WPI-CS-TR-97-4 Spring 1997 (Revised Dec. 1997)

The *EVE* Framework: View Synchronization In Evolving Environments

by

Amy J. Lee
Anisoara Nica
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

The *EVE* Framework: View Synchronization In Evolving Environments *

Amy J. Lee†, Anisoara Nica†, and Elke A. Rundensteiner‡

(†) Department of EECS
University of Michigan, Ann Arbor
Ann Arbor, MI 48109-2122
amylee,anica@eecs.umich.edu

(313) 764-1571

(‡)Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
rundenst@cs.wpi.edu

(508) 831-5815

Abstract

The construction and maintenance of data warehouses (views) in large-scale environments composed of numerous distributed and evolving information sources (ISs) such as the WWW has received great attention recently. Such environments are plagued with changing information because ISs tend to continuously evolve by modifying not only their content but also their query capabilities and interface and by joining or leaving the environment at any time. We are the first to introduce and address the problem of capability (schema) changes of ISs, while previous work in this area, such as incremental view maintenance, has mainly dealt with data changes at ISs. In this paper, we outline our solution approach to this challenging new problem of how to adapt views in such evolving environments. We identify a new view adaptation problem for view evolution in the context of ISs capability changes, which we call *View Synchronization*. We also outline the Evolvable View Environment (EVE) approach that we propose as framework for solving the view synchronization problem, along with our decisions concerning the key design issues surrounding EVE. The main contributions of this paper are: we provide an E-SQL view definition language with which the view definer can direct the view evolution process, we introduce a model for information source description which allows a large class of ISs to participate in our system dynamically, we formally define what constitutes a legal view rewriting, we develop replacement strategies for affected view components which can be shown to be correct, and we provide a set of view synchronization algorithms. A prototype of our *EVE* system has successfully been built using Java, JDBC, Oracle, and MS Access; and is currently running in the CS Department at WPI.

1 Introduction

1.1 Motivation and Problem Definition

Advanced applications such as web-based information services, data warehousing, digital libraries, and data mining typically create and maintain tailored information repositories gathered from among a large number of

*This work was supported in part by the NSF RIA grant #IRI-9309076 and NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular, IBM and Informix.

internetworked information sources (ISs) [Wid95], such as the World Wide Web. There is generally a large variety and number of ISs in these modern environments, each modeled by diverse data models and each supporting different query interfaces and query processing capabilities. Furthermore, individual ISs are autonomous, freely updating both their content and their capabilities, even frequently joining or leaving the environment.

In order to provide efficient information access in such environments, relevant data is often retrieved from several sources, integrated as necessary, and then materialized into what is called a *view* in database terminology [Wid95]. In fact, businesses are beginning to boom that focus exactly on this type of "middle layer" service by offering to collect related information (about products or services) from multiple sources and integrating it into an on-line resource (view) easily accessible by potential information seekers. For instance, many WWW users may be interested in all aspects of travel information including car rental and hotel fares, special bargains and flight availabilities of different airlines. While such information could principally be retrieved by each of the interested customers by querying many ISs and integrating the results into a meaningful answer, it is much preferable if one *travel consolidator service* were to collect such travel-related information from different airlines and travel agent sources on the WWW and to organize such information into materialized views. Besides providing simplified and customized information access to customers who may not have the time nor skill to identify and retrieve relevant information from all sources, materialized views may also offer more consistent availability – shielding customers from the fact that some of the underlying ISs may temporarily become disconnected as well as offering better query performance as all information can be retrieved from a single location.

However, views in such evolving environments introduce new challenges to the database community [Wid95]. One important and as of now not yet addressed problem for these applications is that current view technology only supports *static apriori-specified* view definitions – meaning that views are assumed to be specified on top of a fixed environment [LNR97, RLN97]. Once the underlying ISs change their capabilities, the views derived from them may become undefined. It is this problem of view evolution caused by external environment changes (at the schema level rather than at the data level as done by practically all previous work on view maintenance [BCL89, Wid95, ZGMHW95]) that we tackle in this paper. We call this the *view synchronization* problem [RLN97]. There are two exceptions to this previous view maintenance work for data changes, namely by Gupta et al. [GJM96] and Mohania et al. [MD96]. While we assume that the evolution of the affected view definitions is triggered by capability changes of ISs, Gupta and Mohania assumed that view redefinition was explicitly requested by the user at the view site. Hence, previous work on view redefinition did not deal with the problem of how to salvage the affected view definitions itself (at the schema level) but was exactly told how to modify it. Instead they dealt with efficiently managing changes at the data level to now comply with the modified view definition. Our problem and solution thus is complimentary to work by others as once we have determined an acceptable view redefinition then algorithms proposed by others [GJM96, MD96] on how most efficiently to maintain the view, if materialized, could be applied to our system.

The issues associated with this new problem are now explained by the following example of a travel scenario, which will serve as the basis for all examples throughout the remainder of the paper.

Example 1 *Assume a traveller plans to visit Boston in one month for pleasure. To make his stay in Boston without last minute hastiness, he would like to make arrangements for car rental and hotel stay. The query for getting the necessary information can be specified as an SQL view definition as follows:*

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & \textit{Travel-Info-in-Boston}\ \textsf{AS} \\
\textsf{SELECT} & C.Name,\ C.Address,\ C.Phone,\ H.Name,\ H.Address,\ H.Phone \\
\textsf{FROM} & CarRental\ C,\ Hotel\ H \\
\textsf{WHERE} & (C.City = \text{`Boston'})\ AND\ (C.State = \text{`MA'})\ AND \\
 & (H.City = \text{`Boston'})\ AND\ (H.State = \text{`MA'})
\end{array}
\tag{1}
$$

where CarRental and Hotel are relations that contain the car rentals and lodging information in Boston only.

Assume for some reason the Hotel relation cannot be accessed (this effect could be caused by the IS that provided the Hotel relation to go out of business). Executing the Travel-Info-in-Boston query to get requested data (or to materialize the view) will then cause an error message such as "Error: the Hotel relation is undefined". This is state-of-the-art view technology. We, on the other hand, propose that are may be many potential ways to "remedy" this view definition evolution. To name a few:

1. *Assume there is a EastRegionHotel relation that has the lodging information for the entire eastern region of the USA (that is, EastRegionHotel $\supseteq$ Hotel). Query 1 can be rewritten to have the Hotel relation replaced by the EastRegionHotel relation. This would return the initially expected answer plus possibly additional hotels not in Boston.*
2. *Assume there is a BackBayHotel relation that contains the lodging information in the Back Bay area only (that is, BackBayHotel $\subset$ Hotel). Query 1 can be rewritten to have the Hotel relation replaced by the BackBayHotel relation, which is likely to return a useful answers for the traveller but it will not be a complete listing of all answers for the initial query.*
3. *The traveller may even be content to have the car rental information only, since with a car he can drive around and find a hotel after he arrives in Boston. In this case, removing the Hotel relation and the attributes referencing the Hotel relation from the Travel-in-Boston query is acceptable to the user.*

As illustrated in Example 1, there may be many alternative ways to salvage the affected view definition. The research questions that we hence attempt to answer are:

1. How do we determine which among these possible alternative synchronization options are acceptable to the user (as they are not necessarily equivalent)?
2. What type of information must be available to *EVE* in order to provide sufficient information for finding appropriate replacements for the affected components of a view definition?
3. What are the criteria for a synchronized view definition to be considered correct?
4. What are appropriate strategies for finding correct view synchronizations (replacements) for affected views?

1.2 The *EVE* Approach

In this paper, we define a novel paradigm towards addressing the view synchronization problem that provides a solution to all of the above research questions. We put forth that it is important for the person in charge of defining the virtual information resource (i.e., view) to be able to express preferences about the view evolution process (instead of our system making automatic and generic choices) – as these view definers are the ones that know the criticality and dispensability of the different components of a view for applications and end users of the view.

As these view evolution preferences refer to specific components of view definition, in our system the view definer can directly embed their preferences about view evolution into the view definition itself. We design an extended view definition language (a derivative of SQL, which we call Evolvable-SQL or short E-SQL) that incorporates user preferences for change semantics of the view (see Section 4). Such view preference specification would allow us to avoid human interaction each and every time a change occurs in the environment.

To facilitate the replacement finding task, we exploit a model for information source description (MISD) for capturing the capabilities of each IS as well as the interrelationships between ISs. Similar to the University of Michigan Digital Library system [NR97] and the Garlic project [CHA+95], each IS registers its description expressed by this model in a Meta Knowledge Base (MKB) when joining the system. This Meta Knowledge Base (MKB) thus represents a resource that can be exploited when searching for an appropriate substitution for the affected components of a view in the global environment.

Based on this solution framework of E-SQL and the MISD, we introduce strategies for evolving views transparently. Our proposed view rewriting process, which we call *view synchronization*, finds a view redefinition that meets all view preservation constraints specified by the E-SQL view definition (VD). That is, it identifies and extracts appropriate information from other ISs as replacement of the affected components of the view definition and produces an alternative view definition.

Our goal is to "preserve as much as possible" of the original view extent of the affected view definitions instead of completely disabling them with each IS change [LNR97, RLN97]. To the best of our knowledge, our work is the first to study this view synchronization problem, and no alternate framework designed to solve this problem has been put forth thus far. A *EVE* prototype system has been implemented using Java, JDBC, Oracle, and MSAccess, and it is running at the Worcester Polytechnic Institute, and has been shown in CASCON'97 Technology Showcase ([LNR97]).

1.3 Outline of Paper

The remainder of the chapter is structured as follows. In Section 2, we present the *EVE* framework, and in Section 3 we introduce a web-based travel agency example used as running example throughout the paper. The extended view definition language, E-SQL, designed to add flexibility to current view technology is presented in Section 4. In Section 5, we present the information source description model (MISD), while criteria for selecting appropriate substitutions for view components are given in Section 6. In Section 7, we give our algorithms for the view synchronization problem. Section 8 lists related work in the literature, and Section 9 presents our conclusions.

2 Evolvable View Environment (*EVE*) Framework

Our *view synchronization process* tempts to evolve views, when they are affected by capability changes triggered by the participating ISs. Next, we present the Evolvable View Environment (*EVE*) framework that we propose for tackling the view synchronization problems in dynamic environments (Figure 1). We give an architectural overview of the *EVE* framework next and outline our key design decisions.

IS Registration. Our environment can be divided into two spaces, i.e., the view space and information space. The information space is populated by a large number of external ISs. External ISs are heterogeneous and distributed. Most importantly, they are dynamic and can autonomously change their capabilities, when desired. They could even join or leave the system at any time. An IS is "integrated" in the global framework via a wrapper that serves as a bridge between the information space and the view space. The main functionality of a wrapper is to translate the messages specified in the underlying data definition/manipulation languages into a common language used in the view site, and vice versa. The wrapper is assumed to be intelligent so that it can extract not only raw data, but also meta information about the IS, such as changes at the schema level of the IS, performance data, or relationships with other ISs. Any IS that supports a query interface can participate in our environment.

Meta Knowledge Base (MKB). When an IS joins *EVE*, it advertises to the MKB its capabilities, data model (e.g., the semantic mappings from its concepts to the concepts already in the MKB), and data content. The

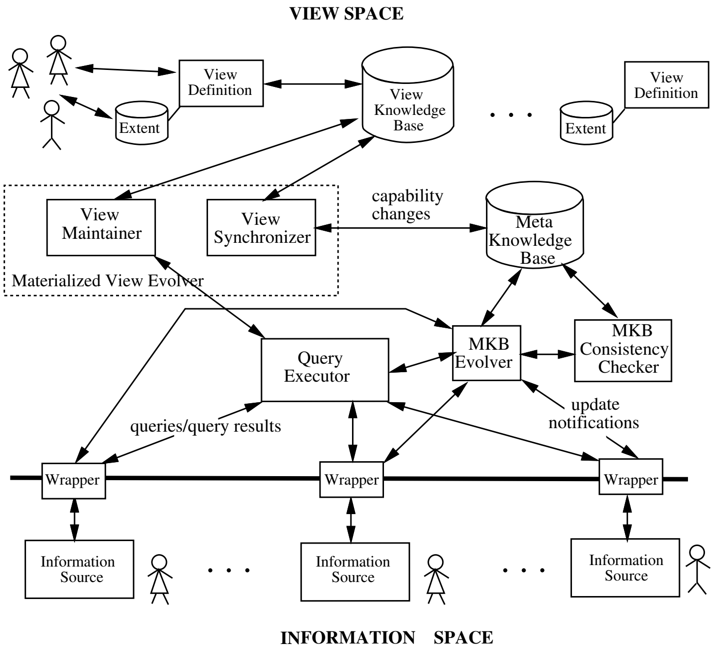


Figure 1: The Framework of Evolvable View Environment (EVE).

information providers have strong economic incentives to provide the meta knowledge of their individual ISs as well as the relationships with other ISs, since populating the MKB makes their data known by the view users, and thus increases the data utilization of their data set (especially, if they offer the same information at a better price).

We have designed a model for information source descriptions (MISD) [LNR97, RLN97] that is capable of describing the content and capabilities of heterogeneous ISs. MISD captures meta knowledge such as an attribute must have a certain type (type integrity constraint), one relation can be meaningful joined with another relation if certain join constraints are satisfied (join constraint), a fragment of a relation is partially or completely contained in another fragment of some other relation (partial/complete information constraint), and so on (see Section 5). The IS descriptions collected in the MKB form an information pool that is critical in finding appropriate replacements for view components when view definitions become undefined (See Section 5) and for translating loosely-specified user requests into precise query plans [NR97].

MKB Evolution. When an underlying IS makes a change to its capabilities (e.g., adds a new relation), the MKB no longer reveals the IS correctly in the sense that the meta knowledge describing the IS and the actual capabilities of the IS are distinct. For this, we have designed the MKB Evolution process to react to capability changes in the information space. In our framework, each IS will via the wrapper interface notify the MKB of any such capability changes so that they can be properly registered in the MKB. The MKB Evolver module will then take appropriate actions to update the MKB [NLR97]. For example, deleting an attribute A from a relation S may cause the MKB evolver to modify a subset constraint between two relations S and R, e.g., "$S \subset R$", into the constraint "$S \subset$ (project all attributes of R besides A from R)". In other cases, some constraints may have to be completely removed from the MKB if they contain references to the deleted attribute.

