure 4.

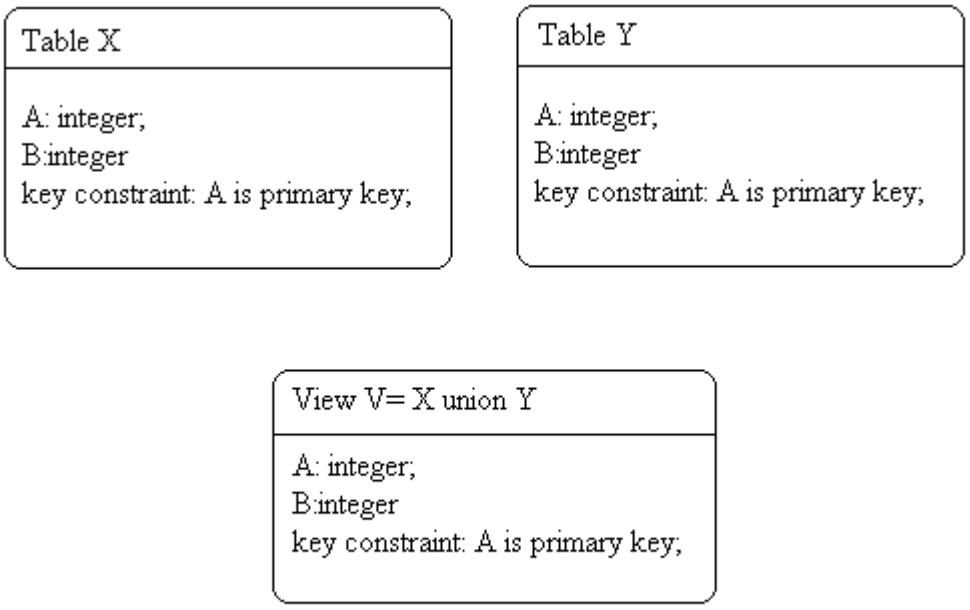

Figure 4. An example database instance

Consider the case, where table *X* contains the tuple (1,2), the tables Y is empty and the table *V* contains the tuple (1,2). Now, consider the insertion of the tuple (1,3) to the table *Y*. A synchronization algorithm will propagate this change to *V*, i.e. the new instance of *V* will be {(1,2) ,(1,3)}. But then, the integrity constraint on *V* will be violated. If we use an integrity constraint restoring algorithm, it may delete one of the tuples (1,2) or (1,3) from the tables *X* or *Y* and as well, the corresponding tuple in *V*. This is because the algorithm may decide that removing one of the tuples is the way to restore the integrity constraints in the database through minimal change.

Another way of imposing the validity of the database is to use update constraints, corresponding to the defined integrity constraints. Note that, if this is the case, an update should not be committed until all updates resulting from it are successfully propagated. In our example, if we try to insert (1,3) in *Y*, a synchronization algorithm will try to propagate the change to *V*, but then the update constraint, which guarantees the validity of *V*, will be fire and the original update to *Y* should be aborted.

1.2 Applications

Most applications working on top of distributed databases exploit materialized views to produce fast access to derived data and lower CPU, disk and network loads. Examples of traditional distributed database applications that use materialized views include banking, billing, network management, distributed query optimization, integrity constraint checking and switching software. More recent applications in this area include data warehousing, OLAP, data replication, data visualization, mobile system and distributed situation monitoring applications. The presented list is by no means complete and it is very likely that some of the newly emerging software technologies will also rely on materialized views to boost performance.

1.2.1 Data Warehouse Example

Generally speaking, a *data warehouse* contains aggregated data derived from a number of data sources and is usually used by *On-Line Analytical Processing* (OLAP) tools and datamining tools for the purpose of decision support (see Figure 5).

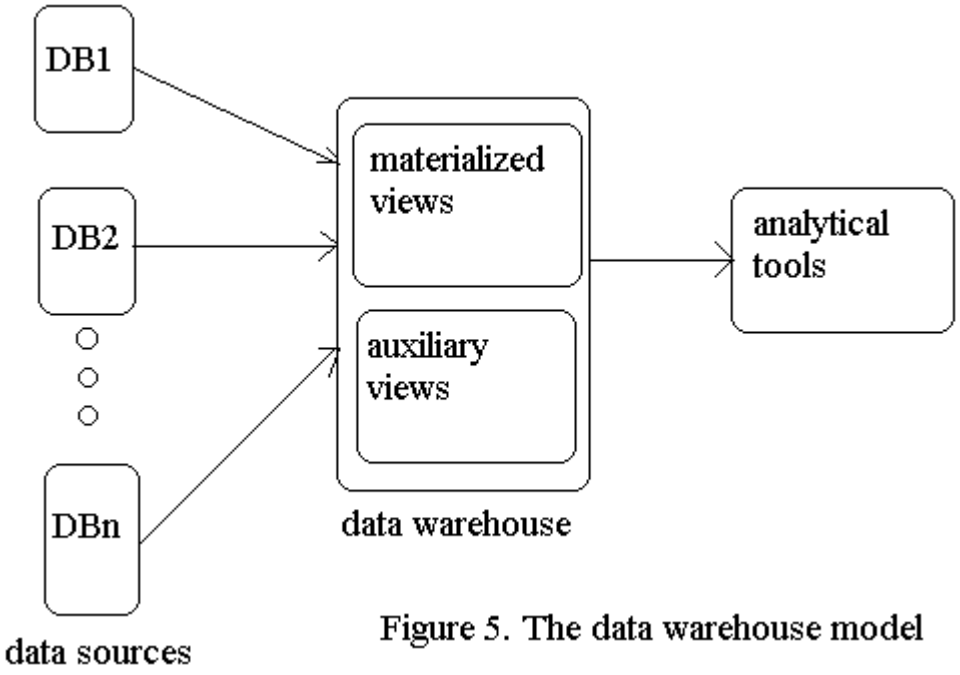

Figure 5. The data warehouse model

The *data sources* consist of several databases, usually containing huge amount of data. The day-to-day transactions of a store-chain, for some period of time, is a typical example of the kind of data stored in the data sources. The *materialized views* on the other hand contain summary data compiled from several data sources. For example the result of retrieve queries may be cached onto the materialized views in order to achieve faster response time to user requests. The auxiliary views in the picture are not mandatory, and are used to contain additional information needed to support the synchronization of the materialized views with the data sources (for more details see Section 2).

