WPI-CS-TR-00-01 January 2000

A History-Driven Approach at Evolving Views Under Meta Data Changes

by

Andreas Koeller
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

A History-Driven Approach at Evolving Views Under Meta Data Changes

Andreas Koeller and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609–2280
{koeller|rundenst}@cs.wpi.edu
Phone: +1–508–831–5857
Fax: +1–508–831–5815

Abstract

Views over distributed information sources rely on the stability of the schemas of underlying databases. In the event of meta data changes in the sources, such as the deletion of a table or column, such views may become undefined. Using meta data about information redundancy, views can be evolved as necessary to be defined on a source schema after a source meta data change. This mechanism provides multiple nonequivalent rewritings for each meta data change that may differ in their view interfaces and/or extents from the original query.

Previous work in view synchronization focused only on deletions of schema elements. We now offer the first approach that makes use of additions also, partly by returning view definitions to previous versions by "backtracking" in the history of views and meta data. This technology enables us to adapt views to temporary meta data changes by cancelling out opposite changes, allows undo/redo-operations on meta data and can therefore improve the quality of evolved views. Also, we can accommodate additions to the information space that do not relate to previous deletes by making use of relationships between in formation sources and elements of the view. Thus, we are providing a complete approach at evolving data and meta data in an information space and rewriting views after those changes in a way that will keep them as close as possible to the original view's intent. One key property of this synchronization process is that it is accomplished without the need for a global invariate domain model (world view) which is unlikely to be available in practice.

Contents

1 Introduction **3**

2 Background: The EVE-System **5**

2.1 Evolvable SQL and the Model for Information Source Description 5

2.2 View Synchronization 6

2.3 Ranking Query Rewritings 8

3 Modeling Meta Data and Meta Data Changes **8**

3.1 Meta Data Model 8

3.2 Changes in Meta Data 10

3.3 Evolution of the MKB under Meta Data Changes 11

3.4 Dependency of Views on Meta Data 11

3.5 Modeling the History of Meta Data and Views 12

4 The History-Driven View Synchronization Algorithm (HD-VS) **13**

4.1 Affected Views 15

4.2 Backtracking in the History of a View 16

4.3 Re-applying a Meta Data Update Sequence 17

4.4 Reconstructing the History Graph 18

4.5 Backtracking to the Beginning 18

4.6 Meta Data Change Operations in Detail 19

4.6.1 *delete*-Operations 20

4.6.2 *add*-Operations 20

5 Evaluation **21**

5.1 Correctness 21

5.2 Improving Views after Meta Data Changes 23

6 Implementation **25**

7 Related Work **25**

8 Conclusion **28**

1 Introduction

In recent years, the number of digital information storage and retrieval systems has increased immensely. Many of those database systems are available in some kind of network, and the task of integrating data from different source databases is an increasingly important one. Solutions to this problem include data warehouses, a term which in the database community refers to materialized views over distributed information sources. Numerous publications deal with problems of maintenance of such views under data updates of the underlying information sources [BLT86, GM95, ZGMW96].

Newer developments, such as the ubiquity of the World Wide Web, emphasize the independence of data producers from data consumers. Independent data producers or providers have control over the capabilities (schema) of their information sources which raises the question of the influence of *meta data updates* (deletions of attributes or relations in underlying databases) on a view. In traditional views (as introduced in the literature, e.g., [BLT86, ZGMW96, AASY97]), meta data updates can render a view definition undefined.

Two general approaches that can address this problem have been presented in the literature. One approach taken by Levy et al. [LSK95] as well as Arens et al. [AKS96] is to create 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. The inverse approach, used in this paper, 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 built in the traditional way over a number of base schemas and those views are adapted to base schema changes by rewriting them using information space redundancy and relaxable view queries [RLN97]. The benefit of this approach is that no pre-defined 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).

On the latter approach, some work has been done in the *EVE*-Project [RLN97, NLR98, NR98b, KRH98] that deals with the rewriting of views under schema updates in the underlying sources in the case of *schema element deletions*. In EVE, algorithms have been defined that can rewrite a view under meta data changes of underlying sources and retain all or a part of the view extent in the new rewritten view. The notion of non-equivalence of view rewritings has been defined [LNR97], and a model for a numeric assessment of the quality and cost of such rewritings has been presented [LKNR99b].

One drawback of previous work is the finality of a view synchronization operation. After the deletion

of an underlying relation that has been used by a view, the view is rewritten to not refer to that relation any more. Even if the same relation is later added back to the information space (for instance, after a temporary unavailability due to a network problem), it will never be used by the view again since without a global domain model the view synchronization algorithm cannot determine in what relationship a new data element stands to other previously available elements.

In this paper, we present a complete approach capable of handling all information source schema changes, namely *adds*, *renames*, and *deletes* of attributes and relations. Also, we rewrite views as necessary under changes of *constraints* across the source databases (such as a *containment relationship* defining that `IS1.Hotels` contains `IS2.BostonHotels`). We show how to adapt view definitions to such changes, which we refer to as *meta data updates*. The main contribution of this paper is the utilization of additional available meta data to keep views as close to their original definition as possible, under a sequence of meta data changes that occurs over time. It turns out that the addition of attributes, relations and other knowledge about the information sources can indeed be beneficial to a view synchronization environment and leads to improved view definitions in a large number of cases.

