The *EVE* Approach: View Synchronization In Dynamic Distributed Environments

Amy J. Lee, *Member*, *IEEE*, Anisoara Nica, *Member*, *IEEE*, and
Elke A. Rundensteiner, *Member*, *IEEE*

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 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 schema 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: 1) we provide an E-SQL view definition language with which the view definer can direct the view evolution process, 2) we introduce a model for information source description which allows a large class of ISs to participate in our system dynamically, 3) we formally define what constitutes a legal view rewriting, 4) we develop replacement strategies for affected view components which are designed to meet the preferences expressed by E-SQL, 5) we prove the correctness of the replacement strategies, and 6) we provide a set of view synchronization algorithms based on those strategies. A prototype of our *EVE* system has successfully been built using Java, JDBC, Oracle, and MS Access.

Index Terms—Data warehouses, view maintenance, query rewriting, view adaptation, view synchronization, view definition language distributed, evolving information sources, and source evolution.

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 internetworked information sources (ISs) [42], 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 [42]. 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 online 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 [42]. One important and as of now not yet addressed problem for these applications is that current view technology generally supports *static a priori-specified* view definitions—meaning that views are assumed to be specified on top of a fixed environment [17], [35]. Once the underlying ISs change their capabilities, the views derived from them may become undefined. It is this problem of view evolution caused by

• *A.J. Lee is with the Center for Human Resources, 921 Chatham Lane #100, Columbus, OH 43221. E-mail: alee@postoffice.chrr.ohio-state.edu.*
• *A. Nica is with iAnywhere Solutions, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2. E-mail: anica@ianywhere.com.*
• *E.A. Rundensteiner is with the Department of Computer Science, Worchester Polytechnic Institute, 100 Institute Rd., Worchester, MA 01609-2280. E-mail: rundenst@cs.wpi.edu.*

Manuscript received 29 Dec. 1997; revised 8 Sept. 2000; accepted 13 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 106131.

external environment changes (at the schema level rather than at the data level as done by practically all previous work on view maintenance [2], [42], [43]) that we tackle in this paper. We call this the *view synchronization* problem [35]. There are two exceptions to this previous view maintenance work for data changes, namely by Gupta et al. [9] and Mohania and Dong [24]. 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 is thus complimentary to work by others as once we have determined an acceptable view redefinition then algorithms proposed by others [9], [24] on how most efficiently to maintain the view, if materialized, could be applied to our system.

Furthermore, Levy et al. [21] as well as Arens et al. [1] have taken an alternative approach to information integration than we propose here based on creating a *global domain model*, i.e., an a priori defined type system fixed in time that defines all possible attributes and relations in a given domain ("world view"). Over such a domain model, information providers define views that specify which part of the world's data they provide. Consumers also query the domain model. An algorithm then rewrites a consumer's query in terms of the providers' views currently available and, thus, provides the consumer with whatever data happens to be available at the moment.

Here, in our approach we explore the inverse approach that does neither rely on a globally fixed domain nor on an ontology of permitted classes of data, both strong assumptions that are often not realistic. Rather, views are assumed to have been built in the traditional way over a number of base schemas and those views now must be adapted to base schema changes by rewriting them using information space redundancy and relaxable view queries as described in this paper. The benefit of this approach is that no predefined domain (which is hard to define and to maintain) is necessary, and that changes in the data provided can still be accommodated by automatically rewriting user queries (without human intervention). The core contribution of this current paper is the development of a solution approach to make this possible.

In [21], it is necessary to establish a world model before any source can provide information—a very complicated and often impossible task. Changes to the world model are not possible in this approach (and in fact are not discussed in the published literature on the world view approach). We expect that it would require a manual redefinition of possibly all information providers' and consumers' queries. Such a respecification of many/all source descriptions is obviously not desirable. Another drawback of this alternative approach is the insufficient handling of redundancy in the information space. If two information providers define partially overlapping view extents, Levy et al.'s algorithms find the "minimal cover" for the queried data, i.e., uses information from a randomly picked information source that satisfies the user's query. In contrast to this approach, we can make use of known overlaps of source data to provide nonequivalent rewrites of queries in the case of the possible unavailability of one of the sources.

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

Example 1. Assume a traveler 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:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } \mathit{Travel\text{-}Info\text{-}in\text{-}Boston}\text{AS} \\ \text{SELECT} & C.Name, C.Address, C.Phone, H.Name, \\ & H.Address, H.Phone \\ \text{FROM} & CarRental C, BostonHotel H \\ \text{WHERE} & (C.Val\text{-}Pak\text{-}Partnership =' Yes') \text{ and} \\ & (H.Val\text{-}Pak\text{-}Partnership =' Yes'), \end{array} \qquad (1)$$

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

Assume, for some reason that the BostonHotel relation cannot be accessed (this effect could be caused if the IS that provided the BostonHotel relation goes out of business). In state-of-the-art view technology, 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 BostonHotel relation is undefined". We, on the other hand, propose several potential ways to "remedy" this view definition evolution. To name a few:

1. Assume there is a MAHotel relation that has the lodging information for the entire Massachusetts state (that is, MAHotel $\supseteq$ BostonHotel). Query 1 can be rewritten to have the BostonHotel relation replaced by the MAHotel 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 $\subset$ BostonHotel). Query 1 can be rewritten to have the BostonHotel relation replaced by the BackBayHotel relation, which is likely to return useful answers for the traveler but it will not be a complete listing of all answers for the initial query.
3. The traveler 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 BostonHotel relation and the attributes referencing the BostonHotel 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 [29] and the Garlic project [3], 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 replacements 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 [17], [35].

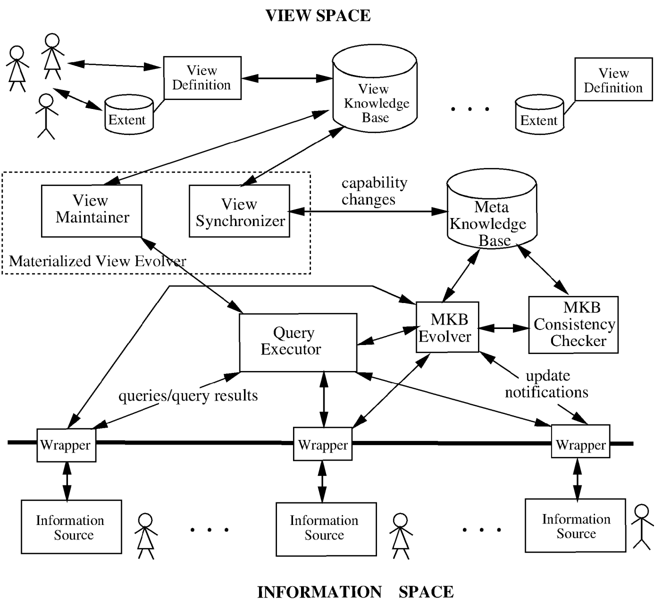

Fig. 1. The framework of the Evolvable View Environment (EVE).

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 MS Access, and it is running in the Database Systems Research Lab at the Worcester Polytechnic Institute. An online EVE demonstration can found at the DSRG project web site at http:\\davis.wpi.edu\dsrg.

The EVE system has also been formally demonstrated in ACM SIGMOD '99 [33].

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 a 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* attempts to evolve views when they are affected by schema 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 (Fig. 1).

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 ISs. 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.

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 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) [17], [35] 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 [29].

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 schema changes in the information space. In our framework, each IS will notify ,via the wrapper interface, the MKB of any such schema changes so that they can be properly registered in the MKB. The MKB Evolver module will then take appropriate actions to update the MKB [26]. 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 (Fig. 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 schema change.

View Synchronization. The *view synchronizer* tool (Fig. 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 that it becomes 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 are 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.

IS ID	Relation Provided	Remark
IS 1:	Customer(SSN, Name, Address, City, Phone, Age)	Customer Information
IS 2:	Tour(TourID, TourName, Type, Duration)	Tour Information
IS 3:	Participate(AcctNo, PSSN, PName, TourID, StartingDate)	Tour Participation Information
IS 4:	FlightRes(PName, Airline, FlightNo, Source, Dest, Date)	Flight Reservation Information
IS 5:	Accident_Ins(AcctID, Holder, Amount, Birthday)	Insurance Information
IS 6:	CarRental(Name, Address, Phone, City, State, Country)	Car Rental Information
IS 7:	Hotel(Name, Address, Phone, City, State, Country)	Hotel Information

Fig. 2. Descriptions of relevant information sources.

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 Fig. 2.