1.2.2 Data Visualization

The purpose of a *Data Visualization Application* is to create visual images of large sets of data, so that the users of the system can better perceive and understand the data. Possible visualizations include graphs, maps, diagrams and s.o. A typical visualization application consists of two components - the *Data Query Module* and the *Graph Display Module* (see Figure 6).

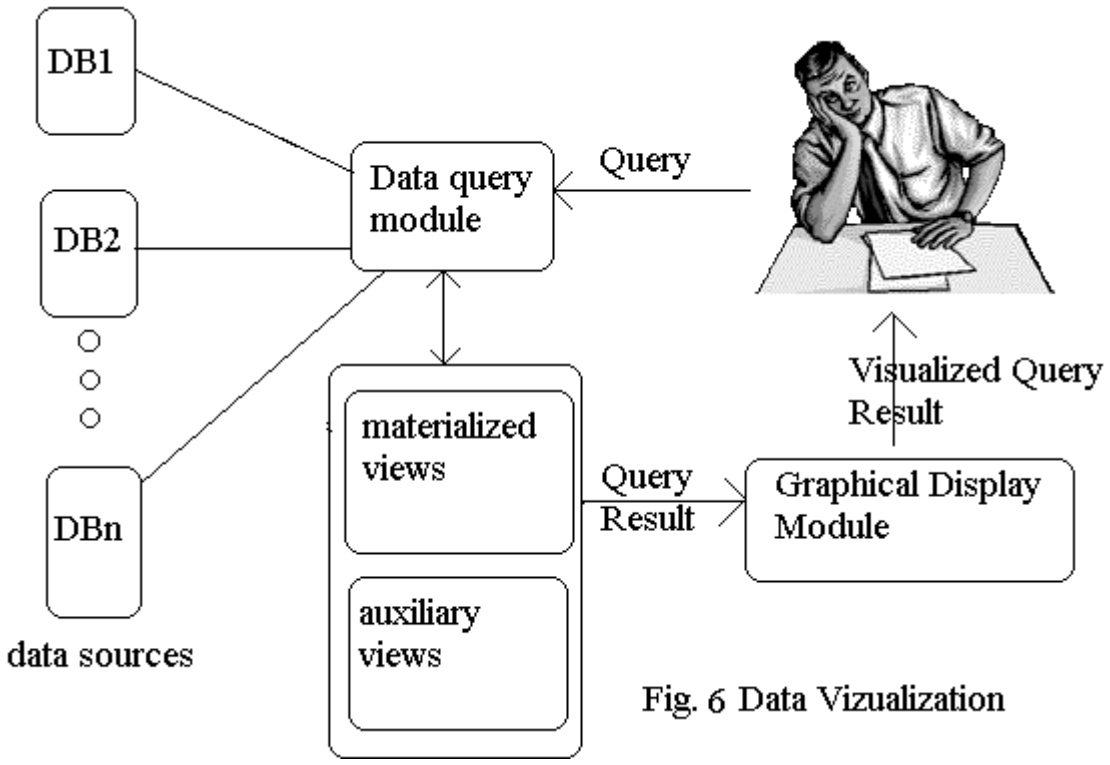

Fig. 6 Data Vizualization

The data query module has the responsibility to calculate the results of user queries against a distributed database and store them as materialized views. The graphical display module has the capability of displaying the data in the materialized views in a way specified by the user. Materialized views are needed in this scenario because the display module usually needs to work with physically stored data that it can display.

1.2.3 Mobile Systems

In recent years, palmtops have become increasingly popular, not only for accessing local information such as phone numbers and appointments, but also as a device for communicating to the outside world. The connection between a palmtop and the outside world (e.g. the internet) is provided through evenly spread *Geo-Positioning Systems*. For example, the palmtop users may want to query the outside world for a particular information. Since the transfer bandwidth is limited because of its physical characteristics, materialized view may be used to store the results of frequently posed queries and in this way reduce the bandwidth traffic. A particular characteristic of the posed queries in a mobile system environment is that the location of the palmtop may be a query parameter.

For example, let's consider the scenario shown in Figure 7.

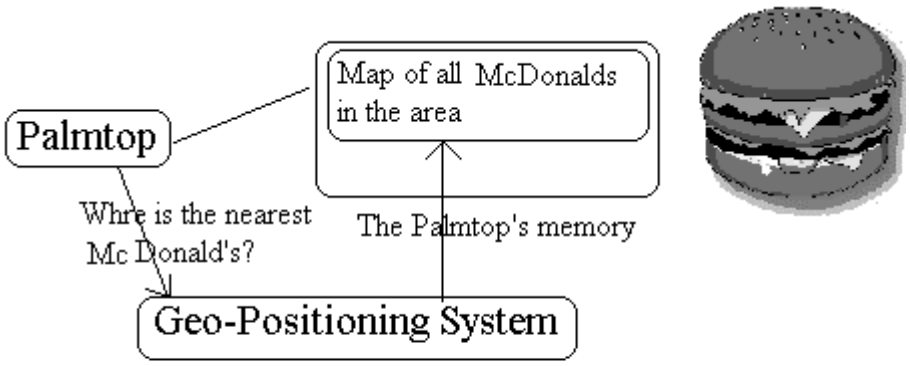

Figure 7. Mobile System Example

The palmtop user wants to know about all McDonalds in the area. The query is sent to the geo-positioning system, which returns the answer. The query result is stored in a materialized view on the palmtop. Now, if five miles down the road the palmtop user posses the same query, only the change to the map have to sent to the laptop, i.e. we have economized bandwidth.