Note that, with all view synchronization algorithms that have been devised so far, only one-step view synchronization is possible (cf. Section 2.2). The POC/SPOC [NR98b], SVS [RLN97], CVS [NLR98], and GRASP [NR98b] algorithms can not synchronize a view under a sequence of schema changes. Rather, they generate a new view definition after each schema change in an information source, without taking any information about earlier IS changes into account. Also, they do not support any meaningful synchronization under $add-$capability-changes. For example, when an information source IS_1 becomes unavailable to the data warehouse, data is either completely removed or at best replaced from a different IS_2. When IS_1 then joins the information space again, it will never be used again in any view. This work describes a way to overcome this problem.

Applications of this technology, for example, lie in views over several independent information sources (which could be WWW sources with SQL-Wrappers) in which the meta data of the underlying information may evolve or in which attributes or relations may be temporarily unavailable. Also, the usefulness of materialized views to a user could be increased when improved or additional underlying information becomes available.

The rest of the paper is organized as follows: Section 2 reviews some background in view synchronization, Section 3 defines the Meta Knowledge Base and the notion of history, Section 4 gives the main History-Driven View Synchronization Algorithm, and Section 5 gives proofs of correctness and demonstrates the improvements achieved by our approach. Section 7 reviews related work and Section 8 concludes the paper.

2 Background: The EVE-System

Since our work bases on the *EVE*-Project [RLN97], we give a very short overview over those elements of the EVE-System that we need as background in this paper. The algorithm described in this paper builds on top of the *one-step* synchronization algorithms described in the *EVE*-Project [RLN97, NLR98, NR98b, KRH98]. As we will see, the new algorithm in some cases falls back on these old algorithms and performs one or a sequence of single-step view synchronizations.

2.1 Evolvable SQL and the Model for Information Source Description

E-SQL or *Evolvable-SQL* is an extension of SQL that allows the view definer to express preferences for view evolution [LNR97]. A user defining a view can specify what information is indispensable, what information is replaceable by similar information from other information sources (ISs), and whether a changing view extent is acceptable. This is the key to obtaining non-equivalent but useful query rewritings as E-SQL provides the *EVE* system with the flexibility to evolve a view under meta data changes in a controlled way while preserving the user's intended semantics. The *view-extent* parameter (cf. Example 1) $\mathcal{VE} \in \{\subseteq, \supseteq, \equiv, \approx\}$ expresses the relationship between the original and rewritten query as required by the view definer. For instance, $\mathcal{VE}$ ="$\supseteq$" requires any query rewriting V_i for the current view V to compute a superset of the original view extent (i.e., $V_i \supseteq V$). The value $\mathcal{VE}$ ="$\approx$" means no restrictions for the extent are given. Also, for each element in the view definition's SELECT-, FROM- and WHERE-clause, respectively, two boolean values determine whether that view element is (1) *dispensable* from the view definition and/or (2) *replaceable* with a meaningful alternative from the information space. Those parameters are named *attribute-dispensable (AD), attribute-replaceable (AR), relation-dispensable (RD), relation-replaceable (RR), condition-dispensable (CD), and condition-replaceable (CR).* For a full description of the E-SQL language, the reader is referred to [LNR97].

Example 1 *A typical E-SQL query would be:*

CREATE VIEW	Asia-Customer ($\mathcal{VE}$ = "$\supseteq$") AS
SELECT	Name, Address, Phone ($\mathcal{AD} = true$, $\mathcal{AR} = true$)
FROM	Customer C ($\mathcal{RR} = true$), FlightRes F
WHERE	(C.Name = F.PName) AND (F.Dest = 'Asia') ($\mathcal{CD} = true$)

The semantics of this query are as follows: Any rewriting of the view query is acceptable as long as the new view extent is a superset of the old one (expressed by $\mathcal{VE}$ = "$\supseteq$"); the attribute `Phone` *is dispensable (expressed by $\mathcal{AD} = true$) and can also be replaced from another source ($\mathcal{AR} = true$); the relation* `Customer` *(but not* `FlightRes`*) can be replaced with another relation ($\mathcal{RR} = true$) and the user will still have use for the view even if the second WHERE-condition cannot be kept valid ($\mathcal{CD} = true$).*

In order to allow for *view synchronization* (i.e. the rewriting of views under schema changes), our system needs to identify view element replacements from other ISs. For this purpose, we express relationships between ISs through constraints (e.g., agreeing data types, functional dependencies between attributes, extent overlaps between relations, stored in the *Meta Knowledge Base (MKB)*). One important constraint is the *containment-constraint*. A containment 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.

Consider the containment constraint shown in Figure 2. This containment constraint shows that the projection on the *Holder* and *Age* attributes of relation *Insurance* forms a superset of the projection on attributes *Name, Age* of relation *BackBay* for all tuples in *Insurance* whose *Amount* is over $1,000,000$ and whose *Age* is under 50.

2.2 View Synchronization

While all of the view synchronization algorithms previously presented are compatible with the algorithm presented in this paper, we give an idea of one of the view synchronization algorithms developed in the context of the *EVE*-system [NR98b, NR98a] as an example. Once a view is defined, *EVE* view synchronization tracks schema changes in all ISs participating in this view and attempts to find replacements for missing view elements using the MKB and E-SQL. For example, after a *delete-relation* capability change on a relation R is detected, the algorithm traverses the information space in order to find possible replacements for those attributes of R that were used by the view V. Example 2 gives an idea of the POC algorithm for view synchronization [NR98b].