Assume the headquarter maintains complete sets of information of the customers, tours, and tour participants in the following formats: Customer(SSN, Name, Address, City, Phone, Age),[1] Tour(TourID, TourName, Type, Duration)—where Type = {luxurious, economy, super–valued}, and Participate(AcctNo, PSSN, PName, 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(AcctID, 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:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW}\mathit{Asia\text{-}Customer}\text{AS} \\ \text{SELECT} & \mathit{Name}, \mathit{Address}, \mathit{Phone} \\ \text{FROM} & \mathit{Customer}\ C, \mathit{FlightRes}\ F \\ \text{WHERE} & (C.Name = F.PName)\ \text{AND} \\ & (F.Dest = '\ Asia') \end{array} \quad (2)$$

Note that Query 2 is a static a priori-specified query. 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. For simplicity's sake, we assume in this work that views are defined by the SELECT-FROM-WHERE SQL syntax with a conjunction of primitive clauses in the WHERE clause, where a primitive clause has one of the following forms:

$$(\langle \mathit{Attribute\_Name} \rangle\ \theta\ \langle \mathit{Attribute\_Name} \rangle)$$

or

$$(\langle \mathit{Attribute\_Name} \rangle\ \theta\ \langle \mathit{value} \rangle)$$

with

$$\theta \in \{<, \leq, =, \geq, >\}.$$

E-SQL is an extension of the SELECT-FROM-WHERE queries augmented with specifications for how the query 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 the E-SQL view definition language is given in Fig. 3.

In Fig. 3, the set Local_Column_List corresponds to the local names given to attributes preserved in the view V. The evolution 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 Fig. 4. In the table, each type of evolution parameter used in E-SQL is represented by a row, with column one giving the parameter name and the abbreviation for the parameter, column two the

1. Note that when it is not necessary to explicitly specify the full name of an attribute, we omit the IS ID.

$$\boxed{\begin{array}{ll}
\textsf{CREATE} & \textsf{VIEW}\ V\ [(Local\_Column\_List)]\ [(\mathcal{VE} =\ [``\equiv"|``\supseteq"|``\subseteq"|``\approx"])]\ \textsf{AS} \\
\textsf{SELECT} & Attribute\_Name\ [(\mathcal{AD} = [true|false], \mathcal{AR} = [true|false])] \\
 & [, Attribute\_Name\ [(\mathcal{AD} = [true|false], \mathcal{AR} = [true|false])] \ldots] \\
\textsf{FROM} & Relation\_Name\ [(\mathcal{RD} = [true|false], \mathcal{RR} = [true|false])] \\
 & [, Relation\_Name\ [(\mathcal{RD} = [true|false], \mathcal{RR} = [true|false])] \ldots] \\
\textsf{WHERE} & Primitive\_Clause\ [(\mathcal{CD} = [true|false], \mathcal{CR} = [true|false])] \\
 & [\textsc{and}\ \ Primitive\_Clause\ [(\mathcal{CD} = [true|false], \mathcal{CR} = [true|false])] \ldots]
\end{array}} \tag{3}$$

Fig. 3. General syntax of the E-SQL view definition language.

possible values of the parameter can take on plus the associated semantics, and column three the default value.

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

Two evolution parameters are attached to each view component. 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 primitive clause 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 (when its value is true) or the view component cannot be replaced (when the value is false). A view definer can also specify that the evolved view extent 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 "$\supseteq$ "), with respect to the original view extent using the $\mathcal{VE}$ parameter. If no restrictions on the view extent are given, then $\mathcal{VE}$ is set to "$\approx$."

When the parameter setting is omitted from the view definition, then the default value is assumed. This means that a conventional SQL query (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 use an example to demonstrate the utility and usage of the evolution parameters.

Example 3. Assume the travel agency (i.e., the view definer) states the following preference when she specifies Query 2:

- The travel agency is willing to put off the phone marketing strategy as long as it can reach out to its customers by sending out promotion invitation letters to them. That is, the customer's phone number attribute is dispensable, if it is deleted from the relation Customer for some reason and a suitable substitute cannot be found.
- The travel agency will accept the name, address, and phone number information even if the data comes from other source(s). (Replacability of these attributes is permitted.)
- Both relations in the FROM clause are essential components of the view. (Then, relation replacability is set of false.)
- The Customer relation may be obtained from other source(s), but not the FlightRes relation. That is, the Customer relation is replacable.

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 ($\mathcal{VE}$)	$\approx$: 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	$\equiv$

Fig. 4. View evolution parameters of E-SQL language.

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

Fig. 5. Possible types of semantic constraints for IS descriptions.

- The first primitive clause in the WHERE clause, an equijoin operation that joins the Customer relation with the FlightRes relation by customer' names, is necessary for the view to be useful to its users. Thus, the system must maintain it. However, the data may come from other source(s).
- The second primitive clause, a local condition specified on the FlightRes relation, finds all the passengers who travel to Asia. The travel agency is willing to accept a view without this condition specified,[2] i.e., having the promotion the travel agency is willing to make contact with more customers, e.g, those traveling to the Far-East as well. Thus, sending invitation letters to all customers traveling by air, but not necessary to Asia, is acceptable though not desirable.

Expressing all the above preferences for evolution mentioned above, we now augment the SQL query given in (2) with the corresponding evolution parameters, which then results in the E-SQL query given in (4).

$$\begin{array}{ll}
\text{CREATE} & \text{VIEW Asia-Customer } (\mathcal{VE} = ''\supseteq'') \text{ AS} \\
\text{SELECT} & \text{Name } (\mathcal{AR} = true), \text{Address } (\mathcal{AR} = true), \\
 & \text{Phone } (\mathcal{AD} = true, \mathcal{AR} = true) \\
\text{FROM} & \text{Customer C } (\mathcal{RR} = true), \text{FlightRes F} \\
\text{WHERE} & (\text{C.Name} = \text{F.PName})\ (\mathcal{CR} = true) \\
 & \text{AND (F.Dest} = '\text{Asia}')\ (\mathcal{CD} = true).
\end{array} \tag{4}$$

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

In summary, E-SQL is a base model of extending SQL with evolution preferences, and many additional extensions are possible to refine the model. For example, information of which sources are acceptable as replacements could be added to the replacement parameter. It is however our goal to keep the model as simple as possible until a clear need for a more fine-grained preference model arises driven by some application needs.

2. 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.

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*. Fig. 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 Fig. 1), forming an information pool that is critical in finding appropriate replacements for view components when view definitions become undefined.

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 $IS.R_1$, $IS.R_2, , IS.R_n$. 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:

$$IS.R(A_1, \ldots, A_n). \tag{5}$$

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:

$$\mathcal{TC}_{R(A_i)} = (R(A_i) \subseteq A_i(Type_i)), \tag{6}$$

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:

$$\begin{aligned} &\mathcal{TC}_{R(A_1,\ldots,A_n)} = \\ &(R(A_1, \ldots, A_n) \subseteq A_1(Type_1) \times \ldots \times A_n(Type_n)) \end{aligned} \tag{7}$$

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.[3] If two attributes are exported with the

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.

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

Fig. 6. Relevant join constraints for Example 2.

same name, they are assumed to have the same type (which must be reflected by the type integrity constraints for their relations).

5.3 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. If no join constraint is specified between two relations, then we consider any possible relationship between them to be coincidential, and, hence, will not attempt to join between them for replacement purposes. A generic join constraint is as follows:

$$\mathcal{JC}_{R_1,R_2} = (C_1 \ AND \ \cdots \ AND \ C_l), \tag{8}$$

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

Example 4. Some of the join constraints for our running example presented in Section 3 are given in the table of Fig. 6.

5.4 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:

$$\begin{aligned}\mathcal{PC}_{R_1,R_2} = \big(&\pi_{A_{i_1},\dots,A_{i_k}}(\sigma_{\mathcal{C}(A_{j_1},\dots,A_{j_l})}R_1) \\ &\theta\pi_{A_{n_1},\dots,A_{n_k}}(\sigma_{\mathcal{C}(A_{m_1},\dots,A_{m_t})}R_2)\big),\end{aligned} \tag{9}$$

where $A_{i_1}, \dots, A_{i_k}, A_{j_1}, \dots, A_{j_l}$ are attributes of R_1;

$$A_{n_1}, \dots, A_{n_k}, A_{m_1}, \dots, A_{m_t}$$

are attributes of

$$R_2; \mathcal{TC}(R_1.A_{i_s}) = \ \mathcal{TC}(R_2.A_{n_s}),$$

for $s = 1, \dots, k$; and θ is $\{\subseteq, \supseteq, \equiv\}$ for the partial ($\subseteq$ and $\supseteq$) or complete ($\equiv$) information contraint, respectively.