View Maintenance. The *view maintainer* tool (Figure 1) in general is in charge of propagating data updates executed on an IS site to all affected views. In our system, this tool will also be in charge of bringing the view content up-to-date after the view definition already has been changed by the view synchronizer in response to a capability change.

View Synchronization. The *view synchronizer* tool (Figure 1) evolves affected views transparently according to users' preferences expressed by our extended view definition language E-SQL. View synchronization is the focus of this paper, and we will present replacement strategies and view synchronization algorithms in later sections.

Global Consistency Checking Across Sources. There are two types of inconsistencies (related to meta knowledge) in *EVE*. The first one is that constraints expressed in the MKB do not correspond to the information actually provided by ISs; and the second one is that different assertions in the MKB contradict each other. The first type of inconsistency occurs when (1) either an IS provider makes an error when entering a MISD description, (2) an update occurred at one IS that causes a constraint that used to hold to become invalid, or (3) the usage and hence content of an IS changes over time without proper notification to the MKB. For example, the information provider for IS_1 inserts the fact that the relation R is equivalent to a relation S in another site IS_2 into the MKB. Now, the provider of IS_2, that is not aware of this assertion made about S in IS_2, inserts a new tuple t that makes the assertion become false.

There are alternative approaches for resolving this inconsistency. For example, (1) insert the tuple t into the relation R as well, (2) reject the insertion into S, (3) modify the invalid assertion in the MKB so to make it valid (i.e., in this case change "$IS_1.R \equiv IS_2.S$" into "$IS_1.R \subset IS_2.S$"), or (4) remove the invalid assertion from the MKB. Since checking and enforcing constraints across distributed autonomous ISs is an extremely difficult problem all on its own, in this work we assume that providers of individual ISs are in charge of assuring that their data is consistent with the meta knowledge collected in the MKB. We do not at this time incorporate a tool into our *EVE* framework that resolves possible inconsistencies. However, once being notified about the entry or removal of some data item by an IS, *EVE* will notify the creators of all constraints in the MKB that may possibly be violated by this data modification. For example, on inserting a new tuple t into the relation S in the above example, both the providers of S and R are notified that the update occurred and that the constraint "$IS_1.R \equiv IS_2.S$" may now be inconsistent. It is up to the providers of IS_1 and IS_2 to determine how to handle this situation, once given the notification.

MKB Consistency. The second type of MKB consistency concerns conflicts between the constraints entered in the MKB, and thus can be detected by our MKB Consistency Checker module without help from the IS providers. One example of this type of conflict is that one information provider declares that a relation R of IS_1 is a strict subset of a relation S in another site IS_2, and at the same time the provider of S claims that the extent of S is a strict subset of R. This is clearly an inconsistency. Our MKB consistency checker discovers such controversial meta knowledge using various types of inference techniques. Once detected, inconsistent assertions are reported to responsible information providers to have the differences resolved.

3 Running Example: The Travel Consolidator Service

To demonstrate our solution approach, we use a travel consolidator service provider as running example throughout this paper. Below we describe the relevant information sources (expressed using relations in our system) and two example SQL views, while additional relations and views are added later in the paper, as needed.

Example 2 *Consider a large travel consolidator which has a headquarter in Detroit, USA, and many branches all over the world. It helps its customers to arrange flights, car rentals, hotel reservations, tours, and purchasing insurances. Therefore, the travel consolidator needs to access many disparate information sources, including domestic as well as international sites. Since the connections to external information sites, such as the overseas branches, are very expensive and have low availability, the travel consolidator materializes the query results (views) at its headquarter or other US branches (at the view site). Some of the relevant ISs are listed in the table in Figure 2.*

Assume the headquarter maintains complete sets of information of the customers, tours, and tour participants in the following formats: Customer(Name, Address, Phone, Age), Tour(TourID, TourName, Type, Duration) - where

IS 1: *Customer Information* *Customer(Name, Address, Phone, Age)*
IS 2: *Tour Information* *Tour(TourID, TourName, Type, Duration)*
IS 3: *Tour Participant Information* *Participate(Participant, TourID, StartingDate)*
IS 4: *Flight Reservation Information* *FlightRes(PName, Airline, FlightNo, Source, Dest, Date)*
IS 5: *Insurance Information* *Accident−Ins(Holder, Amount, Birthday)*
IS 6: *Car Rental Information* *CarRental(Name, Address, Phone, City, State, Country)*
IS 7: *Hotel Information* *Hotel(Name, Address, Phone, City, State, Country)*

Figure 2: Descriptions of Relevant Information Sources.

Type = {luxurious, economy, super−valued}, and Participate(Participant, TourID, StartingDate) that states which customer joins which tour starting on what day. We further assume the local branches keep partial sets of information of its local customers, the tours offered locally, and the participation information of its local customers. The flight reservation information FlightRes(PName, Airline, FlightNo, Source, Dest, Date) is managed by each individual airline company. Insurance information Accident−Ins(Holder, Amount, Birthday) is kept by each individual insurance company. The car rental company and lodging information, CarRental(Name, Address, Phone, City, State, Country) and Hotel(Name, Address, Phone, City, State, Country), are managed by each individual company, respectively.

Let's assume that the travel agency has a promotion for the customers who travel to Asia. Therefore, the travel agency needs to find the customers' names, addresses, and phone numbers in order to send promotion letters to these customers or call them by phone. The view query for getting the necessary information can be specified as follows:

```
CREATE VIEW   Asia-Customer AS
SELECT        Name, Address, Phone
FROM          Customer C, FlightRes F                                  (2)
WHERE         (C.Name = F.PName) AND (F.Dest = 'Asia')
```

In addition, the travel consolidator wants to study the correlation between the type of tour a customer (older than 18) participates in and the type of accidental insurance she buys. Therefore, the second view query can be specified as:

```
CREATE VIEW   Tour-Insurance AS
SELECT        C.Age, T.Type, T.Duration, I.Amount
FROM          Customer C, Participate P, Tour T, Accidental-Insurance I    (3)
WHERE         (C.Name = P.Participant) AND (P.TourID = T.TourID) AND
              (C.Name = I.Holder) AND (C.Age > 18)
```

Note that Queries 2 and 3 are static apriori-specified queries. We use this travel consolidator service example to demonstrate the usage of and interactions among proposed evolution parameters in later sections.

4 E-SQL: The View Definition Language

A novel principle of our approach is to explore the evolution of an affected view based on preferences by its definer. In this section, we thus design the *EVE* view definition language for evolvable views, called Evolvable-SQL or E-SQL, for this purpose. E-SQL is an extension of the SELECT-FROM-WHERE SQL syntax augmented with specifications for how the view definition may be evolved under IS capability changes. *EVE* attempts to salvage the affected views by following the evolution preferences expressed in the evolution parameters of the E-SQL view definitions. The general format of an E-SQL view definition V is given in Query (4) in Figure 3.

$$\begin{array}{ll} \text{CREATE VIEW} & V\ (B_1, \ldots, B_m)\ \ (\mathcal{VE} = \mathcal{VE}_V)\ \text{AS} \\ \text{SELECT} & R_1.A_{1,1}(\mathcal{AD} = \mathcal{AD}_{1,1}, \mathcal{AR} = \mathcal{AR}_{1,1}), \ldots, R_1.A_{1,i_1}(\mathcal{AD} = \mathcal{AD}_{1,i_1}, \mathcal{AR} = \mathcal{AR}_{1,i_1}), \ldots, \\ & R_n.A_{n,1}(\mathcal{AD} = \mathcal{AD}_{n,1}, \mathcal{AR} = \mathcal{AR}_{n,1}), \ldots, R_n.A_{n,i_n}(\mathcal{AD} = \mathcal{AD}_{n,i_n}, \mathcal{AR} = \mathcal{AR}_{n,i_n}) \\ \text{FROM} & R_1(\mathcal{RD} = \mathcal{RD}_1, \mathcal{RR} = \mathcal{RR}_1), \ldots, R_n(\mathcal{RD} = \mathcal{RD}_n, \mathcal{RR} = \mathcal{RR}_n) \\ \text{WHERE} & C_1(\mathcal{CD} = \mathcal{CD}_1, \mathcal{CR} = \mathcal{CR}_1)\ AND\ \ldots\ AND\ C_k(\mathcal{CD} = \mathcal{CD}_k, \mathcal{CR} = \mathcal{CR}_k) \end{array} \tag{4}$$

Figure 3: Syntax of a Generic E-SQL View Definition.

In Figure 3, the set $\{B_1, \ldots, B_m\}$ corresponds to the local names given to attributes preserved in the view V, the set $\{A_{s_{j,1}}, \ldots, A_{s_{j,i_j}}\}$ is a subset of the attributes of relation R_j with $j = 1, \ldots, n$; C_i with $i = 1, \ldots, k$, are primitive clauses defined over the attributes of relations in the FROM clause. A primitive clause has one of the following forms: $(< attribute - name > \theta < attribute - name >)$ or $(< attribute - name > \theta < value >)$ with $\theta \in \{<, \leq, =, \geq, >\}$. And, all parameters $\mathcal{VE}, \mathcal{AD}, \mathcal{AR}, \mathcal{RD}, \mathcal{RR}, \mathcal{CD}$ and $\mathcal{CR}$ and their respective values are defined as given in the table in Figure 4.

Query 4 corresponds to a SELECT-FROM-WHERE SQL query augmented with evolution parameters. For simplicity's sake, we assume in this work that views are defined by SQL queries in the format of Query (4) with a conjunction of primitive clauses in the WHERE clause.

Definition 1 ***View Component.*** *The attributes in the SELECT clause ($\mathcal{A}$), relations in the FROM clause ($\mathcal{R}$), and primitive conditions in the WHERE clause ($\mathcal{C}$) are the basic units of a view. These basic units are called the view components of a view.*

Each view component has attached two evolution parameters. One is the *dispensable parameter*, denoted as $\mathcal{XD}$ where $\mathcal{X}$ could be $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{C}$ for attribute, relation, or condition component, respectively. The *dispensable parameter* states whether the view component is essential and hence must be kept in the evolved view (when the value is false); or the view component could be dropped if a replacement cannot be found (when the value is true). The other is the *replaceable parameter*, denoted as $\mathcal{XR}$ with $\mathcal{X}$ likewise defined as above. The *replaceable parameter* specifies whether the view component could be replaced in the view synchronization process (if its value is true) or the view component cannot be replaced (if the value is false). A view definer can also specify that the evolved view extent could be anything (if the value is "don't care"), or must be equivalent to (if the value is $\equiv$), a superset of (if the value is $\supseteq$), or a subset of (if the value is $\subseteq$), with respect to the original view extent using the $\mathcal{VE}$ parameter.

The evolution parameters $\mathcal{VE}, \mathcal{AD}, \mathcal{AR}, \mathcal{RD}, \mathcal{RR}$, $\mathcal{CD}$, and $\mathcal{CR}$ and their respective values are summaried in Figure 4. Each type of evolution parameter used in E-SQL is represented by a row in the table, with column one giving the parameter name and the abbreviation for the parameter, column two the possible values of the parameter can take on plus the associated semantics, and column three the default value. When the parameter setting is omitted from the view definition, then the default value is assumed. This means that a conventional SQL query

Evolution Parameter		Semantics	Default Value
Attribute-	dispensable ($\mathcal{AD}$)	*true:* the attribute is dispensable *false:* the attribute is indispensable	false
	replaceable ($\mathcal{AR}$)	*true:* the attribute is replaceable *false:* the attribute is nonreplaceable	false
Condition-	dispensable ($\mathcal{CD}$)	*true:* the condition is dispensable *false:* the condition is indispensable	false
	replaceable ($\mathcal{CR}$)	*true:* the condition is replaceable *false:* the condition is nonreplaceable	false
Relation-	dispensable ($\mathcal{RD}$)	*true:* the relation is dispensable *false:* the relation is indispensable	false
	replaceable ($\mathcal{RR}$)	*true:* the relation is replaceable *false:* the relation is nonreplaceable	false
View-	extent ($\mathcal{VE}$)	approximate: no restriction on the new extent $\equiv$: the new extent is equal to the old extent $\supseteq$: the new extent is a superset of the old extent $\subseteq$: the new extent is a subset of the old extent $\approx$: the new extent can be anything	$\equiv$

Figure 4: View Evolution Parameters of E-SQL Language.

(without explicitly specified evolution preferences) has well-defined evolution semantics in our system, i.e., anything the user specified in the original view definition must be preserved exactly as originally defined in order for the view to be well-defined. Our extended view definition semantics are thus well-grounded and compatible with current view technology. Below we now discuss the evolution parameters in detail. Each treatment of the evolution parameters is backed-up with a working example to demonstrate the utility and usage of them.

4.1 Attribute-Dispensable Parameter $\mathcal{AD}$

This parameter tells us whether an attribute from the view interface has to be kept in the modified view definition in order for that view to be acceptable to the view definer. Therefore $\mathcal{AD}$ is associated with the attributes in the SELECT clause. This parameter is of Boolean type. $\mathcal{AD} = true$ means that the attribute is dispensable; whereas $\mathcal{AD} = false$ means that the attribute is essential for the view definition, thus removing the attribute from the view makes the view meaningless to the view user. The default value of the parameter is $false$.

Example 3 *Assume the travel agency is content with the query results of Query 2 with the customer's names and addresses only, i.e., the company is willing to put off the phone marketing strategy if the customer's phone number attribute is deleted from the relation Customer for some reason and a suitable substitute cannot be found. The user can state this preference in the SELECT clause of Query (2) by using the attribute dispensable parameter $\mathcal{AD}$ as follows:*

$$\textsf{SELECT} \qquad Name\ (\mathcal{AD} = false), Address\ (\mathcal{AD} = false), Phone\ (\mathcal{AD} = true)$$

In this example, the view definer instructs the view synchronization system that the customer's name and address are indispensable to the view, since without the customer's names and addresses the travel agency cannot send its promotion letters to the customers. On the other hand, the customer's phone numbers can be omitted from the original view definition, if keeping it becomes impossible.