The common underlying properties for all previously defined view synchronization algorithms are:

- A synchronization occurs after each meta data change, without taking previous synchronization steps into account. The only input data is the current MKB, VKB, and the meta data change that occurred.
- If a view can be rewritten from being valid on the old information space into a valid view on the new information space, the the algorithm will do this rewrite.
- All *deletes* and *renames* of attributes and relations are handled, but not *additions* of meta data or changes of *constraints*.
- If a view is rewritten, it *always deteriorates*, i.e., the quality of a view to a user (i.e., the distance to a user's initial view specification) will remain constant or decrease with every view synchronization step, but never improve.

- If a meta data item is deleted, the algorithms cannot recognize its return. Temporary unavailability of data cannot be handled.

Example 2 gives an idea of the POC algorithm for view synchronization [NR98b].

Example 2 *We define an information space (Meta Knowledge Base) according to Figures 1 and 2. We consider the view* ***Customer-Passengers*** *in Figure 3 and show how to apply the POC algorithm and find a replacement under the "delete relation* ***Customer****" change.*

IS 1: Flight Information **Customer(Name,Address,PhoneNo,Age)** **FlightRes(PName,Airline,FlightNo,Source, Dest,Date)**
IS 2: Insurance Information **Insurance(Holder,Type,Amount,Age)** **PreferredCust(PrefName,PrefAddress, PrefPhone)**
IS 3: Tour Participant Information **Participant(Name,TourID,StartDate,Location)** **Tour(TourID,TourName,Type,NoDays)**

Figure 1: IS Content Descriptions for Example 2

$$\mathcal{CC}_{\text{Customer,Insurance}} = \left(\pi_{\text{Insurance.Holder, Insurance.Age}}(\sigma_{(Insurance.Amount>1{,}000{,}000)\ AND\ (Insurance.Age<50)}\text{Insurance})\right) \supseteq \pi_{\text{Customer.Name,Customer.Age}}\text{Customer}$$

Figure 2: Containment Constraint for Example 2

```
CREATE VIEW   Customer-Passengers AS
SELECT        C.Name, C.Age,
FROM          Customer C , FlightRes F,
WHERE         (C.Name = F.PName)
              AND (F.Dest = 'Asia')
              AND (C.Age < 20)
```

Figure 3: An E-SQL View.

```
CREATE VIEW   Customer-Passengers' AS
SELECT        I.Holder, I.Age,
FROM          Insurance I, FlightRes F
WHERE         (I.Holder = F.PName)
              AND (F.Dest = 'Asia')
              AND (I.Age < 20)
              AND (I.Amount > 1,000,000)
              AND (I.Age < 50)
```

Figure 4: A Possible Rewriting for a View.

The POC algorithm uses containment constraints in the MKB that connect the new relation to the remaining relations in the existing query. Here, we can replace the attribute ***Customer.Age*** *by the similar attribute* ***Insurance.Age*** *in relation* ***Insurance*** *and join the new table with* ***FlightRes*** *using the containment constraint from Figure 2. Then all view elements (i.e., attributes and where-clauses) that depend on the old relation are replaced by view elements using the new relation. A possible rewriting of the query in Figure 3 using this substitution is given by the query in Figure 4.*

2.3 Ranking Query Rewritings

Since view synchronization algorithms may generate many possible query rewritings, one needs to be selected as the new view definition. For this purpose we have developed the QC-Model [LKNR99b] to estimate the *quality* and *cost* of the rewritings. Each legal query rewriting will in general preserve a different amount (extent) and different types (interface) of information, which we refer to as the *quality* of the view. Also, each new view query will cause different *view maintenance costs*, since in general data will have to be collected from a different set of ISs. With these two dimensions, the QC-Model can *compare* different view queries with each other, even if they are not equivalent. This comparison is accomplished by assessing five different factors as outlined below (full algorithm in [LKNR99b]).

- **Quality Factors:** Quality refers to the *similarity* (vs. divergence) between an original view V and its rewriting $V^{(n)}$ and is expressed for an original view V by $Q(V^{(n)})$. The *Degree of Divergence in Terms of the View Interface* determines numerically how different the view interfaces of the two queries are. The *Degree of Divergence in Terms of the View Extent* is determined by the relative numbers of missing and additional tuples in the extent of a view rewriting (as compared to the extent of the original view).

- **Cost Factors:** Cost factors measure the (long-term) cost associated with future *incremental view maintenance* after the view has been rewritten and the extent has been updated. The factors are *Number of Messages* between data warehouse and information sources, *Number of Bytes Transferred* through the network, and *Number of I/Os* at the ISs. Cost factors are combined into a single value using tradeoff factors, and the combined cost value of a view rewriting is expressed by $C(V^{(n)})$.

Normalizing and then combining these factors yields the *QC-Value* for a given rewriting ($QC(V^{(n)})$), a real number between 0 (bad) and 1 (good) to assess the value of a given query rewriting for a particular user (in terms of that user's E-SQL query evolution preferences).