Example 5. 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 (10) states that the MABranch relation is contained in the Customer relation:

$$\begin{aligned}&\mathcal{PC}_{MABranch,Customer} = \\ &\big(\pi_{Name,Address}(MABranch) \subseteq \pi_{Name,Address}(Customer)\big).\end{aligned} \tag{10}$$

6 View Evolution Foundations

Given a schema change of an underlying IS, *EVE* finds views in the VKB affected by the schema 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 schema 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 schema change if the deleted schema 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:*

- ***Rule 1.*** *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. Note that C may be replaced by either one C′ or possibly by more than one new view component. In this case, we say that each of the new view components C′ replaces one view component, namely, C. (See Example 6).*

- ***Rule 2.*** *Assume a new view component C' is used to replace several view components $X_1, ..., X_k$ of the original view with evolution parameter settings being*

$$X_1(par_{1,1} = val_{1,1}, par_{1,2} = val_{1,2}), \ldots,$$
$$X_k(par_{k,1} = val_{k,1}, par_{k,2} = val_{k,2}),$$

where $par_{i,1}$ and $par_{i,2}$ are view evolution parameters for X_i and $val_{i,j} \in \{true, false\}$. Then, we set the two evolution parameters of C' to

$$\big(par_{C',1} = (val_{1,1} \text{AND} \cdots \text{AND } val_{k,1}),$$
$$par_{C',2} = (val_{1,2} \text{AND} \cdots \text{ AND } val_{k,2})\big).$$

(See Example 7).

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

Example 6. An example when Rule 1 is applied is given first. Let a view $V1$ be defined as follows:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } Customer\_In\_Boston \text{AS} \\ \text{SELECT} & C.Phone(\mathcal{AR} = true), C.Name \\ \text{FROM} & Customer\ C \\ \text{WHERE} & (C.City =' Boston'). \end{array} \qquad (11)$$

Assume the *Phone* attribute is deleted from the *Customer* relation. Note that *Phone* is referenced in the SELECT clause, but not in the WHERE clause. We further assume the view synchronizer finds a counterpart in another relation *Phone_Customer*, which can be joined with *Customer* based on attributes other than *Customer.Phone*, i.e.,

$$\mathcal{JC}_{Customer,Phone\_Customer} = (Customer.SSN = Phone\_Customer.SSN).$$

Therefore, one rewriting is as follows:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } Customer\_In\_Boston' \text{ AS} \\ \text{SELECT} & \underline{P.Phone\ (\mathcal{AR} = true)}, C.Name \\ \text{FROM} & Customer\ C, \\ & \underline{Phone\_Customer\ P\ (\mathcal{RR} = true)} \\ \text{WHERE} & (C.City = '\ Boston') \text{ AND} \\ & \underline{(C.SSN = P.SSN)\ (\mathcal{CR} = true)}. \end{array} \qquad (12)$$

In this example, the view component *C.Phone* and its associated evolution parameters ($\mathcal{AR} = true$) are replaced by three new view components, all of which are underlined in Query (12) (using Rule 1 from Definition 4). Each of the new view components has its evolution parameters set equal to that of the replaced view component, i.e., the replaceability parameter is true due to ($\mathcal{AR} = true$) and the dispensibility one takes on the default value due to $\mathcal{AD}$ having the default value.

Example 7. This example now shows how Rule 2 is applied. Let a view *Insured_Participant* be defined as follows:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } Insured\_Participant \text{ AS} \\ \text{SELECT} & P.PName\ (\mathcal{AD} = false, \mathcal{AR} = true), \\ & P.Tour\ ID \\ \text{FROM} & Participate\ P, Accident\_Ins\ A \\ \text{WHERE} & (P.PName = A.Holder) \\ & (\mathcal{CD} = true, \mathcal{CR} = true). \end{array} \qquad (13)$$

Assume *P.PName* is deleted from its site. Note that *P.PName* is referenced in the SELECT and in the WHERE clauses. Therefore, one rewriting is as follows:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } Insured\_Participant' \text{ AS} \\ \text{SELECT} & \underline{C.Name\ (\mathcal{AR}\ =\ true)}, P.TourID \\ \text{FROM} & Participate\ P, Accident\_Ins\ A, \\ & \underline{Customer\ C\ (\mathcal{RR} = true)} \\ \text{WHERE} & \underline{(C.Name\ =\ A.Holder)} \\ & \underline{(\mathcal{CD} = true, \mathcal{CR} = true)} \text{ AND} \\ & \underline{(C.SSN = P.PSSN)\ (\mathcal{CR} = true)}. \end{array} \qquad (14)$$

In this example, there are four new view components (underlined) in *Insured_Participant'*. Two among the four, *Customer* in the FROM clause and ($C.SSN = P.PSSN$) in the WHERE clause, are brought in by the overall replacement process for replacing two affected view components—*P.PName* in the SELECT clause and ($P.PName = A.Holder$) in the WHERE clause of *Insured_Participant*. Therefore, their evolution parameters are set using Rule 2. That is, the evolution parameters of *Customer* and ($C.SSN = P.PSSN$) are both set to ($false, true$).

Definition 5 (Legal Rewriting). *Given a schema 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 not affected by the schema 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 schema change.*[4] *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',*

4. We do not go into depth on how the MKB changes in this paper due to space limitations.

and the nonreplaceable view components are not replaced with information taken from other sources.

P5. *If the view-extent parameter is different than "approximate" ("$\approx$ "), 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 (15).*

$$\pi_{Attr(V)\cap Attr(V')}(V') \; \phi \; \pi_{Attr(V)\cap Attr(V')}(V). \tag{15}$$

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

$$\begin{aligned}&\textit{if view-extent parameter } \mathcal{VE} \textit{ is } ''\equiv'', && \textit{then } \phi \textit{ must be } ''\equiv'';\\ &\textit{ifview-extent parameter } \mathcal{VE} \textit{ is}''\subseteq'', && \textit{then } \phi \textit{ must be } ''\subseteq'' \textit{ or}''\equiv''; \textit{ and}\\ &\textit{if view-extent parameter } \mathcal{VE} \textit{ is } ''\supseteq'', && \textit{then } \phi \textit{ must be } ''\supseteq'' \textit{ or } ''\equiv''.\end{aligned} \tag{16}$$

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 schema 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.

Condition 1 (Type Match Condition). This condition requires that $S.B$ has the same domain type as the attribute $R.A$. That is, there exist in MKB the following constraints for some type $Type_1$:

1. $\mathcal{TC}(S.B) = (S(B) \subseteq Type_1)$ and
2. $\mathcal{TC}(R.A) = (R(A) \subseteq Type_1)$.

Condition 2 (Tuple Linkage Condition). This requirement demands that there exists a meaningful join relationship between the relations R and S which indicates to us that it is semantically meaningful to join the two relations on those attributes. In some cases, the extents of the two relations may be identical by coincidence even if their semantic meaning is unrelated, and should not be used to replace one another. In our model, this means that there exists a *join constraint* in the MKB between R and S such that the attribute $R.A$ is not used in the join condition:

$$\mathcal{JC}_{R,S} = (C_1(\bar{J}_1) \; AND \cdots \; AND \; C_m(\bar{J}_m)) = \mathcal{C}(\bar{J}), \tag{17}$$

where for all $1 \leq i \leq m$, $\bar{J}_i$ denotes an ordered list of attributes defined for R or S, $C_i(\bar{J}_i)$ is a primitive clause involving those attributes, 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 $\bar{J} = \bar{J}_1 \cup \ldots \cup \bar{J}_m$. In short, $\mathcal{C}(\bar{J})$ is a predicate over R and S not making use of attribute R.A.

Condition 3 (Extent Satisfaction Condition). We also need some knowledge about the extent relationships between the relation R and the relation S used as its replacement, which in our model would typically be expressed by some PC or so called containment constraint. For this, let us assume the value of the view-extent parameter of the view V to be δ. We then impose the extent condition given in (18):

$$\begin{aligned}&\pi_{((Attr(V)\cap Attr(R))\setminus\{R.A\})\cup\{S.B\}}\left(R \bowtie_{\mathcal{C}(\bar{J})} S\right)\\ &\phi\pi_{((Attr(V)\cap Attr(R))\setminus\{R.A\})\cup\{R.A\}}(R),\end{aligned} \tag{18}$$

where $\mathcal{C}(\bar{J})$ is the join condition defined by Condition 2, that is R and S are joined using this join criteria given as semantically-correct by the MKB. And $Attr(V)$ represents all the attributes referred in the SELECT and WHERE clauses of the view V. $((Attr(V) \cap Attr(R))$ denotes all attributes of R that are in the view (both in the old and new view), except for R.A and S.B with R.A being replaced by S.B. The above equation thus indicates that if we take all attributes of R used in the view extended by the attribute S.B, where S.B is joined to the remainder using the join constraint from Condition 2, then all such new tuples are in the required extent relationship with the original tuples from R. Note that the projection lists in the above (18) 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 extent relationship operator ϕ in (18) must satisfy the conditions imposed in (16) with respect to the view-extent parameter $\mathcal{VE}$, unless $\mathcal{VE} =''\approx''$. If $\mathcal{VE} =''\approx''$, then, of course, no rigid extent requirements need to be imposed. The above condition stated more formally in (18) is *sufficient* to assure that the view-extent parameter $\mathcal{VE}$ is always satisfied.