4.2 Attribute-Replaceable Parameter $\mathcal{AR}$

This parameter is also associated with the attributes in the SELECT clause. It characterizes whether the associated attribute is allowed to be replaced by another attribute other than the original one. $\mathcal{AR} = true$ means the associated attribute is allowed to be substituted with an "appropriate" attribute (see Section 6.1) either from the same IS or from other IS. On the other hand, $\mathcal{AR} = false$ means that the attribute cannot be substituted by any other data. The default value is $false$.

Example 4 *In addition to instructing our system that the customer name and address have to be kept in the view interface, the user may want to further guide our system as to whether it is acceptable for an attribute to be obtained from other sources besides the original relation. For example, if the user only accepts the customer name and address to come from the Customer relation, but agrees to have the phone number come from other source(s), then the user can augment the SELECT clause of Query (2) with the attribute replaceable parameter $\mathcal{AR}$ as follows:*

$$\textit{SELECT} \qquad Name\ (\mathcal{AR} = false), Address\ (\mathcal{AR} = false), Phone\ (\mathcal{AR} = true)$$

4.3 Relation-Dispensable Parameter $\mathcal{RD}$

This parameter is associated with the participating relations in the FROM clause. It tells the view synchronization system whether the accompanied relation is allowed to be dropped from the original view definition. This parameter is of type Boolean. $\mathcal{RD} = true$ indicates the relation is allowed to be dropped, while $\mathcal{RD} = false$ indicates that the information contained in the relation is crucial to the user, therefore dropping the relation makes the view meaningless. The default value of $\mathcal{RD}$ is $false$.

Example 5 *Let's look at the Query 1 specified in Example 1. Originally the query returns the car rental and lodging information in Boston area. If the traveller would be content with the car rental information only when the lodging information cannot be obtained before hand, then the view definer can set the relation dispensable parameter of the Hotel relation to true, i.e., CarRental C ($\mathcal{AD}$ = false), Hotel H ($\mathcal{AD}$ = true). In this case, removing the Hotel relation and the attributes referencing the Hotel relation from Query 1 is acceptable to the user.*

4.4 Relation-Replaceable Parameter $\mathcal{RR}$

This parameter is associated with the participating relations in the FROM clause. It indicates whether the relation is allowed to be substituted by other relations (or a combination of relations). The parameter value is again of Boolean type. $\mathcal{RR} = true$ indicates the relation is allowed to be substituted with some "appropriate" relation in the global environment (see Section 6.1). and $\mathcal{RR} = false$ indicates that the user does not accept any substitution for the original relation. The default value of $\mathcal{RR}$ is $false$.

Example 6 *The user may augment the Customer relation in the FROM clause of Query 2 by setting the relation-replaceable parameter $\mathcal{RR}$ to true, i.e., Customer C ($\mathcal{RR}$ = true). In this case, if if the Customer relation is removed from its IS (the headquarter), then our system can substitute it with a redundant Customer relation managed at the Asia branch. This may be a less desirable view definition in the sense of resulting in a view that is more expensive to maintain and due to network delay more likely to be not completely up-to-date, but preferable over not having any view supported at all.*

4.5 Condition-Dispensable Parameter $\mathcal{CD}$

This parameter indicates whether a condition in the WHERE clause is allowed to be dropped, e.g., it happens when any one of the condition's operands is no longer available. This parameter, with type Boolean, is associated with the conditions in the WHERE clause. $\mathcal{CD} = true$ means the associated condition may be dropped if it cannot be kept any longer in the view definition, while $\mathcal{CD} = false$ means the condition cannot be dropped for the view definition to still be meaningful to the view user. The default value of $\mathcal{CD}$ is $false$.

Example 7 *Let's look at the WHERE clause of Query (2) with two conditions. The first condition is an equijoin condition that joins the Customer relation with the FlightRes relation by customer' names, and the second one is a local condition specified on the relation FlightRes that finds all the passengers who travel to Asia. Assume the view definer of the view* ***Asia-Customer*** *is willing to accept a view without the second (local) condition specified, as long as the equijoin condition is kept*[1]*. That is, if the second (local) condition is dropped, i.e., the destination information is not kept in the FlightRes relation anymore, then the promotion invitation letters are sent to all customers traveling by air. This preference would be expressed in E-SQL by adding the condition-dispensable parameter to the conditions in the WHERE clause of Query (2) as:*

$$(C.Name = F.Passenger)(\mathcal{CD} = false)\ AND\ (F.Dest = Asia)(\mathcal{CD} = true)$$

4.6 Condition-Replaceable Parameter $\mathcal{CR}$

This (Boolean) parameter is also associated with the conditions in the WHERE clause. It characterizes whether the associated condition is allowed to be replaced by an "appropriate" condition[2]. $\mathcal{CR} = true$ means the condition is allowed to be replaced by a semantically equivalent condition with operand(s) either from the same or from other information source(s). On the other hand, $\mathcal{CR} = false$ means that the condition cannot be substituted, i.e., the information is sensitive and uses of the information do not trust the reliability of alternate ISs. The default value of $\mathcal{CR}$ is $false$.

Example 8 *Let's look at the view* ***Tour-Insurance*** *specified in Example 3. Assume the user allows the condition $(C.Age > 18)$ to be replaced by a semantically equivalent condition expressed by:*

$$(C.Age > 18)(\mathcal{CR} = true)$$

Then when the information provider of the relation Customer decides to drop its attribute Age, our system can replace $(C.Age > 18)$ with $(((today - I.Birthday)/365) > 18)$ to preserve the original WHERE clause.

4.7 View-Extent Parameter $\mathcal{VE}$

This parameter is specified for the view as a whole. It instructs the view synchronization system whether the view extent of the evolved view must be equivalent to ($\equiv$), a superset of ($\supseteq$), or a subset of ($\subseteq$) the original view extent in terms of the common subset of attributes in the original and the evolved view definitions. If no restrictions on the view extent are given, then $\mathcal{VE}$ is set to "approximate". The default value of $\mathcal{VE}$ is ($\equiv$).

[1] Note that in general dropping a local condition is more acceptable than dropping a join condition, since dropping a join condition may change the view definition dramatically. For example, replacing a join condition that returns some subset of tuples by a Cartesian product which then would return all pairwise combinations of tuples from both relations as view result.

[2] We give a formal definition of what we define as appropriate substitution in the context of this work in Section 6.2.1.

Example 9 *The user may augment Query (2) with the view-extent parameter as:*

$$\textsf{CREATE VIEW} \quad \textbf{\textit{Asia-Customer}}\ \textsf{AS}\ (\mathcal{VE} = \text{“} \supseteq \text{”})$$

This means any new view definition must return a view extent equivalent to or larger than the original view extent for the view evolution process to be valid. That is, if originally the ***Asia-Customer*** *view returns the customers who travel to Japan, Korea, or Hong Kong, then the view is still valid if in addition to these customers it also returns the customers who travel to Thailand or Malaysia.*

4.8 Putting Evolution Parameters all Together

Example 10 *Putting together all view evolution parameters for Query (2), we get Query (5).*

$$\begin{array}{ll} \textsf{CREATE VIEW} & \textit{Asia-Customer } (\mathcal{VE} = \text{“} \supseteq \text{”})\ \textsf{AS} \\ \textsf{SELECT} & \textit{Name, Address, Phone } (\mathcal{AD} = true,\ \mathcal{AR} = true) \\ \textsf{FROM} & \textit{Customer C } (\mathcal{RR} = true),\ \textit{FlightRes F} \\ \textsf{WHERE} & (C.Name\ =\ F.PName)\ AND\ (F.Dest\ =\ \text{'Asia'})\ (\mathcal{CD} = true) \end{array} \tag{5}$$

Note that for the view components that have their evolution parameter values omitted, the default value is assumed as indicated in Figure 4. To name a few, the attributes Name and Address in the SELECT *clause are indispensable, and the relation FlightRes is indispensable and nonreplaceable.*

5 MISD: Model for Information Source Description

Information sources may be constructed using different data models, and the wrapper of each information source expresses the capabilities of its underlying information source into a common simple model that is understood by our *EVE* system. MISD allows a large divergent class of ISs to participate in *EVE*. Figure 5 summarizes the type of constraints supported in our current system. Note that other constraints such as key or foreign key constraints could easily be added in the future. These descriptions are collected in a Meta Knowledge Base (MKB) (see Figure 1), forming an information pool that is critical in finding appropriate replacements for view components when view definitions become undefined.

Name	Syntax
Type Integrity Constraint	$\mathcal{TC}_{R.A_i}\ =\ (R(A_i) \subseteq A_i(Type_i))$
Order Integrity Constraint	$\mathcal{OC}_R\ =\ (R(A_1,\ldots,A_n) \subseteq \mathcal{C}(A_{i_1},\ldots,A_{i_k}))$
Join Constraint	$\mathcal{JC}_{R_1,R_2} = (C_1 AND \cdots C_l)$
Partial/Complete Constraint	$\mathcal{PC}_{R_1,R_2} = (\pi_{A_{i_s}}(\sigma_{C(A_{j_1},\ldots,A_{j_l})}R_1)\ \theta\ \pi_{A_{n_s}}(\sigma_{C(A_{m_1},\ldots,A_{m_t})}R_2))$

Figure 5: Possible Types of Semantic Constraints for IS Descriptions.

5.1 Data Content Description

The model used to describe the basic units of information available in each of the ISs is the relational model. An IS has a set of relations $IS.R_1$, $IS.R_2, \ldots, IS.R_n$. A base relation is an n-ary relation with $n \geq 2$. A relation name is not required to be unique in the MKB, but the pair (IS name, relation name) is. That is, if the information source IS exports the relation R then $IS.R$ is assumed to be unique in the MKB. A relation R is described by specifying its information source and the set of attributes belonging to it as follows:

$$IS.R(A_1, \ldots, A_n). \tag{6}$$

5.2 Type Integrity Constraints

The domain types of the attributes A_i are described using *type integrity constraints*, denoted by $A_i(Type_i)$. A *type constraint* for a relation $R(A_1, \ldots, A_n)$ is specified as:

$$\mathcal{TC}_{R(A_i)} \;=\; (\; R(A_i) \subseteq A_i(Type_i) \;) \tag{7}$$

where $A_i(Type_i)$ can be viewed as a one-column relation with domain type $Type_i$. The type integrity constraint of $\mathcal{TC}_{R(A_i)}$ says that any of the possible values of the attribute A_i is contained in the relation $A_i(Type_i)$. The type integrity constraints of the attributes A_1 to A_n of the relation R can be combined into a single type integrity constraint as follows:

$$\mathcal{TC}_{R(A_1,\ldots,A_n)} \;=\; (\; R(A_1, \ldots, A_n) \subseteq A_1(Type_1) \times \ldots \times A_n(Type_n) \;) \tag{8}$$

which says that the attribute A_i is of domain type $Type_i$, for $i = 1, \ldots, n$. For simplicity, we assume that the attribute types are primitive. If two attributes are exported with the same name, they are assumed to have the same type (which must be reflected by the type integrity constraints for their relations)[3].

5.3 Order Integrity Constraints

An *order integrity constraint* on a relation R specifies constraints on the tuples in R, such that the tuples must satisfy the order constraint at any time. For a relation $R(A_1, \ldots, A_n)$, a generic order constraint is specified as follows:

$$\mathcal{OC}_R \;=\; (\; R(A_1, \ldots, A_n) \subseteq \mathcal{C}(A_{i_1}, \ldots, A_{i_k}) \;) \tag{9}$$

where $A_{i_s} \in R$ for s = 1, …, k, and $\mathcal{C}(A_{i_1}, \ldots, A_{i_k})$ is a conjunction of primitive clauses defined over the attributes. A primitive clause has one of the following forms: (<attribute-name> θ <attribute-name>) or (<attribute-name> θ <value>) with $\theta \in \{<, \leq, =, \geq, >\}$. Expression (9) specifies that for any state of the database R and for any tuple $t \in R$, $\mathcal{C}(t[A_{i_1}], \ldots, t[A_{i_k}])$[4] is satisfied.

Example 11 *An insurance relation* ***Expensive-Insurance****, containing all expensive accidental insurances that cover more than* $1,000,000$*, can be expressed by the following order constraint:*

$$\textit{Expensive-Insurance}(Holder, Type, Amount, Birthday) \subseteq (Amount > 1,000,000).$$

[3] In the future, we plan to allow complex types and a hierarchy of types. We anticipate that most of the proposed solution approach will continue to apply to these extended types.

[4] The expression $t[A]$ refers to the value of the attribute A in the tuple t.

5.4 Join Constraints

A join constraint between two relations R_1 and R_2, denoted as $\mathcal{JC}_{R_1,R_2}$, states that tuples in R_1 and R_2 can be meaningfully joined if the join condition, i.e., a conjunction of primitive clauses, is satisfied. A generic join constraint is as follows:

$$\mathcal{JC}_{R_1,R_2} = (C_1 \ AND \ \cdots \ AND \ C_l) \tag{10}$$

where $C_1, \ldots, C_l$ are primitive clauses over the the attributes of R_1 and R_2.