3 Modeling Meta Data and Meta Data Changes

3.1 Meta Data Model

In an evolvable data warehouse as introduced by [RLN97], two types of data stores exist to maintain data warehouses over changing information sources. Those two structures are the *View Knowledge Base (VKB)* and the *Meta Knowledge Base (MKB)*. While the VKB holds view definitions, the MKB records meta data

about the underlying information space [1]. In the *EVE*-Project, such information has been encoded in the form of several different constraints [NLR97], with schema of underlying relational sources being only implicitly defined through *Type Integrity Constraints*.

For the purpose of this paper, we will generalize the notion of an MKB constraint to mean not only the dependencies (containment constraints, functional dependencies), but also the schema of underlying information sources. In this sense, we will describe schemas of underlying information sources by Attribute Constraints and Relation Constraints.

Definition 1 (Meta Knowledge Base) *We assume a finite set (alphabet) Z as a basis for names of attributes and relations. A Meta Knowledge Base MKB is a 6-tuple ($\mathcal{RC}$, $\mathcal{AC}$, $\mathcal{CC}$, $\mathcal{CPC}$,* rel, con*) where*

- *$\mathcal{AC}$ is a finite set of attribute constraints $\{\mathcal{AC}_1, \ldots, \mathcal{AC}_m\}$ where $Name(\mathcal{AC}_i) = A_i$ for $1 \leq i \leq m$ and $A_i \in Z^*$ (defining the attribute name). An attribute defined by $\mathcal{AC}_i$ also has a domain $Dom(\mathcal{AC}_i)$ which could be different for each attribute. An $\mathcal{AC}$ is denoted by $\mathcal{AC}_i =$* name :`type` *(see Example 3). An $\mathcal{AC}$ cannot exist without a Relation Constraint ($\mathcal{RC}$) that it is associated with through the function* rel.

- *$\mathcal{RC}$ is a finite set of relation constraints $\{\mathcal{RC}_1, \ldots, \mathcal{RC}_n\}$ where $Name(\mathcal{RC}_i) = R_i$ for $1 \leq i \leq n$ and $R_i \in Z^*$ (defining the relation name; relation names must be unique).*

- *$\mathcal{CC}$ is a finite set of Containment Constraints $\{\mathcal{CC}_1, \ldots, \mathcal{CC}_p\}$ where $\mathcal{CC}_i = \pi_{A_1, A_2, \ldots, A_m}(R_i)\ \theta\ \pi_{B_1, B_2, \ldots, B_m}(S_i)$ for $1 \leq i \leq p$, $R_i, S_i \in Z^*$ (relation names[2]) and $\theta \in \{\subseteq, \equiv, \supseteq, \approx\}$. For example, $\mathcal{CC}_i = \pi_1(R) \subseteq \pi_2(S)$ states that the projection π_1 on (any instance of) relation R in an information source forms a subset of the projection π_2 of (any instance of) relation S. The projections π of either relation in the $\mathcal{CC}$ are not defined here, but in Containment Pair Constraints (similar to the definition of a the schema of a relation in $\mathcal{AC}$s)*

- *$\mathcal{CPC}$ is a finite set of Containment Pair Constraints $\{\mathcal{CPC}_1, \ldots, \mathcal{CPC}_q\}$ with $\mathcal{CPC}_i = (A, B)$ for $1 \leq i \leq q$, $A, B \in Z^*$ (attribute names). For example, $\mathcal{CPC}_i = (A, B)$ states that a pair of attribute names (A, B) appears in the projection list (π_1, π_2) of a $\mathcal{CC}$ with which $\mathcal{CPC}_i$ is associated through the function* con. *A $\mathcal{CPC}$ can not exist without a $\mathcal{CC}$ that it is associated with through this function.*

- rel: *$\mathcal{AC} \rightarrow \mathcal{RC}$ is a function that associates each attribute constraint with exactly one relation constraint, thus defining the schema of each relation. Two attribute constraints defining the same attribute name*

[1] For a full description see [LNR97].

[2] Relations are not necessarily defined in any $\mathcal{RC}$! The containment could be defined over relations that are only accessible to the IS provider, but not to the data warehouse.

cannot be associated with the same relation constraint (attribute names in a relation schema must be unique).

- con: $\mathcal{CPC} \rightarrow \mathcal{CC}$ *is a function that associates each Containment Pair Constraint with exactly one Containment Constraint, thus defining the structure (the projection lists) of each Containment Constraint.*

Example 3 *Let us assume a travel agency with a national customer database. The Massachusetts branch of our travel agency maintains its own customer database for performance reasons, which is a subset of the national database. Also, the Boston office has a local copy of customers located in the Boston area, which is also a subset of the national database. Another information source is a flight reservation database. The MKB for this setup contains:*