The following theorem states that Conditions 1, 2, and 3 are *sufficient* to obtain a legal rewriting by using the attribute $S.B$ for replacing the attribute $R.A$ in a view definition. By Definition 5, a rewriting is legal if its view-extent parameter $\mathcal{VE}$ is satisfied.

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

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } V(\mathcal{VE} = \delta) \text{ AS} \\ \text{SELECT} & R.A, R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\ \text{FROM} & R, R_1, \ldots, R_n \\ \text{WHERE} & \mathcal{CV}(\bar{W}), \end{array} \qquad (19)$$

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 (17) to the WHERE *clause. V′ obtained in this way is shown in (20) (where the new view components are underlined).*

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } V(\mathcal{VE} = \delta) \text{ AS} \\ \text{SELECT} & \underline{S.B}, R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n \\ \text{FROM} & \underline{S}, R, R_1, \ldots, R_n \\ \text{WHERE} & \underline{\mathcal{CV}'((\bar{W} \setminus \{R.A\}) \cup \{S.B\}) \text{ AND } \mathcal{C}(\bar{J})}, \end{array} \qquad (20)$$

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 (19) where all occurrences of the attribute R.A were replaced by the attribute S.B.*

Then, $V' \; \delta \; V$.

In the view definition in (19), we assume that $R_i.\bar{D}_i$ denotes a subset of attributes from relation R_i projected out in this view. And, $\mathcal{CV}(\bar{W})$ is a complex condition over possibly all relations $R, R_1, \ldots, R_n$ in the FROM clause. Now, let us assume that S and S.B are a relation and a corresponding attribute respectively that according to above meet all attribute substitution criteria (that is, Conditions 1, 2, and 3). This means that by Condition 2, there exists a join constraint for R and S in the MKB along the line of the Tuple Linkage Condition, and by Condition 3, there exists a PC constraint in the MKB along the line of the Extent relationship. Then, we can rewrite V simply by 1) replacing R.A by S.B everywhere in V and 2) by adding the required condition (taken from the join constraint from Condition 2) between R and S to link S with the view in a semantically meaningful manner. This replacement is exactly what is more formally listed in (20).

Proof. The proof for this theorem is lengthy and, thus, is given in the appendix (Appendix A) instead.

The following lemmas are now special cases of $\mathcal{PC}$s and $\mathcal{JC}s$ constraints that are sufficient to assure that Conditions 1, 2, and 3 hold. When we know that the three conditions hold, then this in turn by Theorem 1 would imply that in these cases, the relation S corresponds to a good replacement for relation R. Thus, these theorems below establish guidelines as to what meta knowledge in the form of $\mathcal{PC}$s and $\mathcal{JC}$s constraints would be sufficient for a replacement to be considered legal. In other words, it provides us with situations when Theorem 1 is applicable.

Lemma 1. *Let V be defined as in (19) and $\delta = {}''\supseteq''$. Let S be a relation with the following constraints:*

I. $\mathcal{JC}_{R,S} = (R.\bar{A}_1 = S.\bar{B}_1)$ *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 $\phi = {}''\supseteq''$ for the relation S and the attribute S.B.

Lemma 1 examines now the special case that $''VE = \supseteq''$, i.e., that the new view extent will be a superset or equal to the old view extent. In this case, Constraint (I) in the above lemma assures that the join constraint between R and S required by the Tuple Linkage Condition indeed exists in the MKB. In particular, it assures that the attribute R.A, to be dropped from R by this attribute substitution will not be used by this join condition. Hence, the Tuple Linkage Condition (Condition 2) then holds. Constraint (II) in the above lemma assures that relation R and its replacement relation S stand in the right extent relationship. It is, of course, sufficient to only consider in this attributes of R that are used in the view, the other ones are irrelevant. This first assures the Type Match condition (Condition 1) to also hold. Finally, Constraint (II) assures that all tuples of R for the attributes of interest are all contained in S, including the attributes used by the "join criteria." Hence, R joined with S on this join field given by Constraint (I) in Lemma 1 will get us all the old extent of R back and possibly some additional tuples. Since $R'' \subseteq'' S$, we have $\phi = {}''\subseteq''$.

Lemma 2. *Let V be defined as in (19) and $\delta = {}''\subseteq''$. Let S be a relation with the following costraints:*

I. $\mathcal{JC}_{R,S} = (R.\bar{A}_1 = S.\bar{B}_1 \; AND \; \{\mathcal{C}(\bar{J})\})$ *with* $A \notin (\bar{A}_1 \cup \bar{J})$ *and* $\mathcal{C}(\bar{J})$ *a conjunction of local*[5] *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.*

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

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

Lemma 2 instead examines the case that $VE = "\subseteq"$, i.e., that we must assure that the view rewriting process produces a smaller or equal extent than the old view extent when replacing R.A by S.B. This is very similar to Lemma 1, requiring that both an appropriate join and PC constraint can be identified. The main difference now is that additional conditions may be placed on JC that further restrict what the output view may hold in terms of its extent. Given that S is possibly smaller in terms of number of tuples than R, then it may, of course, be possible that the final view extent may also be smaller when using the attribute from S instead from R.

The above lemmas are special cases of Theorem 1 and their proofs are similiar to the one of Theorem 1 (See Appendix A). They are omitted here.

6.2.2 Principles of Relation Substitution

When a relation $IS_1.R$ referred to 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. In this case, we do not require any join constraint between R and S, since R will be completed removed and, thus, replaced by S. Instead, the conditions set up below check that a) S has all necessary attributes in its interface and b) that the subchunk of S used in the rewritten view will indeed stand in a correct set relationship with the extent of R. This replacement of a complete relation is thus simpler than just replacing one attribute as done in Section 6.2.1.

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 $\mathcal{CR} = true$ must have acceptable counterparts in relation S. Otherwise, the new view using S would no longer be legal by Definition 5.

This requirement can now formally be stated as given below. We use the notation $Attr(V(R))_{\text{SELECT}}(d,r)$ 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:

$$Attr(V(R))_{\text{SELECT}}(d,r) = \{R.A \mid R.A \textit{ in } \text{SELECT } \textit{clause}, \mathcal{AD}(R.A) = d, \mathcal{AR}(R.A) = r\}. \tag{21}$$

And, we use $Attr(V(R))_{\text{WHERE}}(d,r)$, 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:

$$Attr(V(R))_{\text{WHERE}}(d,r) = \{R.A \mid R.A \textit{ in a condition } C \textit{ from } \text{WHERE } \textit{clause}, \mathcal{CD}(C) = d, \mathcal{CR}(C) = r\}. \tag{22}$$

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

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

$$\begin{aligned} &Attr(V(R))_{\text{SELECT}}(false, true) \\ &\cup Attr(V(R))_{\text{WHERE}}(false, false) \\ &\cup Attr(V(R))_{\text{WHERE}}(false, true) \\ &\cup \cup Attr(V(R))_{\text{WHERE}}(true, false) \\ &\cup Attr(V(R))_{\text{WHERE}}(true, true) \subseteq S. \end{aligned} \tag{23}$$

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

$$\begin{aligned} &Attr(V(R))_{\text{SELECT}}(false, true) \cup \\ &Attr(V(R))_{\text{WHERE}}(false, true) \subseteq S. \end{aligned} \tag{24}$$

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). Since our goal is to replace R by S, we must determine their extent relationship. 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:

$$\pi_{\bar{B}}(S)\ \phi\ \pi_{\bar{A}}(R), \tag{25}$$

where $\bar{A} \in R$ 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 $R.\bar{A}$. This just says that all subtuples in the view are in the correct extent relationship by this view evolution constraint.

Thus, the following conditions must hold:

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

$$\begin{aligned}&Attr(V(R))_{\text{SELECT}}(false, true)\\ &\cup Attr(V(R))_{\text{WHERE}}(false, false)\\ &\cup Attr(V(R))_{\text{WHERE}}(false, true)\cup\\ &\cup Attr(V(R))_{\text{WHERE}}(true, false)\\ &\cup Attr(V(R))_{\text{WHERE}}(true, true) \subseteq \bar{A}.\end{aligned} \tag{26}$$

The first part of this equation simply states that any attribute projected by the view and is not dispensible must be kept. The remainder of this equation then states that all attributes used to constrain the original view, i.e., used in conditions in the WHERE clause, must still be preserved in the new rewritten view, otherwise it is guaranteed to be a subset of the old view.