1.2.4 Telephone Switches

Unlike mobile systems, in telephone switches the crucial requirement is minimizing the query response time. Usually, each switch contains a main memory database in C++ structure consisting of several collections of objects. For a phone call to be routed, information from several structures is usually required, i.e. a join between them has to be performed. However, doing a join in real time for such type of applications is unfeasible because of its slow execution. That is why, the results of frequently used joins are usually stored in switches as main-memory materialized views.

1.3 Organization of the Paper

So far we have presented the basic components of a semantic data control engine for a distributed system and have given example of practical application of those components. In the next sections we will be explore existing algorithms for implementing the different the components. In Section 2, we will explore materialized view maintenance algorithms. In Section 3, we will cover existing materialized view update algorithms. In Section 4, we will examine how update constraints for imposing integrity constraints are produced through a technique called integrity constraint compilation In Section 5, we will explore the other method of integrity constraint enforcement - through integrity constraint restoring algorithms. Finally, in Section 6 we will summarize the presented material and present some areas for future research.

1.4 Bibliographic Notes

The presented classification of constraints and updates is an original contribution of the author of the paper. The presented examples of applications of materialized views are based on similar examples presented in [Mumi95] and [GuMu99]. More examples on the application of data synchronization algorithms to the areas of data integration and query optimization can be found in [GJM96], [ZHKF95] and [CKPS95].

2. Materialized View Maintenance

A materialized view is the stored result of a retrieve query. As the data, on which the view is based (the so called underlying data) changes, so should the materialized view to reflect the changes. Materialized view maintenance algorithms do exactly that, i.e. they propagating updates to the underlying data sources to the materialized view. In this section we will first classify existing materialized view maintenance algorithms and then we will describe some of the more prominent ones.

2.1. Classification of Materialized View Maintenance Algorithms.

As the base data, on which materialized views are based, change so should the views themselves in order to correctly reflect the new state of the database . The update of a view is called *view refresh* and the ongoing process of synchronizing the view with the underlying data is called *view maintenance*. There are two general types of view maintenance algorithms - *immediate* and *deferred*. As the names show, in the first type of algorithms a view is updated during the update to the base data, while in the second type of algorithms the view refresh is done somewhere in the future, as a result of the triggering of some event . Such event may include the activation of an alarm clock, user interaction with the system, or may be triggered by a component of the system.

Before we continue our classification of view maintenance algorithms, it would be useful to look at an example (see Figure 2). The materialized view V, stored at *site 3* is calculated as the inner join of two tables R_1 and R_2. Let's assume that at time 1, R_1 is updated to R_1' and R_2 is updated to R_2'. Then V should change to $V' = R_1' \bowtie R_2' = (R_1 + {}_{\triangle}R_1) \bowtie (R_2 + {}_{\triangle}R_2) = R_1 \bowtie R_2 + R_1 \bowtie {}_{\triangle}R_2 + {}_{\triangle}R_1 \bowtie R_2 + {}_{\triangle}R_1 \bowtie {}_{\triangle}R_2$, where ${}_{\triangle}R$ is used to denote the changes made to the table R, which may include insertions, deletions and modifications. As well, $R + {}_{\triangle}R$ is used to represent the result of adding the changes of ${}_{\triangle}R$ to R. If at time 1, V' is recomputed as $R_1' \bowtie R_2'$, without the old value of the view V to be used then we classify this algorithm as *direct view refresh* algorithm. If on the other hand, ${}_{\triangle}V$ is calculated first and then V' is calculated as $V' = V + {}_{\triangle}V$, then we have an *incremental view refresh*. In most cases the incremental algorithms are more efficient than the direct update algorithms, i.e. in the case of small changes to the base tables it is usually cheaper to compute ${}_{\triangle}V$ and then V' as $V + {}_{\triangle}V$, than to recompute V' from scratch.

Continuing our classification, the incremental view maintenance algorithms can be divided into two types: the ones that use *auxiliary views* and the ones that don't. To illustrate the difference on our example, an algorithm of the second type could calculate $\triangle V$ as $R_1 \bowtie \triangle R_2 + \triangle R_1 \bowtie R_2 + \triangle R_1 \bowtie \triangle R_2$ or as $\triangle R_1 \bowtie R_2' + \triangle R_2 \bowtie R_1' - \triangle R_1 \bowtie \triangle R_2$ where "-" is used to indicate difference. If we know, that certain integrity constraints hold on the data, the above expressions could be simplified. Note that, since neither the old values of the underlying tables - R_1 and R_2, nor the new values R_1' and R_2' are stored at *site 3*, they have to be fetched from *sites 1* and *2* when a refresh is performed, which could result in excessive network traffic if those relations are big. Another problem in these type of view maintenance algorithms is that when *site 1* receives a request for the value of R_1', it may have already changed to a new state. This problem is corrected by running an error correcting procedure. Both problems are eliminated if auxiliary views are used.

An auxiliary views contains representative data from the underlying data, which is sufficient to perform a view refresh. Overhead of this type of algorithms is the extra storage spaced used for the auxiliary views and the extra processing time for refreshing the auxiliary views. The later is the case because auxiliary views are materialized views themselves and therefore also need to be refreshed. Just as in the case where auxiliary views are not used, knowledge on existing integrity constraints may improve the algorithm's performance. More specifically, when auxiliary views are used, the knowledge of integrity constraints may be used to reduce the size of those views, which will result in reduced storage and faster performance.

The classification of materialized view maintenance algorithms presented so far is shown in Figure 8.