Example 12 *Some of the join constraints for our running example presented in Section 3 are given in the table of Figure 6.*

$\mathcal{JC}$	*Join Constraint*
JC1	*Customer.Name = FlightRes.PName*
JC2	*(Customer.Name = Accident–Ins.Holder) AND (Customer.Age > 1)*
JC3	*Customer.Name = Participate.Participant*
JC4	*Participate.TourID = Tour.TourID*
JC5	*FlightRes.PName = Accident–Ins.Holder*

Figure 6: Relevant Join Constraints for Example 2

5.5 Partial/Complete Information Constraints

A *partial/complete* ($\mathcal{PC}$) constraint between two relations R_1 and R_2 states that a (horizontal and/or vertical) fragment of R_1 is semantically contained or equivalent to a (horizontal and/or vertical) fragment of R_2 at all times. *EVE* makes use of the $\mathcal{PC}$ constraints to decide if an evolved view extent is equivalent, subset of, or superset of the initial view extent. A generic $\mathcal{PC}$ information constraint between two relations R_1 and R_2 is specified as follows:

$$\mathcal{PC}_{R_1,R_2} = \ (\ \pi_{A_{i_1},\ldots,A_{i_k}}(\sigma_{\mathcal{C}(A_{j_1},\ldots,A_{j_l})}R_1) \ \ \theta \ \pi_{A_{n_1},\ldots,A_{n_k}}(\sigma_{\mathcal{C}(A_{m_1},\ldots,A_{m_t})}R_2) \ ) \tag{11}$$

where $A_{i_1}, \ldots, A_{i_k}, A_{j_1}, \ldots, A_{j_l}$ are attributes of R_1; $A_{n_1}, \ldots, A_{n_k}, A_{m_1}, \ldots, A_{m_t}$ are attributes of R_2; $\mathcal{TC}(R_1.A_{i_s}) = \mathcal{TC}(R_2.A_{n_s})$, for s = 1, ...,k; and θ is $\{\subseteq, \supseteq, \equiv\}$ for the partial ($\subseteq$ and $\supseteq$) or complete ($\equiv$) information constraint, respectively.

Example 13 *Let Customer(Name, Address, Phone, Age) be a relation that maintains all the customer information at the headquarter site, and MABranch(Name, Address) be a relation that manages the customers who reside in Massachusetts. The $\mathcal{PC}$ constraint shown in Eq. (12) states that the MABranch relation is contained in the Customer relation:*

$$\mathcal{PC}_{MABranch,Customer} \ = \ (\ \pi_{Name,Address}(MABranch) \ \supseteq \pi_{Name,Address}(Customer) \ ) \tag{12}$$

6 View Evolution Foundations

Given a capability change of an underlying IS, *EVE* finds views in the VKB affected by the capability change. The view synchronizer in *EVE* attempts to salvage these views by finding appropriate replacements for the affected view components. In this chapter, we first define what constitutes a "legal" view rewriting of an affected view; and then introduce replacement strategies for substituting various affected view components.

6.1 Formal Foundation for View Synchronization

In this section we give a formal definition of what is considered to be a *legal* view rewriting for a view which became obsolete after a capability change of an underlying information source. First we introduce some basic definitions that are used in the legal view rewriting definition.

Definition 2 ***Affected View.*** *A view is "affected" by a* delete-attribute/delete-relation *capability change if the deleted capability is referred to in the SELECT, FROM, and/or WHERE clause(s) of the view.*

Definition 3 ***Amendable View.*** *An affected view defined as above is "amendable", if none of affected view components has its evolution parameters set to* $(false, false)$.

Definition 4 ***Evolution Parameter Assignment.*** *When a view component C' is used to replace an affected view component C, the evolution parameters associated with C' are set by the following rules:*

- <u>*Rule 1.*</u> *If C' is used to replace exactly one view component C of the original view V, the new evolution parameters are set to be the same as those of the original C. If a view component is replaced by more than one new view component, we say that each of the new view components replaces exactly one view component.*
- <u>*Rule 2.*</u> *If a new view component C' is used to replace more than one view component of the original view* $X_1(par_{1,1} = val_{1,1}, par_{1,2} = val_{1,2}), \ldots, X_k(par_{k,1} = val_{k,1}, par_{k,2} = val_{k,2})$ *where* $par_{i,j}$ *are view evolution parameters and* $val_{i,j} \in \{true, false\}$, *we set the evolution parameters of C' as:* $(par_1 = val_{1,1}\ AND\ \cdots\ AND\ val_{k,1},\ par_2 = val_{1,2}\ AND\ \cdots\ AND\ val_{k,2})$.

Next, we show examples applying Rules 1 and 2, respectively.

Example 14 *An example when Rule 1 is applied is given first. Let a view $V1$ be defined as follows:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V1\ \textsf{AS} \\ \textsf{SELECT} & R.A\ (\mathcal{AR} = true),\ R.B \\ \textsf{FROM} & R \\ \textsf{WHERE} & \mathcal{C} \end{array} \tag{13}$$

Assume R.A is deleted from its site, and R.A is referenced in the SELECT clause, but not in the WHERE clause. We further assume the view synchronizer finds a counterpart in another relation S.A, which can be joined with R based on attributes other than R.A, i.e., $\mathcal{JC}_{R,S} = (R.C = S.D)$, *where* $R.C \neq R.A$. *Therefore, one rewriting is as follows:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V1'\ \textsf{AS} \\ \textsf{SELECT} & \underline{S.A\ (\mathcal{AR} = true)},\ R.B \\ \textsf{FROM} & \underline{R, S\ (\mathcal{RR} = true)} \\ \textsf{WHERE} & \mathcal{C}\ AND\ \underline{(R.C = S.D)\ (\mathcal{CR} = true)} \end{array} \tag{14}$$

In this example, the view component $R.A$ and its associated evolution parameters ($\mathcal{AR} = true$) are replaced by three new view components, all of which are underlined in Query (14) (using Rule 1 from Def. 4). Each of the new view components has its evolution parameters set equal to that of the replaced view component, i.e., ($\mathcal{AR} = true$) and $\mathcal{AD}$ has the default value.

Example 15 *The next example shows how Rule 2 is applied. Let a view $V2$ be defined as follows:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V2\ \textsf{AS} \\ \textsf{SELECT} & R.A\ (\ \mathcal{AD} = false,\ \mathcal{AR} = true),\ R.B \\ \textsf{FROM} & R,\ T \\ \textsf{WHERE} & (R.A = T.E)\ (\mathcal{CD} = true, \mathcal{CR} = true)\ AND\ \mathcal{C} \end{array} \tag{15}$$

We have the same assumptions as above, except for the fact that $R.A$ is being referenced in the SELECT *and in the* WHERE *clauses. Therefore, one rewriting is as follows:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V2'\ \textsf{AS} \\ \textsf{SELECT} & \underline{S.B\ (\mathcal{AR} = true)},\ R.B \\ \textsf{FROM} & R,\ T,\ \underline{S\ (\mathcal{RR} = true)} \\ \textsf{WHERE} & \underline{(S.B = T.E)\ (\mathcal{CD} = true, \mathcal{CR} = true)}\ AND \\ & \mathcal{C}\ AND\ \underline{(R.C = S.D)\ (\mathcal{CR} = true)} \end{array} \tag{16}$$

In this example, there are four new view components (underlined) in $V2'$. Two among the four, S in the FROM *clause and ($R.C = S.D$) in the* WHERE *clause, are brought in by the overall replacement process for replacing two old view components – $R.A$ in the* SELECT *clause and ($R.A = T.E$) in the* WHERE *clause of $V2$. Therefore, their evolution parameters are set using Rule 2. That is, the evolution parameters of S and ($R.C = S.D$) are both set to ($false, true$).*

Definition 5 ***Legal Rewriting.*** *Given a capability change $\mathcal{CC}$ and an amendable view V, V' is a* legal *view rewriting for V if the following properties hold:*

P1. The view rewriting V' is no longer affected by the capability change $\mathcal{CC}$, by either dropping or replacing the affected view components in V.

P2. V' is well-defined and can be evaluated in the evolved state of the MKB after the capability change [5]. That is, any attributes and relations referred to in V' must be registered in the new state of the MKB.

P3. New view components are added to V' only if they are used to replace some view components in V. That is, new view components are introduced into V' with some purpose.

P4. The evolution preference conveyed by the evolution parameters (ignoring the view-extent parameter) specified in the view V are satisfied by V'. That is, all the indispensable view components of V are preserved in V', and the non-replaceable view components are not replaced with information taken from other sources.

P5. If the view-extent parameter is different than "don't care" (i.e., "$\approx$"), then it must be satisfied by V'. I.e., the relationship between the view extents of V' and V is imposed by $\mathcal{VE}$'s value. If V' and V have different view interfaces, i.e., the new view definition V' preserves a subset of the attributes of V, we compare the projections on the common set of attributes in both views. To state it more formally, let $Attr(V')$ and $Attr(V)$ be the interfaces of V' and V, respectively, and the relationship between V' and V be defined by Equation 17.

[5] We do not go into depth on how the MKB changes in this paper.

$$\pi_{Attr(V) \cap Attr(V')}(V') \phi \pi_{Attr(V) \cap Attr(V')}(V) \tag{17}$$

The view-extent parameter $\mathcal{VE} = \delta$ is satisfied, if the following relationship between δ and ϕ holds:

$$\begin{array}{ll} \textit{if view-extent parameter } \mathcal{VE} \textit{ is } \text{``}\equiv\text{''}, & \textit{then } \phi \textit{ must be } \text{``}\equiv\text{''}; \\ \textit{if view-extent parameter } \mathcal{VE} \textit{ is } \text{``}\subseteq\text{''}, & \textit{then } \phi \textit{ must be } \text{``}\subseteq\text{'' or ``}\equiv\text{''}; \textit{ and} \\ \textit{if view-extent parameter } \mathcal{VE} \textit{ is } \text{``}\supseteq\text{''}, & \textit{then } \phi \textit{ must be } \text{``}\supseteq\text{'' or ``}\equiv\text{''}. \end{array} \tag{18}$$

P6. If a view component of V is preserved in the view rewriting V', then the evolution parameters attached to it remain the same as those of the original view component. For new view components of V', the evolution parameters are set according to the assignment rules defined in Definition 4.

6.2 Replacement Strategies

In this section, we give formal descriptions of what are considered to be legal replacements for affected view components under a capability change. Any replacement strategy that follows these guidelines can then be proven to be consistent with the evolution semantics of E-SQL views as defined in Section 4. The proposed substitution guidelines represent the foundation based on which we will validate that the *EVE* approach can indeed achieve view preservation in many situations where conventional view management systems would have to declare the affected views to be undefined.

6.2.1 Principles of Attribute Substitution

When an attribute $R.A$ referred in the view V (in the SELECT or WHERE clauses) is deleted from its site, the view synchronizer attempts to find a substitute to replace the deleted attribute, if replacing $R.A$ is permitted. An attribute $S.B$ is said to be an *appropriate substitute* for $R.A$ if the following conditions are satisfied[6].

Condition 1: Type Match Condition. This condition requires that $S.B$ has the same domain type as the attribute $R.A$. I.e., there exist in MKB the following constraints for some type $Type_1$:
(1) $\mathcal{TC}(S.B) = (S(B) \subseteq B(Type_1))$ and
(2) $\mathcal{TC}(R.A) = (R(A) \subseteq A(Type_1))$.

Condition 2: Tuple Linkage Condition. This requirement demands that there exists a meaningful join relationship between the relations R and S. I.e., there exists a *join constraint* in MKB between R and S such that the attribute $R.A$ is not used in the join condition:

$$\mathcal{JC}_{R,S} = (C_1(\bar{J_1}) \ AND \cdots AND \ C_m(\bar{J_m})) = \mathcal{C}(\bar{J}) \tag{19}$$

where for all $1 \leq i \leq m$, $C_i(\bar{J_i})$ is a primitive clause, and $A \notin (\bar{J_1} \cup \ldots \cup \bar{J_m})$. We use the expression $\mathcal{C}(\bar{J})$ to denote the conjunction of all primitive clauses in $\mathcal{JC}_{R,S}$ where $\bar{J} = \bar{J_1} \cup \ldots \cup \bar{J_m}$.

Condition 3: Extent Satisfaction Condition. Let the value of the view-extent parameter of the view V be δ. The condition from Equation 20 is *sufficient* to have the view-extent parameter $\mathcal{VE}$ satisfied:

[6] Note that when it is not necessary to explicitly specify the full name of an attribute i.e., $IS_i.R.A$, we use $R.A$.

$$\pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\}} \left(R \bowtie_{\mathcal{C}(\bar{J})} S\right) \ \phi \ \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{R.A\}}(R) \tag{20}$$

where $\mathcal{C}(\bar{J})$ is the join condition defined by Condition 2, and $Attr(V)$ represents all the attributes referred in the SELECT and WHERE clauses of the view V. Note that the projection lists in Equation (20) represent ordered sets with the attribute $R.A$ on the right hand side having the same position as the attribute $S.B$ on the left hand side. The view-extent parameter $\mathcal{VE} = \delta$, if different than "$\approx$", and ϕ must satisfy the conditions imposed in Equation 18.

The following theorem states that Conditions 1, 2 and 3 are *sufficient* to obtain a legal rewriting (e.g., with the view-extent parameter $\mathcal{VE}$ satisfied) by using the attribute $S.B$ for replacing the attribute $R.A$ in a view definition.

Theorem 1 *Let a view V be defined as follows:*

$$\begin{array}{ll} \textit{CREATE VIEW} & V \ \ (\mathcal{VE} = \delta) \ \textit{AS} \\ \textit{SELECT} & R.A, R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\ \textit{FROM} & R, R_1, \ldots, R_n \\ \textit{WHERE} & \mathcal{CV}(\bar{W}) \end{array} \tag{21}$$

where $R.A \notin R.\bar{D}$.

Let S and $S.B$ be a relation and one of its attributes, respectively, that satisfy Conditions 1, 2 and 3. Let the view V' be obtained from V by replacing all occurrences of the attribute $R.A$ in the view V with the attribute $S.B$ and adding the condition $\mathcal{C}(\bar{J})$ from the join constraint $\mathcal{JC}_{R,S}$ defined in Equation 19 to the WHERE clause. V' obtained in this way is shown in Equation 22 (where the new view components are underlined).

$$\begin{array}{ll} \textit{CREATE VIEW} & V' \ \ (\mathcal{VE} = \delta) \ \textit{AS} \\ \textit{SELECT} & \underline{S.B}, R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\ \textit{FROM} & \underline{S}, R, R_1, \ldots, R_n \\ \textit{WHERE} & \underline{\mathcal{CV}'((\bar{W} \setminus \{R.A\}) \cup \{S.B\}) \ AND \ \mathcal{C}(\bar{J})} \end{array} \tag{22}$$

where $\mathcal{CV}'((\bar{W} \setminus \{R.A\}) \cup \{S.B\})$ is the conjunction of primitive clauses in the WHERE clause of the view V defined in Equation 21 where all occurrences of the attribute $R.A$ were replaced by the attribute $S.B$.