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

$$\begin{aligned}&Attr(V(R))_{\text{SELECT}}(false, true)\cup\\ &Attr(V(R))_{\text{WHERE}}(false, true) \subseteq \ \bar{A}.\end{aligned} \tag{27}$$

In this case, the first part of this equation remains unchanged. The second part however now only assures that the attributes used in conditions that are replacable but not dispensible are preserved. We no longer have to assure that the extent is a subset, hence having dropped some of the original predicates from the WHERE clause will be acceptable.

Finally, we also require that the values of δ and ϕ must satisfy the property from (16), unless the value of the view-extent parameter has no rigid constraint and is "approximate", i.e., $\delta \neq\approx$. This assures that the extent relationship of the old with the new view alignes up in the same subset relationship as the old relation R with the new relation S.

The above three conditions are *sufficient* to have the view-extent evolution parameter $\mathcal{VE}$ satisfied when S is used to replace the relation R. Note that they are however not *necessary*. This is exactly what is stated in a formal manner in the following theorem.

Theorem 2. *Let a view V be defined as follows:*

$$\begin{array}{ll}\text{CREATE} & \text{VIEW } V \ \ (\mathcal{VE} = \delta) \text{ AS}\\ \text{SELECT} & R.\bar{D}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n\\ \text{FROM} & R, R_1, \ldots, R_n\\ \text{WHERE} & \mathcal{CV}(\bar{W}),\end{array} \tag{28}$$

where all attributes of R in $\bar{W}$ are denoted by $R.\bar{D}'$, and all other notations are otherwise equal to those in the view definition in Theorem 1.

Let S be a relation that satisfies Conditions 1, 2, and 3 for relation substitution. 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 (29) (where the new view components are underlined).

$$\begin{array}{ll}\text{CREATE} & \text{VIEW } V' \ \ (\mathcal{VE} = \delta) \text{ AS}\\ \text{SELECT} & \underline{S.\bar{F}}, R_1.\bar{D}_1, \ldots, R_n.\bar{D}_n\\ \text{FROM} & \underline{S}, R_1, \ldots, R_n\\ \text{WHERE} & \underline{\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')}.\end{array} \tag{29}$$

In (29), $S.\bar{F}$ are the attributes from $S.\bar{B}$ corresponding to the attributes from $R.\bar{A} \cap R.\bar{D}$. $S.\bar{F}'$ are the attributes from $S.\bar{B}$ corresponding to the attributes from $R.\bar{A} \cap R.\bar{D}'$. $\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')$ is the conjunction of primitive clauses in the WHERE clause of the view V defined in (28) 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' δ V.

In (29), the attributes $S.\bar{F}$ from $S.\bar{B}$ are used to denote correspondances with the attributes from $R.\bar{A} \cap R.\bar{D}$. For those, we can observe the following. From Conditions 2 and 3, we have that $S.\bar{F}$ corresponds to a superset of the attributes in $Attr(V(R))_{\text{SELECT}}(false, true)$. Similarly, $S.\bar{F}'$, refers to the attributes from $S.\bar{B}$ that correspond 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 $Attr(V(R))_{\text{WHERE}}(false, true)$.

Proof. The proof for this theorem is lengthy and, thus, is given in the appendix (Appendix B) instead.

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:

- 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 [29]. This means that view queries are assumed to prefix the names of relations and attributes with the identifiers of the ISs to which they belong to, if needed to disambiguiate names.

We believe our solution approach could be easily adapted for a more general case when the assumptions are relaxed. The schema 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).

```
00.  Success = TRUE
01.  IF attribute-replaceable (A) = FALSE
02.  THEN IF attribute-dispensable (A) = TRUE
03.          THEN drop A from V    /* report success */
04.          ELSE /* attribute-dispensable (A) = FALSE */
05.                 Success = FALSE    /* report failure */
06.       END IF
07.  ELSE /* attribute-replaceable (A) = TRUE */
08.       IF attribute-dispensable (A) = TRUE
09.       THEN find-substitute-select (A, B)
                   /* see Section 6.2.1 */
10.            IF    found
11.            THEN replace-attribute (A,B)
                     /* report success */
12.            ELSE                      /* not found */
13.                   drop A from V
                          /* report success */
14.            END IF
15.       ELSE /* attribute-dispensable (A) = false */
16.            find-substitute-select (A, B)
                             /* see Section 6.2.1 */
17.       IF found
18.       THEN replace-attribute (A,B)
                   /* report success */
19.       ELSE /* not found */
20.              Success = FALSE       /* report failure */
21.       END IF
22.  END IF
23.  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* schema change.

Example 8. *For easy reference, we redisplay Query (8) first introduced in Section 4.*

$$\begin{array}{ll} \text{CREATE} & \text{VIEW Asia-Customer } (\mathcal{VE} = "\supseteq") \text{ AS} \\ \text{SELECT} & (\text{Name, Address, Phone} \\ & (\mathcal{AD} = true, \mathcal{AR} = true) \\ \text{FROM} & \text{Customer C } (\mathcal{RR} = true), \text{FlightResF} \\ \text{WHERE} & (\text{C.Name} = \text{F.PName})\ (\mathcal{CR} = true) \\ & \text{AND (F.Dest} = '\text{Asia}')\ (\mathcal{CD} = true) \end{array} \tag{30}$$

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 (31) (new components are underlined):*

$$\begin{array}{ll} \text{CREATE} & \text{VIEW Asia-Customer } (\mathcal{VE} = "\supseteq") \text{ AS} \\ \text{SELECT} & (\text{Name, Address} \\ & \underline{C2.Phone(\mathcal{AD} = true, \mathcal{AR} = true)} \\ \text{FROM} & \text{Customer C } (\mathcal{RR} = true, \text{FlightResF}, \\ & \underline{CustomerBak\ C2\ (\mathcal{RD} = true, \mathcal{RR} = true)} \\ \text{WHERE} & (\text{C.Name} = \text{F.PName}) \text{ AND } (F.Dest = '\ Asia') \\ & (\mathcal{CD} = true)\text{AND} \\ & \underline{(C2.Name = C.Name)(\mathcal{CD} = true, \mathcal{CR} = true)} \end{array} \tag{31}$$

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 (30) 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}
02.  Success = TRUE
03.  WHILE (C != empty) AND (Success) DO
04.      take c from C
05.      IF condition-replaceable (c) = FALSE
06.         IF condition-dispensable (c) = TRUE
07.         THEN
08.               C = C - c; drop c from V;
09.         ELSE /* condition-dispensable (c) = FALSE */
10.             Success = FALSE
11.         END IF
12.  ELSE /* condition-replaceable (c) = TRUE */
13.        IF condition-dispensable (c) = TRUE
14.        THEN find-substitute-condition (c, c1)
                /* see Section 6.2.1 */
15.              IF found
16.              THEN replace-condition (c,c1)
17.              ELSE /* not found */
18.                    drop c from V
19.              END IF
20.              C = C - c
21.        ELSE /* condition-dispensable (c) = FALSE */
22.             find-substitute-condition (c, c1) /*
                    see Section 6.2.1 */
23.             IF found
24.             THEN replace-condition (c,c1)
25.                   C = C - c
26.             ELSE /* not found */
27.                  Success = FALSE
28.           END IF
29.        END IF
30.    END IF
31. END DO
```

(Algorithm 5)
Boolean PROCEDUREfind-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 9. 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:

```
CREATE   VIEW $V$ ($\mathcal{VE} = ''\supseteq''$)
SELECT   $A_2, B_1, B_2, C_2$
FROM     $R_1, R_2, R_3$
WHERE    $(A_1 = B_1)$ $(\mathcal{CD} = true, \mathcal{CR} = true)$
         AND $(A_1 = C_1)$ $(\mathcal{CD} = true, \mathcal{CR} = true)$.
```

Fig. 7a 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 Fig. 7b 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 Fig. 7c).

Let's assume that the information provider of R decides to delete $R.A_1$. Obviously, both of the primitive

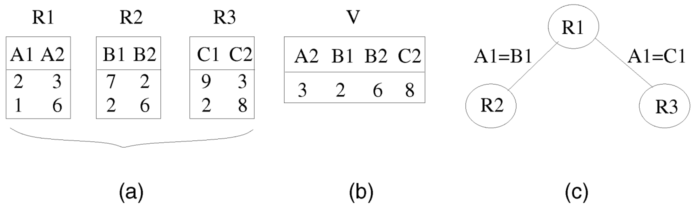

Fig. 7. Example data set.

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:

$$\begin{array}{ll} \text{CREATE} & \text{VIEW } V\ (\mathcal{VE} = {}''\supseteq'') \\ \text{SELECT} & A_2, B_1, B_2, C_2 \\ \text{FROM} & R_1, R_2, R_3. \end{array} \tag{33}$$

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 eight instead of one tuples (see Fig. 8a).

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 [41].