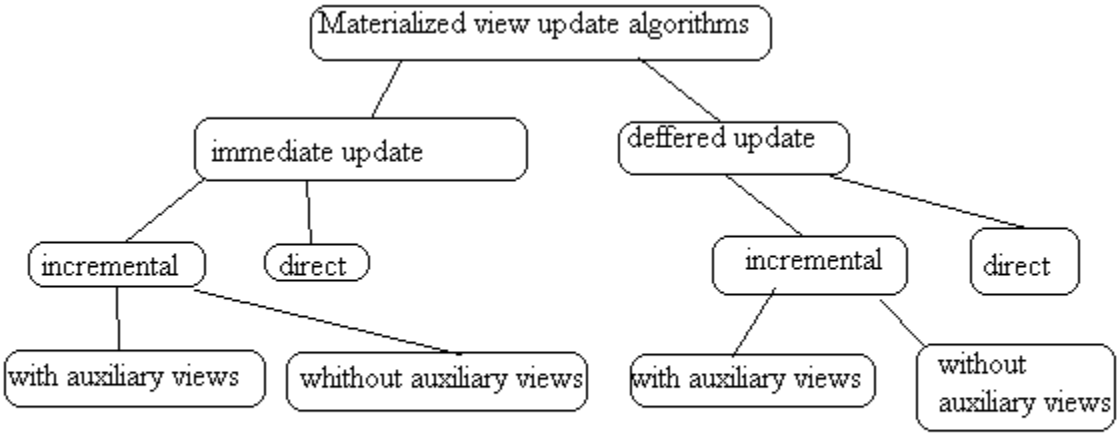

Figure 8. Classification of materialized view maintenance algorithms

In the remainder of this section we will explore in greater details the theory behind materialized view maintenance, with and without auxiliary views. As well, we will give concrete examples of algorithms from literature implementing view maintenance.

2.2 Incremental View Maintenance without Auxiliary Views

Let's first examine how materialized views based on SPJ (Select-Project-Join) queries are managed. Such view V can be written as $\pi_A(\sigma_\gamma(R_1 \bowtie R_2 \bowtie ... R_n)$, where $R_1,...,R_n$ are relations, $\bowtie$ is used to represent an inner join, γ is a predicate condition and A is a subset of the attributes of the tables. Let's examine each operation separately. If $V=\sigma_\gamma(R)$, then if R changes to R', V' will change to $\sigma_\gamma(R')= \sigma_\gamma(R+_\Delta R)= \sigma_\gamma(R)+\sigma_\gamma(_\Delta R)=V+ \sigma_\gamma(_\Delta R)$, i.e. ΔV can be computed as $\sigma_\gamma(_\Delta R)$, without accessing the base tables. Here we have used the fact that selection is a distributive operator relative to union.

Unfortunately, projection is not a distributive operator relative to union. Consider the simple example $R=\{(2,1),(2,2),(3,1)\}$ and $_\Delta R=\{-(2,1)\}$ (here - is used to denote deletion). Then $\pi_1(R+_\Delta R)= \pi_1(\{(2,2),(3,1)\})=\{(2),(3)\}$ and $\pi_1(R)+\pi_1(_\Delta R)=\{(2),(3)\}-\{(2)\}=\{(3)\}$. The problem with the above example is that we used a projection that eliminates duplicates, i.e. information is lost. What most view maintenance algorithms do, to overcome this problem, is to somehow deal only with projections that don't eliminate duplicates and are therefore distributive relative to union.

One way of insuring that duplicate tuples are not eliminated during projection is not to allow duplicates in the first place. This can be achieved by introducing a unique ID value assign to each tuple and requiring all projections to include this ID attribute as part of the projection attribute lists. Since the introduced IDs are unique, projections will not have duplicate tuples in the result to eliminate. In the above example if we add the tuple identifiers as the first attributes we will get $R=\{(1,2,1),(2,2,2),(3,3,1)\}$ and $_\Delta R=\{-(1,2,1)\}$. Now a projection only on the second attribute will not be allowed, because it doesn't include the first attribute, i.e. the ID attribute. If a projection on the first and second attribute are performed we get $\pi_{1,2}(R+_\Delta R)= \pi_{1,2}(\{2,2,2),(3,3,1)\}) = \{(2,2),(3,3)\}$ and $\pi_{1,2}(R) + \pi_{1,2}(_\Delta R)=\{(1,2),(2,2),(3,3)\}-\{(1,2)\}= \{(2,2),(3,3)\}$. Note that in this case, we may view the tuples as being objects with unique identifiers in an object oriented database, which shows that the projection duplicate elimination problem is specific to the relational model.

The second common way of dealing with projections is to "implement" duplicate preserving projections. This can be done by storing next to each tuple an additional value stating how many time the tuple appears in the relation. We redefine projection to preserve duplicates and our example will now look as: $\pi_1(R+_\Delta R)= \pi_1(\{(2,2,\#1),(3,1,\#1)\}) =\{(2,\#1),(3,\#1)\}$ and $\pi_1(R)+\pi_1(_\Delta R)=\{(2,\#2),(3,\#1)\}-\{(2,\#1)\}=\{ (2,\#1) ,(3,\#1)\}$, where we have used $\#x$ after each tuple to show how many times it appears in the relation. A disadvantage of this algorithm is that it increases the storage space of the relations (we have to keep track of how much times each tuple is repeated) and the processing time of executing operations such as projections and joins over them. An advantage of this algorithm is that it makes duplicate elimination, aggregation and sorting faster. As well, as we shall see later in this section, it simplifies view maintenance

algorithms on views produced with queries involving aggregates. The presented algorithm is known in literature as the counting algorithm and has several variations.

Now, let's look at a simple example of a query involving joins. If $V=R_1 \bowtie R_2 \bowtie R_3$ and the underlying relations are changed, the materialized view should be updated to $V'=(R_1+_\Delta R_1)\bowtie(R_2+_\Delta R_2)\bowtie(R_3+_\Delta R_3)=R_1\bowtie R_2\bowtie R_3+R_1\bowtie R_2\bowtie_\Delta R_3+...+_\Delta R_1\bowtie_\Delta R_2\bowtie_\Delta R_3$. We have used the fact the join operator is distributive relative to union to do the expansion. Note that, if not all three base tables are updated, some of the 8 terms in $_\Delta V$ will be canceled. Note in particular, that if V is based on a multi-join between n tables, $_\Delta V$ will unfolded 2^n-1 expressions.