Then $V' \ \delta \ V$.

Proof.

Case 1. $\mathcal{VE} = \delta =$ "$\subseteq$" and $\phi \in \{$ "$\equiv$" , "$\subseteq$"$\}$.

We have to prove that for $\phi \in \{$ "$\equiv$", "$\subseteq$" $\}$ in Condition 3, V' is a subset of V. I.e., $V' \subseteq V$.

Let t' be a tuple in the view V', $t' \in V'$. Then there exist some tuples in S, R, $R_1, \ldots, R_n$ that have been used to derive the tuple t' in V'. I.e., the following properties hold:

(1) $\exists t_S \in S$, such that $t'[S.B] = t_S[S.B]$,
(2) $\exists t'_R \in R$, such that $t'[R.\bar{D}] = t'_R[R.\bar{D}]$,
(3) for all $1 \leq i \leq n$, $\exists t_i \in R_i$, such that $t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i]$,
(4) $t_S, t'_R, t_1, \ldots t_n$ derive t' in V',
(5) $\mathcal{CV}'(t_S[S.B], t'_R, t_1, \ldots t_n)$[7] is satisfied,
(6) $\mathcal{C}(t_S, t'_R)$ is satisfied.

Property (6) implies that t_S and t'_R are two tuples of S and R, respectively, that derive a tuple in the left hand relation from Equation 20 of Condition 3. I.e., $\exists g \in \left(\pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\}}(R \bowtie_{\mathcal{C}(\bar{J})} S)\right)$ such that $g = \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\}} \left(t'_R \bowtie_{\mathcal{C}(t_S, t'_R)} t_S\right)$. Then from Condition 3 (with $\phi \in \{$ "$\equiv$", "$\subseteq$" $\}$) we have that there exists $t_R \in R$ such that

(7) $g \left(= \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\}} \left(t'_R \bowtie_{\mathcal{C}(t_S, t'_R)} t_S\right)\right) = \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{R.A\}} t_R$.

We want to show that $t_R \in R$, $t_1 \in R_1, \ldots, t_n \in R_n$ derive a tuple t in V such that $t = t'$.

From (7), we have that
(8) $t_R[R.A] = t_S[S.B]$ and
(9) $t_R[R.\bar{D}] = t'_R[R.\bar{D}]$ (because $R.\bar{D} \subseteq (((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\})$).

Properties (5) and (8) imply
(10) $\mathcal{CV}(t_R, t_1, \ldots, t_n)$ is satisfied.
From (8), (9) and (10) we have that
(11) the tuples $t_R \in R$, $t_1 \in R_1, \ldots, t_n \in R_n$ derive a tuple $t \in V$.
Properties (8), (9) and the fact that $t_1, \ldots, t_n$ derive $t' \in V'$ as well (properties (3) and (4)), imply
(12) $t = t'$.

From (11) and (12) we have that $t' \in V$. Since t' was an arbitrarily chosen tuple of V', we have proven that $V' \subseteq V$.

<u>Case 2. $\mathcal{VE} = \delta =$ "$\supseteq$" and $\phi \in \{$ "$\equiv$" , "$\supseteq$" $\}$.</u>

We have to prove that for $\phi \in \{$ "$\equiv$", "$\supseteq$" $\}$ in Condition 3, V is a subset of V'. I.e., $V' \supseteq V$. Let t be a tuple in V, $t \in V$. Then there exist some tuples in R, $R_1, \ldots, R_n$ that derive t in V. Thus, the following properties are true:
(1) $\exists t_R \in R$, such that $t[R.\bar{D}] = t_R[R.\bar{D}]$, and $t[R.A] = t_R[R.A]$
(2) for all $1 \leq i \leq n$, $\exists t_i \in R_i$, such that $t[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i]$,
(3) $t_R, t_1, \ldots t_n$ derive t in V,
(4) $\mathcal{CV}(t_R, t_1, \ldots t_n)$ is satisfied.

By definition we know that $t_R[Attr(V) \cap Attr(R)] \in \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{R.A\}}(R)$. Then from Condition

[7] Even so the conjunction of primitive clauses $\mathcal{CV}'$ is defined on a subset of attributes (i.e., $((\bar{W} \setminus \{R.A\}) \cup \{S.B\})$) of the tuples $t_S, t'_R, t_1, \ldots t_n$, we use this notation do denote the conjunction $\mathcal{CV}'$ applied to this set of tuples. We stress the fact that the tuple t_S has at most one attribute in $((\bar{W} \setminus \{R.A\}) \cup \{S.B\})$, that is $S.B$.

3 (with $\phi\ \in\ \{$ "$\equiv$", "$\supseteq$" $\}$), there exists $t'_R \in R$ and $t_S \in S$ such that:
(5) $t_R[Attr(V) \cap Attr(R)] = \pi_{((Attr(V) \cap Attr(R)) \setminus \{R.A\}) \cup \{S.B\}}(t'_R \bowtie_{\mathcal{C}(t'_R, t_S)} t_S)$.

Property (5) implies
(6) $t_R[R.A] = t_S[S.B]$,
(7) $t'_R[(Attr(V) \cap Attr(R)) \setminus \{R.A\}] = t_R[(Attr(V) \cap Attr(R)) \setminus \{R.A\}]$,
(8) $\mathcal{C}(t'_R, t_S)$ is satisfied.

Properties (6) and (7) imply that
(9) $\mathcal{CV}'(t_S[S.B], t'_R, t_1, \ldots, t_n)$ is satisfied.

We want to prove that the tuples $t_S, t'_R, t_1, \ldots, t_n$ derive a tuple t' in V', and this tuple is equal to t, i.e., $t' = t$.

Properties (8) and (9) state that the tuples $t_S, t'_R, t_1, \ldots, t_n$ satisfy the two conditions from the WHERE clause of the view V', thus this set of tuples derive a tuple t' in V'. From (1), (2), (3), (6) and (7) we have that t' is equal to t. More precisely,

(10) $t'[S.B] = t_S[S.B] \stackrel{(6)}{=} t_R[R.A] \stackrel{(1)}{=} t[R.A]$,

(11) $t'[R.\bar{D}] = t'_R[R.\bar{D}] \stackrel{(7)}{=} t_R[R.\bar{D}] \stackrel{(1)}{=} t[R.\bar{D}]$,

(12) for all $1 \leq i \leq n$, $t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i] \stackrel{(2)}{=} t[R_i.\bar{D}_i]$.

Hence, we can conclude that $t' = t$. Since, t was chosen arbitrary from V, we have proven that $V' \supseteq V$.

Case 3. $\mathcal{VE}\ =\ \delta\ =$ "$\equiv$" and $\phi\ =$ "$\equiv$".

We want to show that $V' \equiv V$ when $\phi\ =$ "$\equiv$" in Condition 3. Hence, we have to prove two inclusions:
(I) $V' \subseteq V$ and
(II) $V' \supseteq V$.

The inclusion (I) is implied by Case 1 proven above when $\phi\ =$ "$\equiv$"[8] Similarly, the inclusion (II) is implied by Case 2 with $\phi\ =$ "$\equiv$". Thus, we conclude that $V \equiv V'$.

Q.E.D.

The following theorems are special cases when $\mathcal{PC}$s and $\mathcal{JC}s$ constraints specified for the relation R and a relation S in MKB imply Conditions 1, 2 and 3 and make relation S a good replacement. The theorem thus leads us to guidelines when our meta knowledge in the form of $\mathcal{PC}$s and $\mathcal{JC}$s constraints will be sufficient to identify current replacements.

Theorem 2 *Let V be defined as in Equation 21 and $\delta =$ "$\supseteq$". Let S be a relation with the following constraints:*

[8] Cases 1 and 2 are more general cases proven for $\phi\ \in\ \{$ "$\equiv$", "$\subseteq$" $\}$ and $\phi\ \in\ \{$ "$\equiv$", "$\supseteq$" $\}$, respectively.

(I) $\mathcal{JC}_{R,S} = \left(R.\bar{A}_1 = S.\bar{B}_1\right)$ *with* $A \notin \bar{A}_1$*;*

(II) $\mathcal{PC}_{R,S} = (\pi_{R.\bar{A}}(R) \subseteq \pi_{S.\bar{B}}(S))$ *with*

(1) $R.A \in R.\bar{A}$, $R.\bar{A}_1 \subseteq R.\bar{A}$, $Attr(V) \cap Attr(R) \subseteq R.\bar{A}$*; and*

(2) $S.B \in S.\bar{B}$, $S.\bar{B}_1 \subseteq S.\bar{B}$*;*

(3) $R.A$, $R.\bar{A}_1$ *and* $S.B$, $S.\bar{B}_1$ *have the same position in the attribute vectors* $R.\bar{A}$ *and* $S.\bar{B}$*, respectively.*

Then Conditions 1, 2 and 3 are satisfied for ϕ = "$\supseteq$" *for the relation* S *and the attribute* $S.B$*.*

Theorem 3 *Let* V *be defined as in Equation 21 and* δ = "$\subseteq$"*. Let* S *be a relation with the following constraints:*

(I) $\mathcal{JC}_{R,S} = \left(R.\bar{A}_1 = S.\bar{B}_1 \ AND \ \mathcal{C}(\bar{J})\right)$ *with* $A \notin (\bar{A}_1 \cup \bar{J})$ *and* $\mathcal{C}(\bar{J})$ *a conjunction of local*[9] *primitive clauses.*

(II) $\mathcal{PC}_{R,S} = (\pi_{R.\bar{A}}(R) \supseteq \pi_{S.\bar{B}}(S))$ *with*

(1) $R.A \in R.\bar{A}$, $R.\bar{A}_1 \subseteq R.\bar{A}$, $Attr(V) \cap Attr(R) \subseteq R.\bar{A}$*; and*

(2) $S.B \in S.\bar{B}$, $S.\bar{B}_1 \subseteq S.\bar{B}$*;*

(3) $R.A$, $R.\bar{A}_1$ *and* $S.B$, $S.\bar{B}_1$ *have the same position in the attribute vectors* $R.\bar{A}$ *and* $S.\bar{B}$*, respectively.*

Then Conditions 1, 2 and 3 are satisfied for ϕ = "$\subseteq$" *for the relation* S *and the attribute* $S.B$*.*

6.2.2 Principles of Relation Substitution

When a relation $IS_1.R$ referred in the FROM clause of a view V is deleted from its site, the view synchronizer will under certain conditions, e.g., checking the relevant evolution parameters to see whether the view V can be evolved, attempt to find a substitution for it. A relation $IS_2.S$ is said to be an *appropriate substitute* for $IS_1.R$ if the following three conditions are satisfied.

Condition 1: Type Match Condition. All attributes of relation S that are used as replacements for attributes of relation R must have the same domain type, respectively, i.e., there exist type constraints: $\mathcal{TC}(A) = (\ R(A) \ \subseteq A(Type)\ )$ and $\mathcal{TC}(B) = (\ S(B) \ \subseteq \ B(Type)\ )$ in the MKB for all attribute pairs $(R.A, S.B)$ used for substitution.

Condition 2: Minimal Preservation Condition. This condition requires that relation S must contain *at least* the corresponding attributes of the relation R that are indispensable and replaceable in the view V. That is, all the attributes of R in the SELECT clause with $\mathcal{AD} = false$ and $\mathcal{AR} = true$ and all the attributes of the relation R that appear in the WHERE clause in a condition C with $\mathcal{CD} = false$ and $\mathcal{CR} = true$. This is formally stated below.

We use the notation $Attr(V(R))_{\mathsf{SELECT}}\ (d, r)$ to denote all the attributes of the relation R from the SELECT clause with the evolution parameters set to d and r (d and r can be $false$ or $true$), respectively:

$$Attr(V(R))_{\mathsf{SELECT}}\ (d, r) = \{R.A \mid R.A \text{ in SELECT clause}, \mathcal{AD}(R.A) = d, \mathcal{AR}(R.A) = r\} \tag{23}$$

[9] A local primitive clause is a predicate having only one attribute (e.g., $R.C > 20$).

And, we use $Attr(V(R))_{\mathsf{WHERE}}\,(d, r)$, for the set of all the attributes of relation R used in primitive clauses of the WHERE clause which have the evolution parameters set to d and r, respectively:

$$Attr(V(R))_{\mathsf{WHERE}}\,(d, r) = \{R.A \mid R.A \text{ in a condition } C \text{ from } \mathsf{WHERE} \text{ clause}, \mathcal{CD}(C) = d, \mathcal{CR}(C) = r\} \tag{24}$$

With the notations defined above, we can formally state the minimal preservation condition as:

Case 1. $\mathcal{VE}$ = "$\subseteq$" or "$\equiv$"

$$Attr(V(R))_{\mathsf{SELECT}}\,(false, true) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(false, false) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(false, true) \cup \\ \cup\, Attr(V(R))_{\mathsf{WHERE}}\,(true, false) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(true, true) \;\subseteq\; S. \tag{25}$$

Case 2. $\mathcal{VE}$ = "$\supseteq$"

$$Attr(V(R))_{\mathsf{SELECT}}\,(false, true) \cup Attr(V(R))_{\mathsf{WHERE}}\,(false, true) \;\subseteq\; S. \tag{26}$$

In short, the minimal preservation constraint states that all attributes of R that are essential for the view (i.e., the indispensable attributes) and replaceable (i.e., their attribute-replaceable evolution parameter values are set to $true$) must be obtained from S. Moreover, if the view-extent evolution parameter is "$\subseteq$", then all attributes of R used in the WHERE clause must have replacements in S (we cannot drop a condition from the WHERE clause and still have the view-extent evolution parameter satisfied). Clearly, this is a necessary (but not sufficient) condition in order for the relation R to be replaced by S.