Example 10. 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 Fig. 8b, 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:

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

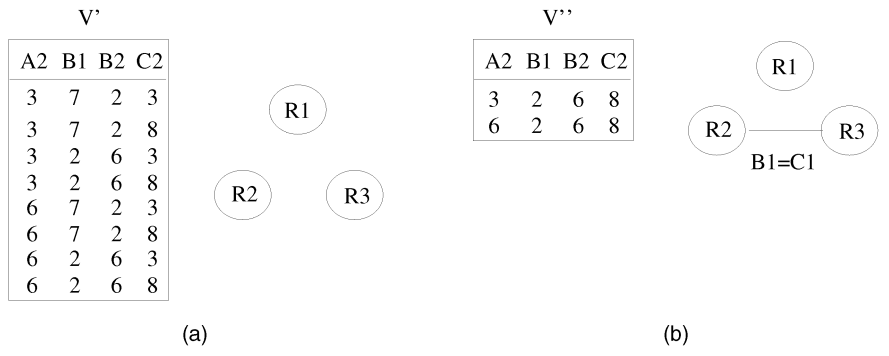

Fig. 8. Two alternative ways to evolve V. (a) Redefined view: V'. (b) Redefined view: V''

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 Fig. 8a versus 8b). 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).

```
1.  AC1 = affected-components (A) /* find components
               that reference A in V */
2.  Success = TRUE
3.  WHILE (AC1 != empty) AND (Success) DO
4.        get component from AC1
5.        IF ( dispensable (component) = FALSE
              AND replaceable (component) = FALSE )
6.      THEN Success = FALSE
7.      END IF
8.      AC1 = AC1 - component
9.  END DO
10. IF (Success) /* it is possible to evolve V */
11. THEN call VEA-delete-attribute (A,SELECT);
12.       IF (Success)
13.       THEN            /* use substitute for SELECT
                              component, if found */
14.           call VEA-delete-attribute'(A,WHERE);
15.       END IF
16. 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 schema 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).

```
01. tempSet = affected-components (VD,R)
             /* view components referring to
                r attrs(R) */
02. code = 2 /* code = 1, must find replacement;
                2, good if finds replacement */
03. WHILE ( tempSet != empty) AND ( code != 1) DO
         /* test for possibility of evolution */
04.   BEGIN /* WHILE */
05.     component = get-component(tempSet)
06.     IF ( dispensable (component) = FALSE AND
              replaceable (component) = FALSE ) THEN
07.           return failure with msg "VD cannot be evolved"
08.     ELSE IF ( dispensable (component) = FALSE ) THEN
09.       BEGIN
10.           code = 1 /* some view component is
                  indispensable, must find replacement */
11.           tempSet = tempSet - component
12.       END
13. END /* WHILE */
14. /* possible to evolve VD */
15. IF ( replaceable (R) = FALSE )
16.   THEN IF ( code = 1 )
17.           THEN return failure with msg
                      "VD cannot be evolved"
18.           ELSE drop affected-component (VD,R)
                      from VD
19.   ELSE /* replaceable (R) = TRUE */
20.     BEGIN
21.         found = find-substitute-relation (VD,R,S)
22.         IF (NOT found) THEN
23.           THEN IF ( code = 1 )
```