$\mathcal{RC}_1$ = Boston, $\mathcal{RC}_2$ = MassBranch, $\mathcal{RC}_3$ = Customer, $\mathcal{RC}_4$ = FlightRes,
$\mathcal{AC}_1$ = Name :`string(50)`, $\mathcal{AC}_2$ = Address :`string(100)`, $\mathcal{AC}_3$ = City :`string(30)`, $\mathcal{AC}_4$ = State :`string(2)`,
$\mathcal{AC}_5$ = Phone :`string(15)`, $\mathcal{AC}_6$ = Name :`string(50)`, $\mathcal{AC}_7$ = Address :`string(100)`,
$\mathcal{AC}_8$ = City :`string(30)`, $\mathcal{AC}_9$ = Phone :`string(15)`, $\mathcal{AC}_{10}$ = Name :`string(50)`,
$\mathcal{AC}_{11}$ = Address :`string(100)`, $\mathcal{AC}_{12}$ = Phone :`string(15)`, $\mathcal{AC}_{13}$ = PName :`string(50)`,
$\mathcal{AC}_{14}$ = Dest :`string(30)`, $\mathcal{AC}_{15}$ = StartDate :`date`,
$\mathcal{CC}_1 = \pi_1(\text{Boston}) \subseteq \pi_2(\text{MassBranch})$, $\mathcal{CC}_2 = \pi_1(\text{Boston}) \subseteq \pi_2(\text{Customer})$,
$\mathcal{CPC}_1 = (\text{Name}, \text{Name})$, $\mathcal{CPC}_2 = (\text{Address}, \text{Address})$, $\mathcal{CPC}_3 = (\text{Phone}, \text{Phone})$,
$\mathcal{CPC}_4 = (\text{Address}, \text{Address})$, $\mathcal{CPC}_5 = (\text{Phone}, \text{Phone})$, $\mathcal{CPC}_6 = (\text{Name}, \text{Name})$,
rel = $\{\mathcal{AC}_1 \rightarrow \mathcal{RC}_3, \mathcal{AC}_2 \rightarrow \mathcal{RC}_3, \mathcal{AC}_3 \rightarrow \mathcal{RC}_3, \mathcal{AC}_4 \rightarrow \mathcal{RC}_3, \mathcal{AC}_5 \rightarrow \mathcal{RC}_3, \mathcal{AC}_6 \rightarrow \mathcal{RC}_2, \mathcal{AC}7_{\rightarrow}\mathcal{RC}_2, \mathcal{AC}_8 \rightarrow \mathcal{RC}_2, \mathcal{AC}_9 \rightarrow \mathcal{RC}_2, \mathcal{AC}_{10} \rightarrow \mathcal{RC}_1, \mathcal{AC}_{11} \rightarrow \mathcal{RC}_1, \mathcal{AC}_{12} \rightarrow \mathcal{RC}_1, \mathcal{AC}_{13} \rightarrow \mathcal{RC}_4, \mathcal{AC}_{14} \rightarrow \mathcal{RC}_4, \mathcal{AC}_{15} \rightarrow \mathcal{RC}_4\}$,
con = $\{\mathcal{CPC}_1 \rightarrow \mathcal{CC}_1, \mathcal{CPC}_2 \rightarrow \mathcal{CC}_1, \mathcal{CPC}_3 \rightarrow \mathcal{CC}_1, \mathcal{CPC}_4 \rightarrow \mathcal{CC}_2, \mathcal{CPC}_5 \rightarrow \mathcal{CC}_2, \mathcal{CPC}_6 \rightarrow \mathcal{CC}_2\}$.

3.2 Changes in Meta Data

Changes to the schema of underlying sources which is modeled in the Meta Knowledge Base cause MKB and VKB changes. This paper describes how to synchronize view definitions with such *meta data changes* under consideration of previous meta data changes in the information space, and how to make use of added meta knowledge by "re-installing" a previously deleted schema element. In order to better specify the meaning of "previously deleted element", we will introduce the concept of *history* of view and meta knowledge to our system.

Definition 2 (meta data change) *Let $\mathcal{C}_i$ be a constraint ($\mathcal{AC}$, $\mathcal{RC}$, $\mathcal{CC}$, or $\mathcal{CPC}$, cf. Definition 1). A* meta data change *$\mathcal{MDC}_i$ is the addition or deletion of $\mathcal{C}_i$ to or from the Meta Knowledge Base, transforming the MKB from a previous state MKB_i to a subsequent state MKB_{i+1}. Valid meta data changes are:* delete-$\mathcal{CPC}$ ($\mathcal{C}_i$), delete-$\mathcal{CC}$ ($\mathcal{C}_i$), delete-$\mathcal{RC}$ ($\mathcal{C}_i$), delete-$\mathcal{AC}$ ($\mathcal{C}_i$), add-$\mathcal{CPC}$ ($\mathcal{C}_i$), add-$\mathcal{CC}$ ($\mathcal{C}_i$), add-$\mathcal{RC}$ ($\mathcal{C}_i$), add-$\mathcal{AC}$ ($\mathcal{C}_i$). *Each $\mathcal{MDC}_i$ has as a parameter exactly one constraint in the MKB which is denoted by*

constraint($\mathcal{MDC}_i$) [3].