Condition 3: Extent Satisfaction Condition. Let the value of the view-extent parameter of the view V be δ. The following condition is *sufficient* to have the view-extent parameter $\mathcal{VE}$ satisfied:

$$\pi_{\bar{B}}(S) \;\phi\; \pi_{\bar{A}}(R) \tag{27}$$

where $\bar{A}$ must be a superset of the attributes covered by S (i.e., attributes mentioned in the minimal preservation condition) and $\bar{B}$ refers to the attributes in S that are used as replacements for attributes $R.\bar{A}$. Thus, the following conditions must hold:

Case 1. $\mathcal{VE}$ = "$\subseteq$" or "$\equiv$"

$$Attr(V(R))_{\mathsf{SELECT}}\,(false, true) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(false, false) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(false, true) \cup \\ \cup\, Attr(V(R))_{\mathsf{WHERE}}\,(true, false) \;\cup\; Attr(V(R))_{\mathsf{WHERE}}\,(true, true) \;\subseteq\; \bar{A}. \tag{28}$$

Case 2. $\mathcal{VE}$ = "$\supseteq$"

$$Attr(V(R))_{\mathsf{SELECT}}\,(false, true) \cup Attr(V(R))_{\mathsf{WHERE}}\,(false, true) \;\subseteq\; \bar{A}. \tag{29}$$

If the value of the view-extent parameter is different than "don't care", i.e., $\delta \neq \approx$, then the values of δ and ϕ must satisfy the property from Equation 18.

The above conditions are *sufficient* but not *necessary*. The following theorem states that Conditions 1, 2 and 3 are *sufficient* to have the view-extent evolution parameter $\mathcal{VE}$ satisfied when S is used to replace the relation R.

Theorem 4 *Let a view V be defined as follows:*

$$
\begin{array}{ll}
\mathsf{CREATE\ \ VIEW} & V \ \ (\mathcal{VE} = \delta) \ \mathsf{AS} \\
\mathsf{SELECT} & R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\
\mathsf{FROM} & R, R_1, \ldots, R_n \\
\mathsf{WHERE} & \mathcal{CV}(\bar{W})
\end{array}
\tag{30}
$$

where all attributes of R in $\bar{W}$ are denoted by $R.\bar{D}'$.

Let S be a relation that satisfies Conditions 1, 2 and 3. Let the view V' be obtained from V by replacing R with S and replacing all the attributes of R with the corresponding attributes of S. V' obtained in this way is shown in Equation 31 (where the new view components are underlined).

$$
\begin{array}{ll}
\mathsf{CREATE\ \ VIEW} & V' \ \ (\mathcal{VE} = \delta) \ \mathsf{AS} \\
\mathsf{SELECT} & \underline{S.\bar{F}}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\
\mathsf{FROM} & \underline{S}, R_1, \ldots, R_n \\
\mathsf{WHERE} & \underline{\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')}
\end{array}
\tag{31}
$$

In Equation 31, $S.\bar{F}$ are the attributes from $S.\bar{B}$ corresponding to the attributes from $R.\bar{A} \cap R.\bar{D}$. From Conditions 2 and 3, we have that $S.\bar{F}$ corresponds to a superset of the attributes in $Attr(V(R))_{\mathsf{SELECT}}(false, true)$.
$S.\bar{F}'$ are the attributes from $S.\bar{B}$ corresponding to the attributes from $R.\bar{A} \cap R.\bar{D}'$. From Conditions 2 and 3, we have that this must be the set of all the attributes of R from the WHERE *clause in Case 1; and, in Case 2, it contains at least the attributes from $Attr(V(R))_{\mathsf{WHERE}}(false, true)$. $\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')$ is the conjunction of primitive clauses in the* WHERE *clause of the view V defined in Equation 30 where all occurrences of the attributes $R.\bar{D}'$ were replaced by the corresponding attributes in $S.\bar{F}'$ or the conditions containing attributes from $R.\bar{D}'$ were dropped (if it is legal to do so).*

Then $V' \ \delta \ V$.

Proof.

<u>Case 1. $\mathcal{VE} = \delta =$ "$\subseteq$" and $\phi \in \{$ "$\equiv$" , "$\subseteq$"$\}$.</u>

We have to prove that for $\phi \in \{$ "$\equiv$", "$\subseteq$" $\}$ in Condition 3, V' is a subset of V (for common attributes). I.e., $V' \subseteq_\pi V$ [10]. For this particular case we have to impose that all attributes of R that appear in the WHERE clause are replaced by attributes of S. That is
(0) $\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')$ has the same set of primitive clauses as $\mathcal{CV}(\bar{W})$[11]. And, $|R.\bar{D}'| = |S.\bar{F}'|$.

Let t' be a tuple in the view V', $t' \in V'$. Then there exists some tuples in $S, R_1, \ldots, R_n$ that derive the tuple t' in V'. I.e., the following properties hold:

[10] $\subseteq_\pi$ is defined in [].
[11] If some of the clauses in the WHERE clause are dropped in V' then in general we cannot prove $V' \subseteq_\pi V$.

(1) $\exists t_S \in S$, such that $t'[S.\bar{F}] = t_S[S.\bar{F}]$,
(2) for all $1 \leq i \leq n$, $\exists t_i \in R_i$, such that $t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i]$,
(3) $t_S, t_1, \ldots t_n$ derive t' in V',
(4) $\mathcal{CV}'(t_S[S.\bar{F}'], t_1, \ldots t_n)$ is satisfied.

From Condition 3 we have that the attributes of S used in the new view definition are among the ones used in the Equation 27. I.e.,
(5) $S.\bar{F}, S.\bar{F}' \subseteq S.\bar{B}$.
Then from Equation 27 (with $\phi \in \{$ "$\equiv$", "$\subseteq$" $\}$) we have that there exists a tuple $t_R \in R$ such that
(6) $t_S[S.\bar{B}] = t_R[R.\bar{A}]$.

We want to show that $t_R \in R$, $t_1 \in R_1, \ldots, t_n \in R_n$ derive a tuple t in V such that $t =_\pi t'$.

From (0), (5) and (6) we have that
(7) $t_S[S.\bar{F}'] = t_R[R.\bar{D}']$[12] and $t_S[S.\bar{F}] =_\pi t_R[R.\bar{D}]$ where $S.\bar{F} \cup S.\bar{F}'$ are all the attributes of S that replace attributes of R (they must include at least the indispensable and replaceable attributes of R described in the Condition 2).

From (0),(4) and (7) we have
(8) $\mathcal{CV}(t_R[R.\bar{D}'], t_1, \ldots t_n)$ is satisfied[13].

Then from (8) we can deduce that
(9) the tuples $t_R \in R$, $t_1 \in R_1, \ldots, t_n \in R_n$ derive a tuple t in V.

Now let's prove that $t =_\pi t'$. From (2) and (9) we have that

(10) for all $1 \leq i \leq n$, $(t_i \in R_i)$, $t'[R_i.\bar{D}_i] \overset{(2)}{=} t_i[R_i.\bar{D}_i] \overset{(9)}{=} t[R_i.\bar{D}_i]$.

From (1) and (7) we have that

(11) $t'[S.\bar{F}] \overset{(1)}{=} t_S[S.\bar{F}] \overset{(7)}{=}_\pi t_R[R.\bar{D}] \overset{(9)}{=} t[R.\bar{D}]$.

In (10) and (11) we have proven that $t =_\pi t'$. Since, t' was an arbitrary chosen tuple of V', we have proven that $V' \subseteq_\pi V$.

Case 2. $\mathcal{VE} = \delta =$ "$\supseteq$" and $\phi \in \{$ "$\equiv$" , "$\supseteq$" $\}$.

We have to prove that for $\phi \in \{$ "$\equiv$", "$\supseteq$" $\}$ in Condition 3, V is a subset of V'. I.e., $V' \supseteq_\pi V$.

Let t be a tuple in V, $t \in V$. Then there exist some tuples in R, $R_1, \ldots, R_n$ that derive t in V. Thus, the following properties are true:
(1) $\exists t_R \in R$, such that $t[R.\bar{D}] = t_R[R.\bar{D}]$,
(2) for all $1 \leq i \leq n$, $\exists t_i \in R_i$, such that $t[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i]$,
(3) $t_R, t_1, \ldots t_n$ derive t in V,

[12] From (0) we have that $|S.\bar{F}'| = |R.\bar{D}'|$.
[13] HERE IS THE PLACE WHERE WITHOUT (0) we cannot prove $V' \subseteq V$.

(4) $\mathcal{CV}(t_R, t_1, \ldots t_n)$ is satisfied.

From Condition 3 (with $\phi \in \{$ "$\equiv$", "$\supseteq$" $\}$), there exists $t_S \in S$ such that
(5) $t_R[R.\bar{A}] = t_S[S.\bar{B}]$.

Property (5) implies
(6) $t_S[S.\bar{F'}] =_\pi t_R[R.\bar{D'}]$,
(7) $t_S[S.\bar{F}] =_\pi t_R[R.\bar{D}]$.

We want to prove that the tuples $t_S, t_1, \ldots, t_n$ derive a tuple t' in V', and this tuple is equal to t, i.e., $t' =_\pi t$.

Property (6) implies that
(8) $\mathcal{CV}'(t_S[S.\bar{F'}], t_1, \ldots, t_n)$[14] is satisfied.

Properties (8) states that the tuples $t_S, t_1, \ldots, t_n$ satisfy the condition from the WHERE clause of the view V', thus this set of tuples derive a tuple t' in V'. From (1), (2), (3), (6) and (7) we have that t' is equal to t. More precisely,

(9) $t'[S.\bar{F}] = t_S[S.\bar{F}] \overset{(7)}{=_\pi} t_R[R.\bar{D}] \overset{(1)}{=} t[R.\bar{D}]$,

(10) for all $1 \leq i \leq n$, $t'[R_i.\bar{D_i}] = t_i[R_i.\bar{D_i}] \overset{(2)}{=} t[R_i.\bar{D_i}]$.

Hence, we can conclude that $t' =_\pi t$. Since, t was chosen arbitrary from V, we have proven that $V' \supseteq_\pi V$.

<u>Case 3. $\mathcal{VE} = \delta =$ "$\equiv$" and $\phi =$ "$\equiv$".</u>

We want to show that $V' \equiv_\pi V$ when $\phi =$ "$\equiv$" in Condition 3. Hence, we have to prove two inclusions:
(I) $V' \subseteq_\pi V$ and
(II) $V' \supseteq_\pi V$.

The inclusion (I) is implied by Case 1 proven above when $\phi =$ "$\equiv$" with the restriction imposed in (0). Similarly, the inclusion (II) is implied by Case 2 with $\phi =$ "$\equiv$". Then, we conclude that $V \equiv_\pi V'$ when the restriction imposed in Case 1 at (0) is satisfied.

Q.E.D.

7 View Synchronization Algorithms

In this section, we present the view synchronization algorithms which serve as proof of concept that adaptability of views can indeed be achieved within our proposed *EVE* framework. For the remainder, we make the following simplifying assumptions:

[14] Note that this case can be proven in general when $\mathcal{CV}'$ is obtained from $\mathcal{CV}$ by dropping some of the conditions and replacing the attributes of R.

- A relation R appears in the FROM clause only once.
- At least one attribute of R is referenced in the SELECT and/or WHERE clause, i.e., no redundant relations are listed in the FROM clause.
- We consider precisely-defined view queries only and not loosely-specified ones as studied in [NR97].

We believe our solution approach could be easily adapted for a more general case when the assumptions are relaxed. The capability changes supported in EVE and thus treated below are listed next:

1. del-attr(IS.R.A): delete the attribute A from the relation R residing at site IS.
2. add-attr(IS.R.A): add an attribute A to the relation R at site IS.
3. chg-attr-name(IS.R.A,B): change an attribute's name from A to B in the relation R at site IS.
4. del-rel(IS.R): delete the relation R from the site IS.
5. add-rel(IS.R): add a relation R to the site IS.
6. chg-rel-name(IS.R,S): change the relation's name from R to S at site IS.

7.1 The Delete-Attribute Evolution Operator–del-attr($IS_1.R.A$)

Deleting the attribute A from $IS_1.R$ could potentially affect a view V in three ways:

1. A appears in the SELECT clause of V only.
2. A appears in the WHERE clause of V only.
3. A appears in both the SELECT and WHERE clauses of V (i.e., a combination of cases 1 and 2).

Below, we now provide solutions to each of these three cases one by one.

Case 1: A appears in the SELECT clause of V only.

When an attribute is deleted from the SELECT clause, the view synchronizer decides whether V is amendable by taking the attribute's *attribute-dispensable* $\mathcal{AD}$ and *attribute-replaceable* $\mathcal{AR}$ parameters, and the *view-extent* $\mathcal{VE}$ parameter into account to decide whether the affected view can be evolved into a valid view definition. The view evolution algorithm (VEA) for this case is listed below.

Algorithm 1 ***VEA-delete-attribute(A,SELECT):***

```
Success = TRUE
IF attribute-replaceable(A) = FALSE
THEN IF attribute-dispensable(A) = TRUE
     THEN drop A from V                    /* report success */
     ELSE /* attribute-dispensable(A) = FALSE */
```

```
            Success = FALSE                   /* report failure */
       END IF
ELSE /* attribute-replaceable(A) = TRUE */
  IF attribute-dispensable(A) = TRUE
  THEN find-substitute-select(A, B)         /* see Section 6.2.1 */
       IF   found
       THEN replace-attribute(A,B)          /* report success */
       ELSE /* not found */
           drop A from V                    /* report success */
       END IF
   ELSE /* attribute-dispensable(A) = false */
        find-substitute-select(A, B)        /* see Section 6.2.1 */
        IF found
        THEN replace-attribute(A,B)         /* report success */
        ELSE /* not found */
            Success = FALSE                 /* report failure */
        END IF
  END IF
END IF
```

Algorithm 2 ***PROCEDURE replace-attribute(R.A,S.B):***

```
begin
1. drop A from the SELECT clause
2. add the relation S, that B belongs to, to the FROM clause along with R.
3. add the join constraint between R and S to the WHERE clause (Section 6.2.1).
4. add B to the SELECT clause
end
```

Algorithm 3 ***Boolean PROCEDURE find-substitute-select(in: R.A, out: S.B):***

```
begin
    the strategy of appropriate attribute substitution is outlined in Section 6.2.1.
end
```

Next, we use an example to show how the view synchronization algorithm finds a legal rewriting for a view affected by a *delete-attribute* capability change.