```
24.                THEN return failure with msg
                        "VD cannot be evolved"
25.                ELSE drop affected-component
                        (VD,R) from VD
26.        ELSE /* found */
27.            replace-relation (R,S)
28.  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).

```
01.  tempSet = affected-attr-components (VD,R)
             U affected-condition-component (VD,R)
02.  While (tempSet != empty) DO
03.        BEGIN /* WHILE */
04.            component = get-component (tempSet)
04.            IF substitute S.B for component exists IN S
05.                THEN replace component by S.B
06.                ELSE drop component from VD
07.              tempSet = tempSet - component
08.          END /* WHILE */
09.  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 schema changes of participating ISs. In [35], 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 [17], 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 and actual algorithms for achieving view synchronization for all basic schema change operations are the key contributions of this current work. We also develop as well as prove theorems on the correctness of the proposed replacement strategies. The synchronization algorithms we introduce here are based on containment constraints, while of course view synchronization can also be explored for other types of meta knowledge, such as functional dependencies or join constraints—then requiring new appropriate strategies [7]. In more recent work, we have also looked at a cost model for trading off the quality versus cost aspects of nonequivalent rewritings generated by view synchronization [12], [13]. While no one has addressed the topic of view synchronization as such, there are several issues we address for *EVE* that relate to work done before in other contexts as now described below.

Gupta et al. [9] and Mohania and Dong [24] 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.

Some work has been done on rewriting queries using materialized views [16], [20], [19], [38], [15], [40], [39]. This work is relevant to the EVE project, although it generally deals with rewriting queries into *equivalent ones* using underlying views. Cohen et al. [5] discuss the problem of rewriting aggregate queries using views.

Work on the *World View* concept by Levy et al. [21] is closely related to ours in terms of its goal of information integration, but not the approach taken. In [21], the notion of the *world-view* is introduced as a global, fixed domain model of a certain part of the world on which both information providers and consumers must define views. This work is in some sense an approach inverse to ours [35]. Where Levy et al. describe information sources in terms of a world model, we incrementally establish our world model in terms of the available sources. Levy's model provides a solution to a subset of problems that we also solve. It is however necessary to establish a world model before any source can provide information—a very complicated and often impossible task. Changes to the world model are not possible in this approach or would require a manual redefinition of both information providers' and consumers' queries. Another drawback of the approach is the insufficient handling of redundancy in the information space. If two information providers define partially overlapping view extents, Levy's algorithms find the *minimal cover* for the queried data, i.e., uses information from a randomly picked information source that satisfies the user's query. In contrast to this approach, we can make use of known overlaps of source data to provide nonequivalent rewrites of queries in the case of the possible unavailability of one of the sources. With the help of a quality measure (QC-Value [14]), we can also decide which of a number of given information sources provides the *best* answer to a query.

The DWQ (Data Warehouse Quality) Esprit Project [25], [11] addresses many problems related to the quality of data warehouses. In this context, they also investigate the issues of evolution of data warehouses. Quix [32] describes a process model for the capture of all changes made to any component of a data warehouse management system into a meta repository. Such changes may include the addition or removal of a view by the (human) data manager, Thus, like EVE, they make use of a meta repository in support of data warehouse evolution. However, their focus is on the methodology and management of the process of the meta repository design to assure quality of a data warehouse, while our particular problem of generating nonequivalent view rewritings over evolving warehouses and establishing preference models for evolution have not been addressed in the DWQ project.

Papakonstantinou et al. [31], [30] 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 meta knowledge model, their data model allows information sources to describe their capabilities (including their schema properties), but they don't assume that these capabilities could be changed and, thus, they do not address the view synchronization problem. The same author has also done work on query rewriting without using views, for example, in *capabilities-based query rewriting* for mediator-based systems [23], in which a query (or multiple queries) are formulated based on query capabilities of underlying sources. Florescu et al. [8] have worked on a similar problem in multidatabase systems with an ODMG-based meta model.

The *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, like the SIMS [1] and SoftBot [7] projects. In the SIMS project, a unified schema is a priori 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. The two projects do not address the particular problem of evolution under schema changes of participating information sources.

CoBase by Chu et al. [4], [6] relates to our work in that they also use the notion of relaxation of the *query extent*, similar to our E-SQL approach [35]. Chu et al. established an SQL, extension called CSQL (cooperative SQL), which relaxes the strictness of SQL-*where*-conditions, i.e., it relaxes restrictions on the extent, but not the interface of a view query, as E-SQL does. Given explictly available knowledge about an application's domain, queries can be relaxed in a stepwise manner by altering local WHERE-conditions of a query until it returns approximate results to a user. Chu et al.'s work differs from ours in that it is limited to relaxing the values of local conditions in queries, whereas we handle relaxation of all elements in a Project-Select-Join-SQL-query. In contrast to CSQL, in which a manually established order of relaxation of conditions is needed to compare two rewriting possibilities, we have also defined a comprehensive model of quality and cost to automatically assess the desirability of a query rewriting [13], [14] (of which our algorithms would normally generate several) in order to help a view synchronization algorithm to find tradeoffs among query rewritings.

Lakshmanan et al. [22] discuss an SQL extension called *SchemaSQL*. SchemaSQL can query not only the data of a relational database but also the schema such as sets of attribute and relation names, and can treat such sets of meta data analogously to and simultaneously with regular data within one query. This language then can be used by a database designer to describe schema transformations between diverging relational schemas. The automatic generation of query restructurings (a la view synchronization) or preference models for evolution are not within the scope of query language design per se and, thus, are not considered in the SchemaSQL work. SchemaSQL and E-SQL are complimentary, and extensions of our E-SQL preference model to now also work for SchemaSQL (that is, meta data) queries would be one among several possibly interesting future works.

In an earlier project on transparent schema evolution (TSE) technology [36], [37], we had explored a solution to a different yet related evolution problem, 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 [27], [28], 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 schema 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 [10]. 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 [28]. 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 the *EVE* System

A prototype of the *EVE* system has been implemented by members of the Database Systems Research Group (DSRG) 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 the participating ISs are built on top of Oracle and Microsoft Access. The communication between *EVE* and the information space is via JDBC. The view synchronization algorithms for the different basic schema changes presented in Section 7 have been fully implemented. An online EVE demonstration can found at the DSRG web site at http://davis.wpi.edu/dsrg. The EVE system has also been formally demonstrated at ACM SIGMOD'99 [33].

9.2 Contributions

Our effort is one of the first works to study the new problem of view definition adaptation in dynamic environments. This problem, which we call *view synchronization*, corresponds to the process of adapting view definitions triggered by schema 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 schema. 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 the feasibility of the proposed ideas, and its demonstration at ACM SIGMOD'99).

9.3 Future Directions

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 including Web-based information services, electronic catalog providers, etc.

In a recent article in the Communications of ACM [34], we lay out a large array of possible future tasks to spawn in this area. In fact, we have already embarked on attempting to tackle some of these open issues, including models for capturing the quality as well as the cost of nonequivalent rewritings produced by view synchronization algorithms [13] as well as algorithms for view maintenance of the view extent under both schema and data changes of the sources [44], [45].

APPENDIX A

PROOF FOR THEOREM 1

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

We have to prove that for $\phi \in \{''\equiv'', ''\subseteq'' v\}$ 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, $R_1, \dots, R_n$ 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. $t_S, t'_R, t_1, \dots t_n$ derive t' in V',
5. $\mathcal{CV}'(t_S[S.B], t'_R, t_1, \dots t_n)$[6] 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 (18) of Condition 3. That is,
$$\exists g \in \left(\pi_{((Attr(V)\cap Attr(R))\setminus\{R.A\})\cup\{S.B\}}(R \bowtie_{\mathcal{C}(\bar{J})} S)\right)$$
   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,

6. Even so, the conjuction of primitive clauses $\mathcal{CV}'$ is defined on a subset of attributes (i.e., $((\bar{W}\{R.A\}) \cup \{S.B\}))$ of the tuples $t_S, t'_R, \dots t_n$, we use this notation to 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 $((\bar{W}\{R.A\}) \cup \{S.B\})$, that is $S.B$.

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 $V' \subseteq V$.

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., $V' \supseteq V$. 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 $1 \le i \le n$, $\exists t_i \in R_i$, 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 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. $t'[S.B] = t_S[S.B] \overset{(6)}{=} t_R[R.A] \overset{(1)}{=} t[R.A]$,
11. $t'[R.\bar{D}] = t'_R[R.\bar{D}] \overset{(7)}{=} t_R[R.\bar{D}] \overset{(1)}{=} t[R.\bar{D}]$,
12. for all $1 \le i \le n$, $t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i] \overset{(2)}{=} t[R_i.\bar{D}_i]$.

Hence, we can conclude that $t' = t$. 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 $\phi = {}''\equiv''$.[7] Similarly, the inclusion (II) is implied by Case 2 with $\phi = {}''\equiv''$. Thus, we conclude that $V \equiv V'$.□

APPENDIX B

PROOF FOR THEOREM 2

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 (for common attributes). That is, $V' \subseteq_\pi V$.[8] 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

1. $\mathcal{CV}'((\bar{W} \setminus R.\bar{D}') \cup S.\bar{F}')$ has the same set of primitive clauses as $\mathcal{CV}(\bar{W})$[9] WHERE clause are dropped in V' then in general we cannot prove $V' \subseteq_\pi V$. 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:
2. $\exists t_S \in S$, such that $t'[S.\bar{F}] = t_S[S.\bar{F}]$,
3. for all $1 \le i \le n$, $\exists t_i \in R_i$, such that

$$t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i],$$

4. $t_S, t_1, \ldots t_n$ derive t' in V',

7. Cases 1 and 2 are more general cases proven for $\Phi \in \{''\equiv'', ''\subseteq''\}$ and $\{''\equiv'', ''\supseteq''\}$, respectively.