Definition 3 (inverse meta data change operations) *Two meta data change operations* $\mathcal{MDC}_1 \in \{\text{add-}\mathcal{CPC}, \text{add-}\mathcal{CC}, \text{add-}\mathcal{RC}, \text{add-}\mathcal{AC}\}$ *and* $\mathcal{MDC}_2 \in \{\text{delete-}\mathcal{CPC}, \text{delete-}\mathcal{CC}, \text{delete-}\mathcal{RC}, \text{ delete-}\mathcal{AC}\}$ *are inverse iff both refer to the same constraint, i.e.,* $\text{constraint}(\mathcal{MDC}_1) = \text{constraint}(\mathcal{MDC}_2)$. *We denote the inverse of a meta data change operation* $\mathcal{MDC}_i$ *by* $\mathcal{MDC}_i^{-1}$ *(e.g.* $\text{delete-}\mathcal{RC}(\mathcal{RC}_1) = (\text{add-}\mathcal{RC}(\mathcal{RC}_1))^{-1}$*).*

3.3 Evolution of the MKB under Meta Data Changes

We will now have a look at the effects of meta data changes on the MKB. For most of the eight operations, the respective constraint is simply added to or removed from the MKB. Some changes, however, imply other changes. The meta data change operations *delete-*$\mathcal{RC}$ and *delete-*$\mathcal{CC}$ trigger deletions of Attribute Constraints and Containment Pair Constraints, respectively. Those implicit operations are executed on the MKB immediately after the meta data change is detected, but before the view synchronization algorithm is run on the view definitions in the VKB. Similarly, *add-*$\mathcal{RC}$ and *add-*$\mathcal{CC}$-operations require subsequent additions of sub-constraints. This is done in the same fashion. In the case that the set of attributes in a relation has changed while this relation was unavailable, appropriate *add-*$\mathcal{AC}$ or *delete-*$\mathcal{AC}$ operations are triggered after the *add-*$\mathcal{RC}$ operation. Note that renames of relations or attributes cause all $\mathcal{RC}$ or $\mathcal{AC}$ in all MKB states to be renamed at once and do not constitute a $\mathcal{MDC}$ in the sense of Definition 2.

Example 4 *We will give an example of dependencies between meta data change operations. Assume the MKB of our running example. Consider the state of the MKB as defined in Example 3 (cf. node* N_0 *in Figure 5). We execute the operation* $\text{delete}-\mathcal{RC}_1(\text{Boston})$ *on that MKB. First, the set* $\mathcal{D}$ *will be constructed as the set of attributes in Relation Boston, i.e.,* $\mathcal{D} = \{\text{Name}, \text{Address}, \text{Phone}\}$. *This set is then subtracted from the set of attribute constraints, i.e.,* $\mathcal{AC}_{10}, \mathcal{AC}_{11}, \mathcal{AC}_{12}$ *are deleted, and* $\mathcal{RC}_1$ *is deleted from the set of relation constraints. A function* rel' *defined over the new set of attribute constraints is derived from* rel *as computing the same values as before for all* $\mathcal{AC}$*s except the deleted ones (for which it is undefined).*

3.4 Dependency of Views on Meta Data

In previous view synchronization algorithms [RLN97, NLR98, KRH98, NR98b] the rewriting of a view is accomplished with the help of constraints that state (partial) redundancy in the information space. For instance, a Containment Constraint $\mathcal{CC} = \pi_1(R) \subseteq \pi_2(S)$ specifying that relation R is a subset of relation S can be used to replace relation R by relation S in a view if the former becomes unavailable. The meaning of that constraint is that the information provider(s) of relations R and S guarantee that the constraint is

[3] Note that two constraints that carry the same data are not necessarily the same. For instance, in Example 3, $\mathcal{AC}_1 \neq \mathcal{AC}_{10}$.

and remains valid. Even when relation R becomes unavailable, this condition will still be kept valid. (If this $\mathcal{CC}$-constraint would also be deleted from the MKB, it could not be utilized by the view synchronization process). We define a rewritten view $V^{(1)}$ to be *dependent* on a $\mathcal{CC}$-constraint if it was used in the above sense in the rewriting of a view V.

Definition 4 (constraint-dependent) *A view rewriting $V^{(1)}$ for a view V is* constraint-dependent *on a constraint $(\mathcal{CC}, \mathcal{CPC}, \mathcal{AC}, \mathcal{RC})$ if the following conditions are satisfied:*

1. *A view rewriting $V^{(1)}$ is constraint-dependent on a Containment Constraint $\mathcal{CC}_1 = \pi_1(R)\ \theta\ \pi_2(S)$ iff relation R in V has been replaced by relation S in $V^{(1)}$ using $\mathcal{CC}_1$.*

2. *A view rewriting $V^{(1)}$ is constraint-dependent on a $\mathcal{CPC}_i = (R, S)$ iff $\mathcal{CPC}_i$ is part of a $\mathcal{CC}$ on which $V^{(1)}$ is constraint-dependent and R and S are used in the definitions of V and $V^{(1)}$, respectively.*

3. *A view rewriting $V^{(1)}$ is constraint-dependent on an $\mathcal{AC}$ or $\mathcal{RC}$ iff the attribute or relation defined in those constraints is used in the view definition of $V^{(1)}$.*

4. *A view rewriting $V^{(2)}$ for a view $V^{(1)}$ that is constraint-dependent on $\mathcal{CC}_i = \pi_1(R)\theta\pi_2(S)$ is also constraint-dependent on $\mathcal{CC}_i$ as long as relation $S \in V^{(1)}$ is retained in $V^{(2)}$. $V^{(2)}$ is not constraint-dependent on $\mathcal{CC}_i$ if relation S was dropped from $V^{(2)}$ without replacement or replaced with a relation T using a $\mathcal{CC}_j = \pi_1(R)\theta\pi_2(S)$ (i.e., not using $\mathcal{CC}_i$).*

5. *Constraint-dependency for a $\mathcal{CPC}(R.A, S.B)$ has the same property as $\mathcal{CC}_i$ in #4.*

Note that any further rewriting of view $V^{(1)}$ can but does not have to retain the view's constraint-dependency on any constraint (point 4 in the definition).