One proposed way, to avoid calculating all the expressions in $_\Delta V$ is based on tuple marking. What such algorithms do, is to mark the tuples in the base relations as inserted, deleted or old. Then a join between the tables can be performed, in such a way that tuples are matched in the join, only if one of them is a new tuple. Whether the resulting tuples are to be part of insertion or deletion in $_\Delta V$ or to be ignored is determined by the tuples joined to form the resulting tuple. For example if an inserted tuple is joined with a deleted tuple, the resulting tuple will be ignored in $_\Delta V$. On the other hand if a deleted tuple is joined with a deleted tuple, the resulting tuple (the tuple calculated by joining the two tuples) will be added as deletion to $_\Delta V$.

Note that the formula for computing ΔV when a join condition is involved, and in general, could be simplified if information about the integrity constraints that hold on the underlying data are known. Let's look at a simple example to see how this can be done. Suppose table R_1 has attributes ($\underline{A}$,B), where A is a primary key and B is a foreign key to the table R_2, which has attributes ($\underline{B}$,C), where B is a primary key. Then if $V=R_1\bowtie R_2$, ΔV can be calculated as $_\Delta R_1\bowtie_\Delta R_2+_\Delta R_1\bowtie R_2+R_1\bowtie_\Delta R_2$. However, knowing the integrity constraints which hold on the tables, we know that $R_1\bowtie_\Delta R_2$ will be empty because of the referential constraint between the two tables, which will simplify the formula for ΔV. There are still many open research questions in the area of exploiting integrity constraints to optimize view maintenance algorithms (for references to existing algorithms see the bibliographical notes).

Maintenance algorithms, for views based on queries with outer joins and duplicate preserving relational algebra operators and with recursive queries are not covered in this survey paper because of their complexity. However, algorithms for doing so are known in literature (see the bibliographic notes of this section for references).

To show how materialized views defined with queries involving aggregates can be maintained, let's look at a simple example. Suppose $R_1=\{(1,2),(2,1),(2,4)\}$ and $V = {}_1\mathcal{F}_{\min(2)}(R_1)$. Initially V will contain $\{(1,2),(2,1)\}$. It is clear that V can not be updated, knowing only the changes made to R_1. In general the aggregates *min* and *max* are not self-maintainable, i.e. information about the state of the base tables is needed. Indeed

there isn't a fast way of incrementally updating such views. In the above example, if the tuple (1,2) is deleted from R_1 an algorithm for calculating ${}_{\Delta}V$ has to scan the whole relation R_1 to find the new minimum value, and the time complexity of doing so is equal to re-computing V from scratch. If we have only self-maintainable aggregate such as *count* or *sum* in the definition of V, an incremental refresh of V can be done by using the tuple counting technique described earlier. For example let's have the above example, but redefine V as ${}_1\mathcal{F}_{\mathsf{sum}(2)}(R_1)$ or V={(1,2),(2,5)} . Now, suppose the tuple (1,2) is deleted from R_1. We can not simply subtract (1,2) from V and change it to {(1,0),(2,5)}, because this is obviously incorrect. On the other hand if we keep track on the number of original tuples used to produce each aggregation, we will store V as {(1,2,#1),(2,5,#2)} and will be able to perform the deletion of (1,2) from V correctly.

This concludes our overview of view maintenance algorithms that don't use auxiliary views. In the above presentation we have only demonstrated how some basic algorithms for such maintenance work, without going into great details. In the next section we will do the same thing for algorithms in which auxiliary data is used.

2.3 Incremental View Maintenance with Auxiliary Views

Each materialized view may have a number of auxiliary views associated with it. The purpose of the auxiliary views is to allow the corresponding materialized view to be maintained based only on the changes made to the underlying data and not on the underlying data itself. Formally, if Σ is a database schema, D a database instance of it, Q is the query producing the materialized view and *changes* is the set of allowed changes on the schema, the predicate that represents $\{R_i\}_{i=1}^n$ being a set of auxiliary views is:

(auxiliary_views $\{R_i\}_{i=1}^n$,Σ,D,Q,changes) ≡ (∃ query Q') s.t. (∀μ∈changes(Σ)) [if (D+μ)∈Models(Σ) then Q(D+μ)=Q'(μ,Q(D), $\{R_i\}_{i=1}^n$)]

The above expression says, that $\{R_i\}_{i=1}^n$ is a set of auxiliary views only if V can be refreshed using them, the changes to the base tables and the old value of the materialized view. Existing research in the area of materialized views has concentrated on the problem of finding the smallest auxiliary set of views for which the above auxiliary_views predicate holds. Clearly, the more is known about the integrity constraints that hold on the data, the smaller the auxiliary views can be made.

Let's look at an example to demonstrate the approach. Suppose we have a table $X(\underline{A},B,C)$ where A is a primary key and B and C are foreign keys referencing the tables $Z(\underline{B},D,M)$ and $Y(\underline{C},F,G)$ with primary keys B and C respectively. Let's as well assume that the integrity constraints of the database are guaranteed through a multiple update constraint, which as well allows only for tuples to be inserted and deleted, but doesn't allow more than one operation on the same tuple in the same transaction. Let the materialized view V be defined as $V=\pi_{A,F,D}(\sigma_{D>10 \wedge F<3}(X \bowtie_{X.C=Y.C} Y \bowtie_{X.B=Z.B} Z)$. Now note that V can be rewritten as $V = \pi_{A,B,C}(X) \bowtie_{X.C=Y.C} \pi_{C,F}(\sigma_{F<3}(Y)) \bowtie_{X.B=Z.B} \pi_{B,D}(\sigma_{D>10}(Z))$ because