8. $\subseteq_\pi$ is defined in []
9. If some of the clauses in the WHERE clause are dropped in V', then, in general, we cannot prove $V' \subseteq_\pi V$.

5. $\mathcal{CV}'(t_S[S.\bar{F}'], t_1, \ldots t_n)$ 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 (25). That is,
6. $S.\bar{F}, S.\bar{F}' \subseteq S.\bar{B}$.
   Then, from (25) (with

$$\phi \in \{''\equiv'', ''\subseteq''\})$$

   we have that there exists a tuple $t_R \in R$ such that,
7. $t_S[S.\bar{B}] = t_R[R.\bar{A}]$.
   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 (1), (6), and (7) we have that,
8. $t_S[S.\bar{F}'] = t_R[R.\bar{D}']$[10] 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 (1),(5), and (8) we have
9. $\mathcal{CV}(t_R[R.\bar{D}'], t_1, \ldots t_n)$ is satisfied.[11]
   Then, from (9), we can deduce that,
10. 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 (3) and (10), we have that,
11. for all $1 \le i \le n$, $(t_i \in R_i)$,

$$t'[R_i.\bar{D}_i] \overset{(2)}{=} t_i[R_i.\bar{D}_i] \overset{(9)}{=} t[R_i.\bar{D}_i].$$

   From (2) and (8) we have that
12. $t'[S.\bar{F}] \overset{(1)}{=} t_S[S.\bar{F}] \overset{(7)}{=}_\pi t_R[R.\bar{D}] \overset{(9)}{=} t[R.\bar{D}]$.
   In (11) and (12), 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 $1 \le i \le n$, $\exists t_i \in R_i$, 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.
   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. $\mathcal{CV}'(t_S[S.\bar{F}'], t_1, \ldots, t_n)$[12] 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 (2), (3), (4), (7), and (8), 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 $1 \le i \le n$, $t'[R_i.\bar{D}_i] = t_i[R_i.\bar{D}_i] \overset{(2)}{=} t[R_i.\bar{D}_i]$.
   Hence, we can conclude that $t' =_\pi t$. Since, t was chosen arbitrary from V, we have proven that $V' \supseteq_\pi V$.

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

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 (1).

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 (1) is satisfied. □

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 proofread the paper and for implementing components of the EVE system. This work was supported in part by the US National Science Foundation NYI grant #IRI 94-57609. The authors would also like to thank their industrial sponsors, in particular, IBM and Informix.

REFERENCES

[1] Y. Arens, C.A. Knoblock, and W.-M. Shen, "Query Reformulation for Dynamic Information Integration," *J. Intelligent Information Systems,* vol. 6, nos. 2/3, pp. 99-130, 1996.

[2] J.A. Blakeley, N. Coburn, and P.-A. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Trans. Database Systems,* vol. 14, no. 3, pp. 369-400, Sept. 1989.

[3] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J.H. Williams, J. Thomas, and E.L. Wimmers, "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach," *Proc. Fifth Int'l Workshop Research Issues in Data Eng. (RIDE): Distributed Object Management,* 1995.

[4] W.W. Chu, M.A. Merzbacher, and L. Berkovich, "The Design and Implementation of CoBase," *SIGMOD Record,* vol. 22, no. 2, pp. 517-522, June 1993.

[5] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting Aggregate Queries Using Views," *Proc. ACM Symp. Principles of Database Systems,* C. Papadimitriou, ed., May 1999.

[6] W.W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson, "CoBase: A Scalable and Extensible Cooperative Information System," *Intelligent Information Systems (JIIS),* vol. 6, nos. 2/3, pp. 223-259, 1996.

10. From (1), we have that $\mid S.\bar{F}' \mid = \mid R.\bar{D}' \mid$.

11. HERE IS THE PLACE WHERE WITHOUT (1) we cannot prove $V' \subseteq V$.

12. 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.

[7] O. Etzioni and D. Weld, "A Softbot-Based Interface to the Internet," *Comm. ACM,* vol. 37, no. 7, pp. 72-76, July 1994.

[8] D. Florescu, L. Raschid, and P. Valduriez, "Using Heterogenous Equivalence for Query Rewriting in Multidatabase Systems," *Proc. Third Int'l Conf. Cooperative Information Systems,* 1995.

[9] A. Gupta, H.V. Jagadish, and I.S. Mumick, "Data Integration Using Self-Maintainable Views," *Proc. Int'l Conf. Extending Database Technology (EDBT),* pp. 140-144, 1996.

[10] S. Galindo-Legaria, "Outerjoins as Disjunctions," *Proc. SIGMOD,* pp. 348-358, 1994.

[11] M. Jarke, M.A. Jeusfeld, C. Quix, and P. Vassil-iadis, "Architecture and Quality in Data Warehouses: An Extended Repository Approach," *Information Systems,* vol. 24, no. 3, pp. 229-253, 1999.

[12] A.J. Lee, A. Koeller, A. Nica, and E.A. Rundensteiner, "Data Warehouse Evolution: Trade-Offs between Quality and Cost of Query Rewritings," Technical Report WPI-CS-TR-98-2, revised in 1999., Worcester Polytechnic Inst., Dept. of Computer Science, 1998.

[13] A.J. Lee, A. Koeller, A. Nica, and E.A. Rundensteiner, "Data Warehouse Evolution: Trade-Offs between Quality and Cost of Query Rewritings," *Proc. IEEE Int'l Conf. Data Eng.,* Poster Session p. 255, Mar. 1999.

[14] A.J. Lee, A. Koeller, A. Nica, and E.A. Rundensteiner, "Non-Equivalent Query Rewritings," *Proc. Int'l Database Conf.,* pp. 248-262, July 1999.

[15] A.Y. Levy, A.O. Mendelzon, and Y. Sagiv, "Answering Queries Using Views," *Proc. ACM Symp. Principles of Database Systems,* pp. 95-104, May 1995.

[16] A. Levy, I.S. Mumick, Y. Sagiv, and O. Shmueli, "Equivalence, Query Reachability and Satisfiability in Datalog Extensions," *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 109-122, May 1993.

[17] A.J. Lee, A. Nica, and E.A. Rundensteiner, "Keeping Virtual Information Resources Up and Running," *Proc. IBM Centre for Advanced Studies Conf. (CASCON '97),* pp. 1-14, Nov. 1997. (Best paper award.)

[18] A.J. Lee, A. Nica, and E.A. Rundensteiner, "The EVE Framework: View Synchronization in Evolving Environments," Technical Report WPI-CS-TR-97-4, Worcester Polytechnic Inst. Dept. of Computer Science, 1997.

[19] A.Y. Levy, A. Rajaraman, and J.D. Ullman, "Answering Queries Using Limited External Processors," *Proc. 15th ACM Symp. Principals of Database Systems (pods),* pp. 227-237, June 1996.

[20] A. Levy and Y. Sagiv, "Constraints and Redundancy in Datalog," *Proc. 11th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 67-80, June 1992.

[21] A.Y. Levy, D. Srivastava, and T. Kirk, "Data Model and Query Evaluation in Global Information Systems," *J. Intelligent Information Systems,* special issue on Networked Information Discovery and Retrieval, vol. 5, no. 2, pp. 121-143, 1995.

[22] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "SchemaSQL—A Language for Interoperability in Relational Multi-Database Systems," *Proc. 22nd Int'l Conf. Very Large Data Bases,* T.M. Vijayaraman et al., eds., pp. 239-250, Sept. 1996.

[23] C. Li, R. Yerneni, V. Vassalos, H. García-Molina, Y. Papakonstantinou, J.D. Ullman, and M. Valiveti, "Capability Based Mediation in TSIMMIS," *Proc. SIGMOD,* pp. 564-566, 1998.

[24] M. Mohania and G. Dong, "Algorithms for Adapting Materialized Views in Data Warehouses," *Proc. Int'l Symp. Cooperative Database Systems for Advanced Applications,* Dec. 1996.

[25] Y. Vassiliou and M. Jarke, "Data Warehouse Quality: A Review of the DWQ Project," *Proc. Second Conf. Information Quality,* 1997.

[26] A. Nica, A.J. Lee, and E.A. Rundensteiner, "The Complex Substitution Algorithm for View Synchronization," Technical Report WPI-CS-TR-97-8, Worcester Polytechnic Inst. Dept. of Computer Science 1997.

[27] A. Nica, A.J. Lee, and E.A. Rundensteiner, "The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems," *Proc. Int'l Conf. Extending Database Technology (EDBT '98),* pp. 359-373, Mar. 1998.

[28] A. Nica and E.A. Rundensteiner, "Loosely-Specified Query Processing in Large-Scale Information Systems," *Int'l J. Cooperative Information Systems,* vol. 7, no. 1, pp. 77-104, 1998.

[29] A. Nica and E.A. Rundensteiner, "On Translating Loosely-Specified Queries into Executable Plans in Large-Scale Information Systems," *Proc. Second IFCIS Int'l Conf. Cooperative Information Systems CoopIS '97,* pp. 213-222, June 1997.

[30] Y. Papakonstantinou, H. García-Molina, and J. Ullman, "Medmaker: A Mediation System Based on Declarative Specifications," *Proc. IEEE Int'l Conf. Data Eng.,* 1996.

[31] Y. Papakonstantinou, H. García-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 251-260, Mar. 1995.

[32] C. Quix, "Repository Support for Data Warehouse Evolution," *Proc. Int'l Workshop Design and Management of Data Warehouses (DMDW '99),* pp. 4.1-4.9, June 1999.

[33] E.A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li, "Evolvable View Environment," *Proc. SIGMOD'99 Demo Session,* pp. 553-555, May/June 1999.

[34] E.A. Rundensteiner, A. Koeller, and X. Zhang, "Maintaining Data Warehouses over Changing Information Sources," *Comm. ACM,* vol. 43, no. 6, pp. 57-62, June 2000.

[35] E.A. Rundensteiner, A.J. Lee, and A. Nica, "On Preserving Views in Evolving Environments," *Proc. Fourth Int'l Workshop Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information,* pp. 13.1-13.11, Aug. 1997.

[36] Y.G. Ra and E.A. Rundensteiner, "A Transparent OO Schema Change Approach Using View Schema Evolution," *IEEE Int'l Conf. Data Eng.,* pp. 165-172, Mar. 1995.

[37] Y.G. Ra and E.A. Rundensteiner, "A Transparent Schema-Evolution System Based on Object-Oriented View Technology," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 4, July/Aug. 1998.

[38] A. Rajaraman, Y. Sagiv, and J.D. Ullman, "Answering Queries Using Templates With Binding Patterns," *Proc. ACM Symp. Principles of Database Systems,* pp. 105-112, May 1995.

[39] A. Rajaraman and J.D. Ullman, "Integrating Information by Outerjoins and Full Disjunctions," *Proc. ACM Symp. Principles of Database Systems,* pp. 238-248, 1996.

[40] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy, "Answering Queries with Aggregation Using Views," *Proc. Int'l Conf. Very Large Data Bases,* pp. 318-329, 1996.

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

[42] J. Widom, "Research Problems in Data Warehousing," *Proc. Int'l Conf. Information and Knowledge Management,* pp. 25-30, Nov. 1995.

[43] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment," *Proc. SIGMOD,* pp. 316-327, May 1995.

[44] X. Zhang and E.A. Rundensteiner, "Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework" *Information Systems,* vol. 27, no. 4, pp. 219-243, 2002.

[45] X. Zhang and E.A. Rundensteiner, "DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment," *Data Warehousing and Knowledge Discovery,* Sept. 2000.

Amy Lee the received BA degree in economics from the National Chung-Hsing University, Taipei, Taiwan, the MA degree in economics and the MS degree in computer science from University of Minnesota, Minneapolis, and the PhD degree from the University of Michigan, Ann Arbor. She was an assistant professor in the Department of Information and Systems Management at the Hong Kong University of Science and Technology. Her current research interests include data warehousing over distributed information sources, web information systems, and knowledge management. Dr. Lee is currently a systems developer/engineer with the Center for Human Resource Research, Columbus, Ohio.

Anisoara Nica received the MS and PhD degrees in electrical engineering and computer science from University of Michigan, Ann Arbor, in 1997 and 1999, respectively. Since 1998, she has been a part of the Research and Development team at the iAnywhere Solutions, a subsidiary of Sybase, Inc. Her research interests include query processing, query optimization, and mobile and wireless computing.

Elke Rundensteiner received the BS degree (Vordiplom) from the J.W. Goethe University, Frankfurt, West Germany, the MS degree from Florida State University, and the PhD degree from the University of California, Irvine, all in computer science. She is currently an associate professor in the Department of Computer Science at the Worcester Polytechnic Institute, after having been a faculty member at the University of Michigan, Ann Arbor. Dr. Rundensteiner has been active in the database research community for more than 15 years. Her current research interests include database evolution and migration, web data management, data warehousing for distributed systems, and information integration, exploration and visualization. She has published more than 100 publications in these and related areas. Her research has been funded by government agencies including US National Science Foundation, ARPA, NASA, CRA, DOT, and by industry including IBM, Verizon Labs, GTE, AT&T, Intel, Informix, and GE. Dr. Rundensteiner has received numerous honors, including Fulbright, US National Science Foundation Young Investigator, and IBM Partnership Award.