Example 5 *Assume that a view rewriting $V^{(i)} = (S \bowtie T)$ is constraint-dependent on the containment constraint $\mathcal{CC}_i = \pi_1(R)\theta\pi_2(S)$ (i.e., relation S is the replacement for a relation R which was part of V at an earlier stage). A meta data change* delete-$\mathcal{RC}(T)$ *would not affect that constraint dependency since relation S, and therefore $\mathcal{CC}_i$, would still be used in $V^{(i+1)}$. On the other hand, a further meta data change can make $V^{(i)}$ independent of any constraint by replacing the view element that causes the dependency (such as* delete-$\mathcal{RC}(S)$ *or* delete-$\mathcal{CC}(R, S)$ *in this example).*

3.5 Modeling the History of Meta Data and Views

We will now define a way to model the *history* of meta data and views, i.e., the sequence of changes in meta data that have occurred on a MKB and VKB.

Definition 5 (history graph) *Let a view V be rewritten into $V^{(n)}$ by a meta data change sequence $\mathcal{S} = \{\mathcal{MDC}_1, \mathcal{MDC}_2, \ldots, \mathcal{MDC}_n\}$. The* history $\mathcal{H}(V^{(n)})$ *of view rewriting $V^{(n)}$ is an graph $\mathcal{H} = (N, E)$ with N the set of nodes representing the states of both MKB and VKB at a certain point in time, and E the (labeled directed) edges representing transitions from one state to the next (meta data changes). Each node N_i is a pair $< MKB_i, VKB_i >$ and each edge E_i is labeled with the meta data change $\mathcal{MDC}_i$ it represents. Node N_0 contains the original Meta Knowledge and View Knowledge Bases (i.e., the meta knowledge at view definition time and the originally defined views) while N_n represents the current state of the system.*

Note that this graph defines a linear sequence, i.e., each node N_i has exactly one edge E_i leading to N_{i-1} and one edge E_{i+1} leading to N_{i+1}, except N_0 and N_n who have one edge each.

An obvious space-optimization (at the expense of execution time) on this structure is the storage of only meta data *changes* rather than the entire MKB at each node and generating the other MKB states at runtime. We are modelling the history graph in the way described to simplify the algorithm given later.

Example 6 *We will introduce an example to be used in the subsequent discussion. Using the MKB from Example 3, Figure 5 contains a part of a history graph of view V. The original view V is defined on an information space containing $\mathcal{RC}_1, \mathcal{RC}_2, \mathcal{RC}_3, \mathcal{CC}_1$, and $\mathcal{CC}_2$. We assume two meta data change operations, namely* delete-$\mathcal{RC}_1$ *and* delete-$\mathcal{CC}_1$. *The Meta Knowledge Base and the View Knowledge Base are evolved after each of these operations according to the meta-information available (using the POC algorithm, Section 2.2).*

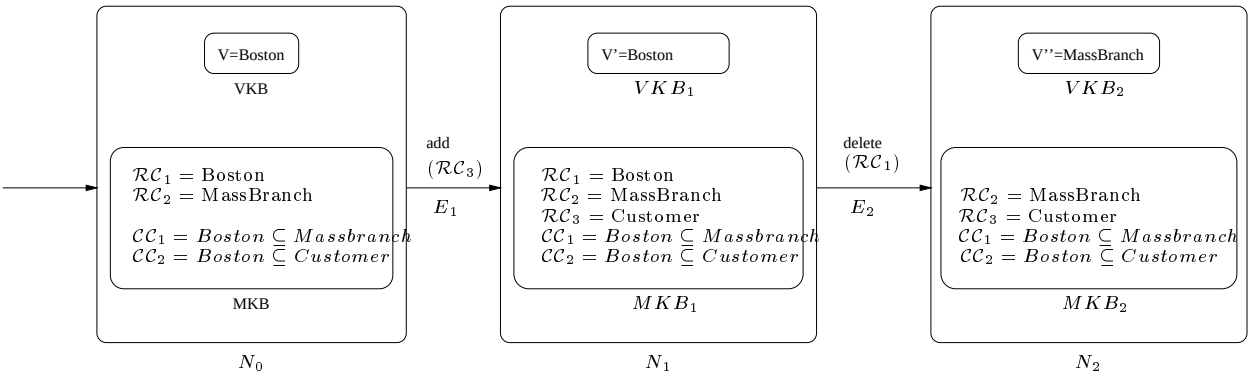

Figure 5: The History of Views and Meta Knowledge

4 The History-Driven View Synchronization Algorithm (HD-VS)

In this section, we will give the History-Driven View Synchronization algorithm (HD-VS). We will contrast the properties that this algorithm achieves with the list given for one-step algorithms in Section 2.2.