Example 16 *For easy reference, we re-display Query (5) first introduced in Section 4.8 over the ISs as defined in Section 3.*

$$\begin{array}{ll} \textsf{\textit{CREATE VIEW}} & \textit{Asia-Customer } (\mathcal{VE} = \text{``}\supseteq\text{''}) \textsf{\textit{ AS}} \\ \textsf{\textit{SELECT}} & \textit{Name, Address, Phone } (\mathcal{AD} = true,\ \mathcal{AR} = true) \\ \textsf{\textit{FROM}} & \textit{Customer C } (\mathcal{RR} = true), \textit{FlightRes F} \\ \textsf{\textit{WHERE}} & (C.Name = F.PName)\ AND (F.Dest = \text{'}Asia\text{'})\ (\mathcal{CD} = true) \end{array} \tag{32}$$

We assume that the travel agency has the Customer relation backed up at the Boston branch to guarantee availability and reliability of the information service. That is, our MKB holds the $\mathcal{PC}$ constraint (CustomerBak $\equiv$ Customer) and the join constraint ($\mathcal{JC}_{CustomerBak,Customer}$ = (CustomerBak.Name = Customer.Name)).

Assume the Phone attribute is deleted from the Customer relation at the headquarter. Upon receiving this del-attr(Customer.Phone) *notification, the view synchronizer checks with the MKB in order to find an "appropriate" counterpart of it (based on the process in Section 6.2.1). In this case,* CustomerBak.Phone *is found to be a promising candidate. In this example, steps 16 - 19 of the View Evolution Algorithm VEA-delete-attribute (algorithm 1) are executed. Using this algorithm, one valid strategy of rewriting Asia – Customer into Asia – Customer′ thus results into Equation (33) (new components are underlined):*

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & \textit{Asia-Customer'}\ (\mathcal{VE} = \text{``}\supseteq\text{''})\ \textsf{AS} \\
\textsf{SELECT} & \textit{Name, Address,} \underline{\textit{C2.Phone}(\mathcal{AD} = \textit{true}, \mathcal{AR} = \textit{true})} \\
\textsf{FROM} & \textit{Customer C}\ (\mathcal{RR} = \textit{true}), \textit{FlightRes F}, \\
& \underline{\textit{CustomerBak C2}\ (\mathcal{RD} = \textit{true}, \mathcal{RR} = \textit{true})} \\
\textsf{WHERE} & (\textit{C.Name} = \textit{F.PName})\ \textit{AND}(\textit{F.Dest} = \textit{'Asia'})\ (\mathcal{CD} = \textit{true})\ \textit{AND} \\
& \underline{(\textit{C2.Name} = \textit{C.Name})(\mathcal{CD} = \textit{true}, \mathcal{CR} = \textit{true})}
\end{array}
\tag{33}
$$

This legal rewriting uses the join constraint $\mathcal{JC}_{CustomerBak,Customer}$ to obtain the phone number from the relation CustomerBak.

Note that there may be several alternative solutions for salvaging a view. For example, if the Name and Address attributes in Query 32 are allowed to be taken from other sources, then the Customer relation could be replaced entirely by the CustomerBak relation – even if only the attribute Phone is deleted from the Customer relation but not the entire Customer relation. The main advantage of the latter rewriting is that the join operation between the relations Customer and CustomerBak can be avoided entirely, which should reduce the view computation and view maintenance costs. Our current view synchronizer starts with the simplest strategy of view rewriting and progressively explores alternative more complex view synchronization solutions until one is found that is valid given the view evolution constraints as well as the constraints in the MKB. Hence, while our current view synchronizer will find one solution for view evolution if one exists based on our chosen set of view synchronization algorithms, it is not guaranteed to select the "best" one. In the future, we will explore optimization strategies that address the issue of selecting the "best" solution for view evolution given cost criteria, such as costs of accessing ISs, availability and contracts with ISs, communication costs, view self-maintainability, etc.

Case 2: A appears in the WHERE clause of V only.

When a condition in the WHERE clause is affected because one of its operands A is deleted from its IS, our system takes the *condition-dispensable* $\mathcal{CD}$, *condition-replaceable* $\mathcal{CR}$, and *view-extent* $\mathcal{VE}$ parameters into account to decide whether the affected view is amendable. If it is amendable, then the view synchronizer tries to remedy it. The view evolution algorithm that handles cases when one or more WHERE conditions of a view V, denoted by c = ($R.A\ \theta\ operand_2$), are affected by the removal of the attribute A is given next.

Algorithm 4 ***VEA-delete-attribute(A,WHERE):***

```
01. C = {affected conditions}
```

```
Success = TRUE
WHILE (C != empty) AND (Success) DO
   take c from C
   IF condition-replaceable(c) = FALSE
      IF condition-dispensable(c) = TRUE
      THEN
           C = C - c;  drop c from V;
      ELSE /* condition-dispensable(c) = FALSE */
          Success = FALSE
      END IF
   ELSE /* condition-replaceable(c) = TRUE */
       IF condition-dispensable(c) = TRUE
       THEN find-substitute-condition(c, c1) /* see Section 6.2.1 */
            IF found
            THEN replace-condition(c,c1)
            ELSE /* not found */
                drop c from V
            END IF
            C = C - c
       ELSE /* condition-dispensable(c) = FALSE */
            find-substitute-condition(c, c1) /* see Section 6.2.1 */
            IF found
            THEN replace-condition(c,c1)
                 C = C - c
            ELSE /* not found */
                 Success = FALSE
            END IF
       END IF
   END IF
END DO
```

Algorithm 5 *Boolean PROCEDURE find-substitute-condition(C,C'):*

```
begin
   //  Section 6.2.1 describes how substitution C' for C  is found
   //  by finding replacements for its attributes.
end
```

Algorithm 6 *PROCEDURE replace-condition(C,C'):*
// $C = (R.A\ \theta\ operand_2)$
// $C' = (S.B\ \theta\ operand_2)$

```
1. drop C from the WHERE clause
2. add the relation S, that B belongs to, to the FROM clause
3. add the join constraint between R and S to the WHERE clause
4. add C' to the WHERE clause
```

Example 17 *Let's assume a view is specified on* $R_1(A_1, A_2), R_2(B_1, B_2)$*, and* $R_3(C_1, C_2)$ *as follows:*

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & V\ (\mathcal{VE} = \text{"}\supseteq\text{"})\ \textsf{AS} \\
\textsf{SELECT} & A_2, B_1, B_2, C_2 \\
\textsf{FROM} & R_1, R_2, R_3 \\
\textsf{WHERE} & (A_1 = B_1)\ (\mathcal{CD} = true, \mathcal{CR} = true)\ AND\ (A_1 = C_1)\ (\mathcal{CD} = true, \mathcal{CR} = true)
\end{array}
\tag{34}
$$

Figure 7(a) shows a valid database state of $R_1(A_1, A_2), R_2(B_1, B_2), R_3(C_1, C_2)$, and Figure 7(b) the view extent of V derived from R_1, R_2, and R_3 (with one tuple). In the view definition V, $R_1(A_1, A_2)$, $R_2(B_1, B_2)$, and $R_3(C_1, C_2)$ are related to each other through the join conditions: $(A_1 = B_1)$ and $(A_1 = C_1)$ (see Figure 7.c).

R1

A1	A2
2	3
1	6

R2

B1	B2
7	2
2	6

R3

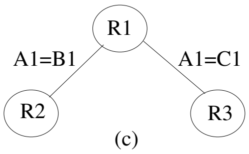
C1	C2
9	3
2	8

(a)

V

A2	B1	B2	C2
3	2	6	8

(b)

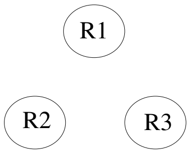


(c)

Figure 7: Example Data Set.

V'

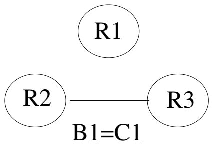
A2	B1	B2	C2
3	7	2	3
3	7	2	8
3	2	6	3
3	2	6	8
6	7	2	3
6	7	2	8
6	2	6	3
6	2	6	8



(a) Redefined View: V'.

V''

A2	B1	B2	C2
3	2	6	8
6	2	6	8



(b) Redefined View: V''.

Figure 8: Two Alternative Ways to Evolve V.

Let's assume that the information provider of R decides to delete $R.A_1$. Obviously, both of the primitive clauses in the WHERE *clause of the view definition V are affected. When EVE fails to find appropriate replacements for these conditions, both primitive clauses are dropped since their condition-dispensable ($\mathcal{CD}$) parameters are set to true. Hence, V is rewritten into V' as follows:*

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & V'\ (\mathcal{VE} = \text{"}\supseteq\text{"})\ \textsf{AS} \\
\textsf{SELECT} & A_2, B_1, B_2, C_2 \\
\textsf{FROM} & R_1, R_2, R_3
\end{array}
\tag{35}
$$

That is, the original view definition V becomes a Cartesian product in V', because the new view definition V' has an empty WHERE *clause and the relations have no common attribute names, hence, no natural join takes place.*

In the redefined view definition $V^{'}$, R_1, R_2, and R_3 are no longer related to each other through any join conditions. As a consequence, the view extent now contains 8 instead of 1 tuples (see Figure 8(a)).

When a condition from the WHERE clause has to be dropped (as in the above example), more sophisticated techniques could be used to evolve the view in order to preserve the original view to a larger degree. The basic idea is to make inferences based on the implicit constraints hidden in the conditions of the original WHERE clause to help our system preserve the original view. While there are several potential solution approaches, we propose below one such technique that improves upon the algorithm described above.

Algorithm 7 ***PROCEDURE** replace-condition*(C,C'):*

1. *Find any implicit constraints in the WHERE clause by computing the transitive closure of the conditions;*
2. *Add these implicit constraints to the WHERE clause;*
3. *Remove the affected conditions from the WHERE clause.*

To be more precise, let's consider a view definition V with a conjunction $\mathcal{C}$ of primitive clauses in the WHERE clause and attribute A appearing only in the WHERE clause. Let $\mathcal{C}'$ be the conjunction of all the primitive clauses in $\mathcal{C}$ which don't use the attribute A (i.e., $\mathcal{C}'$ is obtained from $\mathcal{C}$ by dropping the primitive clauses that contain A). Let $\mathcal{C}''$ be obtained from $\mathcal{C}$ by finding first the transitive closure of $\mathcal{C}$ and then removing the primitive clauses that contain attribute A (see Step 1 to Step 3 from above). Let V' be obtained from V by replacing the conjunction $\mathcal{C}$ with $\mathcal{C}'$ in the WHERE clause; and V'' be obtained from V by replacing the conjunction $\mathcal{C}$ with $\mathcal{C}''$ in the WHERE clause. Then, we have that $V \subseteq V'' \subseteq V'$ for any database instance. The proof of this statement follows immediately from the theorem of containment for conjunctive queries with built-in predicate given by Ullman in [Ull89].

Example 18 *Continuing with the above example, our system finds an implicit constraint in the WHERE clause between R_2 and R_3, namely, $R_2.B_1 = R_3.C_1$, derived from $R_2.B_1 = R_1.A_1$ and $R_1.A_1 = R_3.C_1$ by transitivity. We add this constraint into the WHERE clause. After removing the conditions containing A_1, the WHERE clause is left with one join condition: $B_1 = C_1$. As shown in Figure 8.b, R_2 is joined with R_3 in the modified view $V^{''}$ through the join condition $B_1 = C_1$, but R_2 and R_3 are not joined with R_1 any longer (hence, the Cartesian product is used to combine these two relations in the modified view). The evolved view definition $V^{''}$ is given below:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V^{''}\ (\mathcal{VE} = \text{“}\supseteq\text{”})\ \textsf{AS} \\ \textsf{SELECT} & A_2, B_1, B_2, C_2 \\ \textsf{FROM} & R_1, R_2, R_3 \\ \textsf{WHERE} & (B_1 = C_1)\ (\mathcal{CD} = true, \mathcal{CR} = true) \end{array} \tag{36}$$

In this case, our system is able to preserve the original view "to a larger degree" in the sense of only generating one superfluous tuple compared to the original view extent. (See Figure 8(a) versus 8(b)). While in $V^{'}$, all the information of R_1, R_2, and R_3 is dumped to the user, $V^{''}$ comes close to providing to the user only what he requested to begin with. It is not only less meaningful, but also more expensive to ship such extra unneeded data.

Case 3: A appears in both the SELECT and WHERE clauses of V.

The main idea is to (1) go through the affected view components of V once to decide the possibility of view evolution, and (2) if V has the potential to be evolved, then find a substitute for the affected SELECT component and, if no failure happens when replacing/dropping the SELECT component, replace the WHERE components by the corresponding substitute, as needed.

Algorithm 8 ***VEA-delete-attribute(A,ALL):***

```
AC1 = affected-components(A)  /* find components that reference A in V */
Success = TRUE
WHILE (AC1 != empty) AND (Success) DO
   get component from AC1
   IF ( dispensable(component) = FALSE AND replaceable(component) = FALSE )
   THEN Success = FALSE
   END IF
   AC1 = AC1 - component
END DO
IF (Success)           /* it is possible to evolve V */
THEN call VEA-delete-attribute(A,SELECT);
    IF (Success)
    THEN               /* use substitute for SELECT component, if found */
         call VEA-delete-attribute'(A,WHERE);
    END IF
END IF
```

VEA-delete-attribute'(A,WHERE) is identical to VEA-delete-attribute(A,WHERE) procedure introduced earlier, except that now if a replacement of A by $A^{'}$ had been found by the successful execution of the VEA-Delete-Attribute(A,SELECT) procedure earlier, then use $A^{'}$ in place of A in the WHERE clause without taking any further replacement steps.