selection and join, and duplicate preserving projection and join are commutative. We have duplicate preserving projection is this key because in all projections the key of the table is projected. In general however, this is not the case and tuple identifiers or tuple counters may have to be added as described in the previous section. Now V_{new}, the new value of V will be equal to $V_{new} = \pi_{A,B,C}(X + \Delta X) \bowtie_{X.C=Y.C} \pi_{C,F}(\sigma_{F<3}(Y + \Delta Y)) \bowtie_{X.B=Z.B} \pi_{B,D}(\sigma_{D>10}(Z + \Delta Z))$. Taking in account the integrity referential constraint that hold on the schema, we see that the $\Delta Y \bowtie X$ and $\Delta Y \bowtie Z$ will be empty and that V_{new} can be rewritten as $V + \pi_{A,B,C}(\Delta X) \bowtie_{X.C=Y.C} \pi_{C,F}(\sigma_{F<3}(\Delta Y)) \bowtie_{X.B=Z.B} \pi_{B,D}(\sigma_{D>10}(\Delta Z)) + \pi_{A,B,C}(\Delta X) \bowtie_{X.C=Y.C} \pi_{C,F}(\sigma_{F<3}(\Delta Y)) \bowtie_{X.B=Z.B} \pi_{B,D}(\sigma_{D>10}(Z + \Delta Z)) + \pi_{A,B,C}(\Delta X) \bowtie_{X.C=Y.C} \pi_{C,F}(\sigma_{F<3}(Y + \Delta Y)) \bowtie_{X.B=Z.B} \pi_{B,D}(\sigma_{D>10}(\Delta Z))$, which shows that we need to store only $\pi_{B,D}(\sigma_{D>10}(Z))$ and $\pi_{C,F}(\sigma_{F<3}(Y))$ as the auxiliary views.

In similar way, auxiliary views can be defined over queries containing outer joins, recursive definitions and aggregates.

Note that there is a correspondence between the algorithms that use and don't use auxiliary views. In both of them, formulas for calculating ΔV that rely as much as little as possible on the underlying tables are tried to be computed in identical ways by exploiting integrity constraints. The difference is that small reliance on the underlying tables, in the first type of algorithms, is used to reduce disk storage, while in the second it is reduced to reduce network traffic during materialized view refresh.

2.4 Summary

In this section we have classified different existing algorithms for materialized view maintenance and demonstrated how some of them work, mostly through examples. We didn't go in much details because of the limited scope of the paper.

2.5 Bibliographic Notes

The presented classification of view maintenance algorithms is an original one. However it is partially based on the material presented in the survey paper on materialized view updates - [GuMu95]. Good reference on the problem of error correction for algorithms that don't use auxiliary views are [AESY97] and [SaBe00]. Marked tuple algorithms (also known as tagging tuple algorithms) are first described in [BLT86]. For a reference on tagging tuples in distributed databases see [BDMW98]. Good papers on the topic of exploiting integrity constraints to optimize view maintenance algorithms are [Huyn96], [QGMW96] and [Stan01]. Outer join view are described in [GJM94]. There are two basic algorithms for maintaining view with recursive definition - the DRed Algorithm and the Propagation/Filtration algorithm. The first is described in [GMS93] and the second in [HD92]. A fundamental paper on the problem of maintaining views defined with duplicate preserving algebraic expressions is [GrLe95]. The first paper to propose an overall solution to the problem of maintaining view with aggregates is [MQM97].

For the auxiliary view presentation we have used material from [Stan01] and [QGMW96]. Other reference on the subject include [AMV98] and [Huyn96].

3. Materialized View Update

A materialized view update algorithm tries to translate updates made to materialized views to the underlying tables. Unless the information stored in the materialized view is not "equivalent" to the information stored in the underlying tables there isn't a unique way of propagating the update. The idea behind view updates is to specify at view creation time in what ways are updates to the view to be translated to the underlying data.

3.1 Translating Materialized View updates to the Underlying Data

Let's have a database schema Σ, and two materialized views V_1 and V_2 defined by the queries Q_1 and Q_2 over Σ. We will write that $V_1 \geq V_2$ iff $\forall D_1,D_2 \in \text{Models}(\Sigma)$, $Q_1(D_1)=Q_1(D_2) \Rightarrow Q_2(D_1)=Q_2(D_2)$. Informally, this definition tells us that if $V_1 \geq V_2$ then if we know the value of V_1 we can compute the value of V_2, i.e. V_1 is more "informative" than V_2. If $V_1 \leq V_2$ and $V_2 \leq V_1$, we will write $V_1=V_2$ and say that the two views are equivalent. The presented notions of expressiveness and equivalence can be extended to set of views. A set of views defined by the set of queries $\boldsymbol{S_1}$ is more informative than a set of views defined by the set of queries $\boldsymbol{S_2}$, or $\boldsymbol{S_1} \geq \boldsymbol{S_2}$ iff $\forall D_1,D_2 \in \text{Models}(\Sigma)$, $[\forall Q_1 \in \mathbf{S_1}\ Q_1(D_1)=Q_1(D_2)] \Rightarrow [\forall Q_2 \in \mathbf{S_2}\ Q_2(D_1)=Q_2(D_2)]$.

Two views V_1 and V_2, defined over the same set of relations $\boldsymbol{R}$ are complement relative to $\boldsymbol{R}$ if the set of views $\{V_1,V_2\}$ is equivalent to $\boldsymbol{R}$. Informally given a view V, its complement view is a view containing the missing information from $\boldsymbol{R}$. Unless V is equivalent to $\boldsymbol{R}$ or has nothing in common with $\boldsymbol{R}$ (i.e. V complement is equivalent to $\boldsymbol{R}$) V has more than one complementing views relative to $\boldsymbol{R}$. A translation of a view V with underlying tables $\boldsymbol{R}$ can be identified uniquely by specifying which complement of V is to remain unchanged during the update. Note that in the case where V is equivalent to $\boldsymbol{R}$ is the trivial case in which there is an unique translation from V to $\boldsymbol{R}$, and the case where V has nothing in common with $\boldsymbol{R}$ the changes to V don't have to be propagated to $\boldsymbol{R}$. In other cases the complement which should remain invariant during updates has to be specified.

Let's look at the example from figure 9. As well we have the constraints that *EMP* is a primary key for the employee table and E.DEP is a foreign key to the department table.

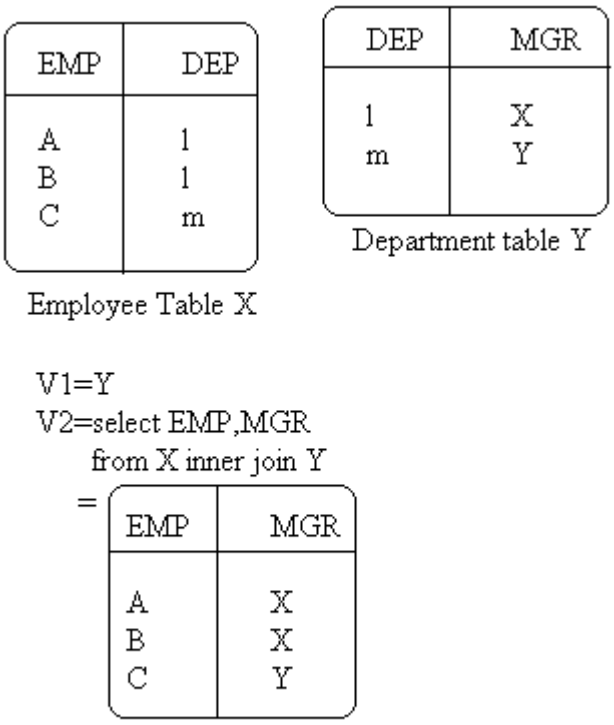

Figure 9. Example for view update

Suppose we have defined the translations on V_2 in such a way that the complement view V_1 is to remain invariant under the translation. Then if employee *A* is replaced by employee *F* in V_2, employee *A* will be replaced by employee *F* in table *X* and *Y* will remain unchanged. On the other hand note that if a view V_3=X is to be the invariant compliment of V_2 than the above update will be unfeasible.

This example shows that by specifying what compliment of the view is to remain invariant will restrict the number of possible translations of an update to the view to *at most* one, i.e. in some cases a translation may me unfeasible. We can allow only updates to materialized views, for which there is a translation relative to the specified invariant compliment. This can be done by "compiling" the invariant complaint constraint in update constraint. In the above example, if V_3 is chosen as the invariant complement of V_2, then a corresponding update constraint will be (FALSE, FALSE,t .new(EMP) = t.old(EMP)), i.e. only modifications on the MGR attribute are allowed.

The presented model of disallowing changes to the selected complement is considered by some to be too restrictive. Other modals for selecting unique translations have also been developed based on other factors.

3.2 Bibliographical Notes

The presented in this section material is from [BaSp81] and [Kele95]. Related material on the problem can be found in [Kele82], [Kele85],[Kele86],[DaBe78], [DaBe82] and [Furt79].

4. Constraint Compilation

The problem of constraint compilation is to enforce static integrity constraints by "compiling" them in appropriate update constraints. When such update constraints are imposed, the validity relative to the compiled integrity constraints will be guaranteed.

4.1 Defining integrity constraints

Integrity constraints in their most general form can be expressed as:

$(Q_1 x_1) \ldots (Q_n x_n) \vee A_i$

where Q_1 through Q_n are existential or universal quantifiers and x_1 trough x_n are variables appearing in the predicates A_i. Each predicate P_i can be a positive or a negative literal. Each literal is either of the form $P(x_1,...x_k,Q(x_1,...x_k))$ where Q is a query over the database on which the integrity constraint is defined. The predicate P is a build in predicate like <,=,∈. For example suppose we have the database EMP(ENO,ENAME,TITLE), PROJ(PNO, PNAME,BUDGET). Then, we can express the constraint that the total duration for all employees in the CAD project is less than 100 as (this example is from [ÖzVa99]):

$\forall j$ {[PROJ(j)]=>[j∈(select * from PROJ as P where P.NAMD="CAD") => (select sum(G.DUR) from ASG as G where G.PNO=j.PNO)<100)]}

4.2 Constraint Enforcement

One example of compiling integrity constraints was already given in Section 1.1.1. In this section we will show a more complicate example that can use materialized views to perform integrity constraint enforcement.

Let's go back to the example from the previous sub-section. To implement the constraint: *the total duration for all employees in the CAD project is less than 100* hold we may store the current total duration for all employees in the CAD project as a materialized view *V* with single attribute *A*. Then, we may specify the update constraint (t=∅,FALSE,t(new).A<100) for the materialized view *V*. This constraint specifies that we may insert a tuple in V only if it is empty, we may not delete tuples from *V* and we may modify the value of *V* only if the newly inserted value is less than 100. As it can be seen this materialized view algorithm is more powerful then the simple condition

compilation algorithm presented in Section 1.1.1. The disadvantage of this algorithm is the added load of maintaining one more materialized view.

4.3. Bibliographical Notes

The presented in this section material is from [ÖzVa99], [SiVa84] and [Cive88].

5. Integrity Constraint Restoring Algorithms

In a distributed database constraints may often be violated. This can be a result of allowing updates to the database that violates the integrity constraints, or as a result of constraints imposed on materialized views.

5.1. The algorithm

Let Σ be a database schema and *IC* be a set of integrity constraints. Let Models(Σ,*IC*) be the set of all instances of Σ that don't violate the integrity constraints *IC*. Let's define the "distance" between two instances of Σ as the minimum number of insertions and deletions that have to be performed to transfer one of the database to the other. If D_1 and D_2 are databases we will denote the distance between them as $|D_1\text{-}D_2|$. Now we will define the set of repairs of a database instance *D* of Σ, relative to Σ and *IC* as:

$$Repair(D,\Sigma,IC) = \{D' \mid D' \in Models(\Sigma,IC) \wedge [\forall (D'' \in Models(\Sigma,IC)) |D\text{-}D'| <= |D\text{-}D''|]\}$$

In words, the set of repairs of a database instance *D* is the set of databases that are closest to it relative to the defined metric and satisfy the *IC*. Note that there may be more than one repairs for a given database.

Let's look two examples. Consider the table R={(1,2),(1,3),(2,1)}, where the first attribute is a key. The set of minimal repairs is {{(1,2),(2,1)},{(1,3),(2,1)}}. Consider the tables R_1={(1,2),(2,1)} and R_2={(1)} and the constraint $R_1.2 \subseteq R_2.1$. One possible repair is to delete the tuple (1,2) from R_1 and a second possible repair is to add the tuple (2) to R_2. Note that in general there may be exponentially many repairs to the number of tuples in a table, but this is not a problem because we are looking for any repair.