7.2 The Add-Attribute Evolution Operator

This *add-attr(IS.R.A)* operator reports that a new attribute A has been added to the relation R at site IS. We assume EVE does not attempt to further optimize existing views using the newly added attribute, so this capability change does not affect any of the existing views in our current system.

7.3 The Change-Attribute-Name Evolution Operator

This *chg-attr-name(IS.R.A,B)* operator changes the name of an attribute A of $IS.R$ to a new name B. This operation does not affect the view definitions that refer to $R.A$, assuming our system keeps a name-mapping table in the MKB along with other meta knowledge. Even if a name changes more than once, our system could keep track of this information in the same entry of the name mapping table. The alternate solution of identifying all locations where the old name of the attribute was being used both in the MKB and in the VKB and replacing the old name by the new name is also straightforward, yet potentially expensive.

7.4 The Delete-Relation Evolution Operator

The delete-relation operator removes a relation R from its IS, and it affects views that reference R in their FROM clauses. Since (1) several attributes of the deleted relation R may be referenced in a view definition, and (2) it is generally more expensive to find an appropriate replacement for an affected view component that references an attribute of R than to check the possibility of view evolution, we propose to handle the view synchronization problem in two steps. First, we evaluate the possibility of view evolution by examining the view evolving parameters of each of the affected view components in V. Basically, if there is an affected view component whose evolving parameters are *(dispensable(component)* $=$ *false,* and *replaceable(component)* $=$ *false)* then it is impossible to evolve the view definition. As soon as we decide that evolving a component of V is impossible (given its evolving parameters), our system will report failure without looking further.

Otherwise, the second stage is to find appropriate replacements for the affected view components using a simple (one-step) solution shown below.

Algorithm 9 ***VEA-delete-relation(R)***

```
tempSet = affected-components(VD,R) /* view components referring to R or attrs(R) */
code = 2 /* code = 1, must find replacement; 2, good if finds replacement */
WHILE ( tempSet != empty) AND ( code != 1) DO  /* test for possibility of evolution */
  BEGIN /* WHILE */
    component = get-component(tempSet)
    IF ( dispensable(component) = FALSE AND replaceable(component) = FALSE ) THEN
      return failure with msg "VD cannot be evolved"
    ELSE IF ( dispensable(component) = FALSE ) THEN
      BEGIN
        code = 1 /* some view component is indispensable, must find replacement */
        tempSet = tempSet - component
      END
  END /* WHILE */
/* possible to evolve VD */
IF ( replaceable(R) = FALSE )
  THEN IF ( code = 1 )
         THEN return failure with msg "VD cannot be evolved"
         ELSE drop affected-component(VD,R) from VD
  ELSE /* replaceable(R) = TRUE */
    BEGIN
      found = find-substitute-relation(VD,R,S)
      IF (NOT found) THEN
        THEN IF ( code = 1 )
               THEN return failure with msg "VD cannot be evolved"
               ELSE drop affected-component(VD,R) from VD
      ELSE /* found */
        replace-relation(R,S)
    END /* replaceable(R) = TRUE */
```

Note that affected-components(R,V) set contains the relation R listed in the FROM clause, the attributes of R preserved in the SELECT clause, and the conditions in the WHERE clause that have one or two attributes of R as their operands.

Algorithm 10 ***PROCEDURE replace-relation(R,S):***

```
tempSet = affected-attr-components(VD,R) U affected-condition-component(VD,R)
While (tempSet != empty) DO
  BEGIN /* WHILE */
```

```
    component = get-component(tempSet)
    IF substitute S.B for component exists IN S
      THEN replace component by S.B
      ELSE drop component from VD
    tempSet = tempSet - component
  END /* WHILE */
replace R by S in FROM clause of VD
```

7.5 The Add-Relation Evolution Operator

This *add-rel(IS.R)* operator adds a new relation R to the IS site. It does not affect any views described in VKB, since none of the existing views refer to this new relation.

7.6 The Change-Relation-Name Evolution Operator

This *chg-rel-name(IS.R,S)* operator changes the name of the relation from R to S at site IS. Similarly to the chg-attr-name operation, this operation does not affect the view definitions that refer to R, assuming our system keeps a name-mapping table in the MKB along with other meta knowledge.

8 Related Work

To our knowledge, we are the first to study the problem of view synchronization caused by capability changes of participating ISs. In [RLN97], we establish a taxonomy of view adaptation problems that identifies alternate dimensions of the problem space, and hence serves as a framework for characterizing and hence distinguishing our view synchronization problem from other (previously studied) view adaptation problems. In [LNR97], we then lay the basis for the solutions presented in this current paper by introducing the overall *EVE* solution framework, in particular the idea of associating evolution preferences with view specifications. However, formal criteria of correctness for view synchronization as well as actual algorithms for achieving view synchronization are the key contributions of this current work. While no one has addressed the view synchronization problem as such, there are several issues we address for *EVE* that relate to work done before in other contexts as outlined below.

Gupta et al. [GJM96] and Mohania et al. [MD96] address the problem of how most efficiently to maintain a materialized view after a view redefinition explicitly initiated by the user takes place. They study under which conditions this view maintenance can take place without requiring access to base relations, i.e., the self-maintainability issue. Their algorithms could potentially be applied in the context of our overall framework, once *EVE* has determined an acceptable view redefinition. Their results are thus complimentary to our work.

In the work of Levy et al. [LSK95], a global information system is designed using the world-view approach where the external ISs are described relative to the unified world-view relations. The language used here to describe external relations relative to the world-view schema parallels our MKB description language, except the fact that we don't have an apriori defined schema. Further, we introduce the concept of a join constraint in our model that allows expressing default conditions among external relations that should be used by the system to attempt to integrate information instead of evaluating (blindly) all possible Cartesian combinations based on value matches (full disjunction) [NR98]. The problem of view evolution as posed by our work, i.e., that the world view itself may evolve, is not discussed in [LSK95].

Papakonstantinou et al. [PGMW95, PGMU96] are pursuing the goal of information gathering across multiple sources. Their proposed language OEM assumes queries that explicitly list the source identifiers of the database from which the data is to be taken. Like our MISD model, their data model allows ISs to describe their capabilities, but they don't assume that these capabilities could be changed and thus they do not address the view synchronization problem.

EVE system can be seen as an information integration system using view technology to gather and customize data across heterogeneous ISs. On this venue, related work that addresses the problem of information integration are among others the SIMS [AKS96] and SoftBot [EW94] projects. In the SIMS project, a unified schema is apriori defined and the user interaction with the system is via queries posed against the unified schema. Although addressing different issues, SIMS's process of translating a user query into subqueries targeting external relations raises some of the same problems as finding the right substitution for an affected view component in *EVE*. The SoftBot project has a very different approach to query processing as they assume that the system has to discover the "link" among data sources that are described by action schemas. While related to our view synchronization algorithms, the SoftBot planning process also has to discover connections among ISs when very different source description languages are used. None of the two projects address the particular problem of evolution under capability changes of participating external ISs.

We give a solution for a related problem in our transparent schema evolution (TSE) project [RR95, RR97], namely, to use view technology to handle schema changes transparently. However, this TSE work is all done in a centralized environment, assuming one single global database that is cooperating, i.e., that is maintaining all information possibly still used by any views defined on top of it. In the TSE framework, a user specifies schema changes against her special-tailored view schema defined over one common base schema. The TSE system is responsible for deriving an alternate view schema to simulate the effects of schema evolution while preserving the current view schemas. In TSE, the existing view schemas are not affected by schema changes, because the original base schema upon which they all are defined is always preserved. Unlike the problem addressed in this current paper, a delete operation specified against a view is not actually executed as a delete against the base schema rather simply desired data is hidden from that particular view. Thus the view evolution problem of *EVE* is not an issue in TSE.

In the University of Michigan Digital Library project [NR98, NR97], we have proposed the Dynamic Information Integration Model (DIIM) to allow ISs to dynamically participate in an information integration system. The DIIM query language allows loosely specified queries that the DIIM system refines into executable, well-defined queries based on the capability descriptions each IS exports when joining the DIIM system. For this, the notion of *connected relations* is introduced as a natural extension of the concept of full disjunction [GL94]. In the default case when only natural joins are defined in the IS descriptions in the MKB it then can be shown that the semantics of these two concepts (connected rules and full disjunction) are equivalent [NR97]. AI planning techniques are used in DIIM for query refinement. In *EVE*, instead, we now assume that precise (SQL) queries are used to define views (instead of loosely-specified ones), and thus query refinement in the sense of DIIM is not needed.

9 Conclusion

9.1 Current Status of *EVE*

A prototype of the *EVE* system has been implemented by the Database Systems Research Group at Worcester Polytechnic Institute. The *EVE* graphical user interface, the MKB, the MKB evolver, the VKB, and the view synchronizer are implemented using Java and C++, and the participating ISs are built on top of Oracle and Microsoft Access. The communication between *EVE* and the information space is via JDBC. The set of view synchronization algorithms presented in Section 7 are all fully implemented in this current prototype. The *EVE* system has been demonstrated at the CASCON'97 Technology Showcase in Toronto, Canada [LNR97].

9.2 Conclusion

Our effort is the first work to study the new problem of view adaptation in dynamic environments. This problem, which we call *view synchronization*, corresponds to the process of adapting view definitions triggered by capability changes of ISs. We propose the Evolvable View Environment (*EVE*) architecture as a generic framework within which to solve view adaptation when underlying ISs change their capabilities. The *EVE* approach is described in detail in the current paper. To summarize, the main contributions of this paper are:

- The identification of an open problem with current view technology in the context of dynamic large-scale environments such as the WWW, which we coin the *view synchronization problem.*
- The development of a general solution approach (and architecture), called the *EVE* framework, for addressing this view evolution problem based on the concept of view synchronization.
- The proposal of an extended view definition language, called E-SQL, that is capable of defining flexible views by incorporating view change preferences into the view definition.
- The design of an IS description model, called MISD, that can capture capabilities of diverse ISs, and thus serves as foundation for the view synchronization process.
- The development of formal foundations for view evolution and correctness criteria for the replacement of affected components of a view definition with alternate components.
- The introduction of a complete set of algorithms for view synchronization for all standard schema changes. The proposed algorithms generate view definitions as output that are consistent with both the change semantics expressed by E-SQL as well as the MISD descriptions captured in the meta knowledge base (MKB).
- The presentation of several scenarios that demonstrate that *EVE* maintains views in situations where state-of-the-art view technology would simply render the views undefined.
- The implementation of *EVE* concepts in a working system to demonstrate feasibility of the *EVE* ideas, and its demonstration at the CASCON'97 Technology Showcase in Toronto, Canada.

In short, this paper has opened up a new direction of research by identifying view synchronization as an important and so far unexplored problem of current view technology in dynamic large-scale environments such as the WWW. This work has laid a solid foundation for addressing the new problem of how to maintain views in dynamic environments, and is thus likely to be beneficial for many diverse applications such as web-based information services, electronic catalog providers, etc.

In the future, we plan to design and implement more complex view synchronization algorithms so a larger number of affected views could survive in dynamic distributed environments.

Acknowledgments. The authors would like to thank students at the University of Michigan Database Group and at the Database Systems Research Group at WPI for their interactions on this research. In particular, we thank Andreas Koeller, Yong Li, Xin Zhang, and Esther Dubin (CRA summer research student) for helping to proof-read the paper and for implementing components of the EVE system.

References

[AKS96] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6 (2/3):99–130, 1996.

[BCL89] J. A. Blakeley, N. Coburn, and P-A Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[CHA+95] M. J. Carey, L. M. Haas, P. M. Schwarz M. Arya, W. F. Cody, R. Fagin, Myron Flickner, A. W. Luniewski, Wayne Niblack, Dragutin Petkovic, J. H. Williams J. Thomas, and Edward L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering(RIDE): Distributed Object Management*, 1995.

[EW94] O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communication of ACM*, 1994.

[GJM96] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, 1996.

[GL94] C. Galindo-Legaria. Outerjoins as disjunctions . *Proceedings of SIGMOD*, 1994.

[LNR97] A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proceedings of IBM Centre for Advanced Studies Conference CASCON97, Best Paper Award*, pages 1–14, November 1997.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems. Special Issue on Networked Information Discovery and Retrieval*, 1995.

[MD96] M. Mohania and G. Dong. Algorithms for Adapting Materialized Views in Data Warehouses. *International Symposium on Cooperative Database Systems for Advanced Applications*, December 1996.

[NLR97] A. Nica, A.J . Lee, and E. A. Rundensteiner. View Synchronization with Complex Substitution Algorithms. Technical Report WPI-CS-TR-97-8, Worcester Polytechnic Institute, Dept. of Computer Science, 1997.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. View Preservation in Evolveable Large-Scale Information Systems. *To appear in Proceedings of International Conference on Extending Database Technology (EDBT'98)*, Valencia, Spain, March 1998.

[NR97] A. Nica and E. A. Rundensteiner. On Translating Loosely-Specified Queries into Executable Plans in Large-Scale Information Systems. In *Proceedings of Second IFCIS International Conference on Cooperative Information Systems CoopIS'97*, pages 213–222, June 1997.

[NR98] A. Nica and E. A. Rundensteiner. Loosely-Specified Query Processing in Large-Scale Information Systems. *International Journal of Cooperative Information Systems*, 1998.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A Mediation System Based on Declarative Specifications. In *Proceedings of IEEE International Conference on Data Engineering*, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[RLN97] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.

[RR95] Y. G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Schema Evolution. In *IEEE International Conference on Data Engineering*, pages 165–172, March 1995.

[RR97] Y. G. Ra and E. A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, September 1997.

[Ull89] J.D. Ullman. *Principle of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of International Conference on Information and Knowledge Management*, pages 25–30, November 1995.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.