Note that a repair can have a cascading effective. This because a repair is an update and when it is performed other integrity or duplication constraints may be violated as a result. The termination of such a cascading algorithm is not guaranteed. A possible solution to this problem is exploring conflict graphs containing the data objects that violate integrity or duplication constraints.

5.2. Bibliographical Notes

The material in this section was based solely on [ABC99] as one of the unique references in the area of repairing invalid databases.

6. Conclusion

In this paper we presented a formal framework for specifying and imposing constraints on the data in a distributed database environment. Although imposing replication constraints and imposing integrity constraints are inherently related, we only hinted on their relationship. Problems on how integrity constraint imposing algorithms can interfere with duplication constraint imposing algorithms or update constraint imposing algorithms were not covered. Synchronizing the three type of algorithms that guarantee the validity of integrity, update and duplication constraints is an open research problem.

References

[ABC99] M. Arenas, L. Bertossi, J. Chomicki, Consistent Query Answers in Inconsistent Databases, In *ACM Symposium on Principles of Database Systems (PODS)*, Philadelphia, May 1999.

[AESY97] D. Agrawal, A.El Abbadi, A.Singh and T.Yurek, Efficient View Maintenance at Data Warehouses, In *ACM SIGMOD*, 1997.

[AMV98] S. Abiteboul , J. McHugh , M. Rys , V. Vassalos , J. Wiener, *Incremental Maintenance for Materialized Views over Semistructured Data*, 24th International Conference on Very Large Databases (VLDB), August, 1998.

[BaSp81] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Trans. Database Syst.* 6(4), pages 557-575, December 1981

[BDMW98] James Bailey, Guozhu Dong, Mukesh K. Mohania and Xiaoyang Sean Wang. Incremental View Maintenance By Base Relation Tagging in Distributed Databases. In *Distributed and Parallel Databases* 6(3): pages 287-309, 1998

[BLT86] Jose A Blakely, Pre-Ake Larson, Frank Wm Tompa, Efficiently Updating Materialized Views, In *ACM SIGMOD*, 1986.

[Cive88] F.N.Civelek, A. Dogac, and S. Spaccapietra. An Expert System Approach to View Definition and Integration. In *Proc. 7th Int'l. Conf. on Entity-Relationship Appraoch*, pages 229-249, November 1988.

[CKPS95] Surajit Chaudhuri, Ravi Krishanmaurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh IEEE International Conference on Data Engineering (ICDE)*, pages 190-200, Taipei, Taiwan, March 6-10 1995.

[DaBe78] U. Dayal and P.Bernstein. On the Updatability of Relational Views. In *Proc. 4th Int. Conf. on Very Large Dada Bases*, pages 368-377, September 1978.

[DaBe82] U.Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. In *ACM Trans. on Database Systems*, 7(3), 1982

[Furt79] A.L.Furtado *et al*. Permitting updates through views on databases. In *Information Systems*, 4(4), 1979.

[GJM94] Ashish Gupta, Hosagrahar V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. Technical Memorandum 113880-941101-32, AT&T Bell Laboratories, November, 1994.

[GJM96] Ashish Gupta, Hosagraphar V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth Interanational Conference on Extending Database Technology (EDBT)*, pages 140-144, Avignon, France, March 25-29 1996.

[GrLi95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD*, pp. 328-339, 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining view incrementally. In *ACM SIGMOD*, 1993.

[GuMu95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Vies: Problems, Techniques and Applications, In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, June 1995.

[GuMu99] Ashish Gupta and Inderpal Singh Mumick. Materialized views - Techniques, Implementations and Applications, MIT Press, 1999.

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

[Huyn96] N. Huyn, *Exploiting Dependencies to Enhance View Self-Maintainability*, unpublished, http://www-db.stanford.edu/pub/papers/fdvsm.ps, 1996.

[Kele82] A. M. Keller. Update to Relational Databases through Views Involving Joins. In *Proc. 2nd Int. Conf. on Databases: Improving Usability and Responsiveness*, pages 363-384, June 1982.

[Kele85] A.M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth Symposium on Principles of Database Systems,* Portland, OR, 1985

[Kele86] A.M. Keller. The role of semantics in translating view updates. *IEEE Computer*, 19(1), pages 63-73, 1986

[Kele95] Arthur Michael Keller. Updating Relational Databases through Views. Ph.D. dissertation at Stanford. June 1995.

[Mumi95] Indrpal Singh Mumick. The Rejuvenation of Materialized Views. In *Proceedings of the Sixth International Conference on Information Systems and Management of Data (CISMOD)*, Bombay, India, November 15-17, 1995.

[MQM97] Inderpal Singh Mumick, Dallan Quass, Barinderpal Singh Mumick: Maintenance of Data Cubes and Summary Tables in a Warehouse. In *ACM SIGMOD*, pp. 100-111, 1997

[ÖzVa99] M.Tamer Özsu and Patrick Valduriez. Principles of Distributed Database Systems. Prentice Hall,1999.

[QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. *Making Views Self-Maintainable for Data Warehousing*, Proc. of Fourth Intl. Conf. on Parallel and Distributed Information Systems (PDIS '96), December 1996.

[Stan01] Lubomir Stanchev, Reducing the Size of Auxiliary Data Needed to Maintain Materialized Views by Exploiting Integrity Constraints, University of Waterloo Technical Report CS2001-02, 2001.

[SaBe00] Kenneth Salem and Kevin Beyer, How to Roll a Join: Asynchronous Incremental View Maintenance, In *ACM SIGMOD*, 2000

[SiVa84] E. Simon and P. Valduriez. *Design and Implementation of an Extendible Integrity Subsystem*. In *Proc. ADM SIGMOD Int. Conf. on Management of Data*, pages 9-17, June 1984.

[ZHKF95]Gang Zhou, Richard Hull, Roger King, and Jean-Claude Franchitti. Using object matching and materialization to integrate heterogeneous databases. In *Proc. of 3rd International Conference on Cooperative Information Systems*, pages 4-18, 1995.