- As before, a synchronization occurs after each meta data change for an affected view. However, now input data are not only the current but also all previous states of MKB and VKB as well as the meta data changes that occurred, i.e., the complete history.
- If a view can be rewritten, the algorithm rewrites a valid view on the old information space into a valid view on the new information space. In certain cases, the algorithm falls back on the one-step view synchronization algorithms (e.g., POC).
- Now, not only *deletes*and *renames* but also all *adds* of meta data are considered. Meta data includes both *schema* such as relations and attributes, as well as other *constraints* such as containment constraints.
- If a view is rewritten, its quality may increase over the previous version (whereas the one-step algorithms could never improve on quality), depending on the meta data change that caused the synchronization.
- If a meta data item is deleted, a *cancellation* with a previous add is attempted and vice-versa. That means that *temporary unavailability* of data can be accounted for. The algorithm is capable of *returning* to a previous view definition if appropriate.

HD-VS uses three main concepts: backtracking in the history of a view, re-applying a part of a meta data update sequence from that history, and reconstructing part of the view's history graph in the process of re-application of meta data changes. Figure 6 gives an overview over the algorithm's main concept that are described in detail later in this section. Four meta data changes are executed on a view V. After each meta data change, HD-VS checks whether the history of the view contains information that could be used to rewrite view V in an appropriate way. Two consecutive *delete-constraint* operations both lead to backtracking in the history of the view but do not find an appropriate inverse operation to cancel with, and in both cases HD-VS falls back on a single-step view-synchronization algorithm (e.g., POC). The third operation, an *add-constraint* operation, also leads to unsuccessful backtracking but in the case of an addition to the information space, no POC-view synchronization is executed (since a view synchronization is not *necessary* for the view to remain valid). The fourth operation, an *add-operation* inverse to a previous *delete-*operation, leads to successful backtracking. Here, the two operations *add-X* and *del-X* cancel out, so the idea of the algorithm is to apply a meta data-sequence to V that does not contain these two operations, but all others. The remaining two operations (*del-Y* and *add-U* in our example) are applied in the abovementioned fashion, but do not lead to further cancellations, so the final view obtained is $V_2^{(2)}$.

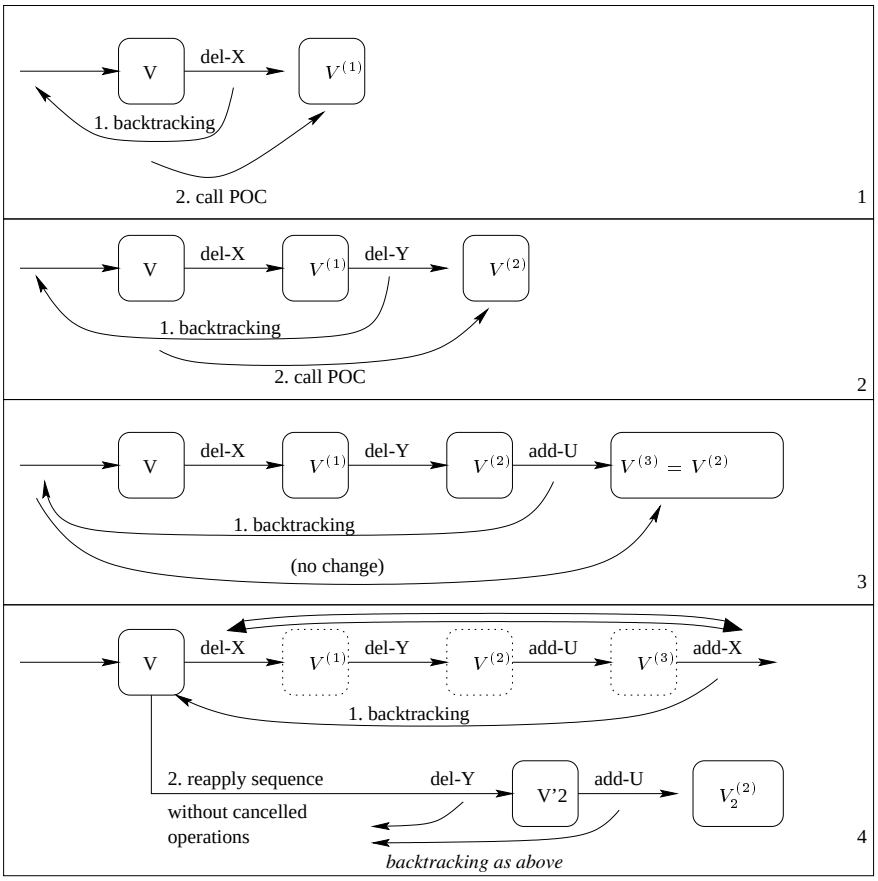

Figure 6: The History-Driven View Synchronization Algorithm (HD-VS).

4.1 Affected Views

View synchronization has to be performed when a view is affected after a schema change in the information space. Views whose definition are still valid after the change do not have to be synchronized. Therefore, we will define the concept of *affected views*.

Definition 6 (Affected View) *A view $V^{(i)}$ defined in the VKB is* affected *by a meta data change $\mathcal{MDC}$ in MKB_i iff:*

(1) a constraint $\mathcal{C}$ is added *to the MKB ($\mathcal{MDC} \in$ {add-$\mathcal{CPC}$, add-$\mathcal{CC}$, add-$\mathcal{RC}$, add-$\mathcal{AC}$}) and the view $V^{(i)}$ was at some point in its history constraint-dependent on $\mathcal{C}$ or*

(2) a constraint $\mathcal{C}$ is deleted *from the MKB ($\mathcal{MDC} \in$ {delete-$\mathcal{CPC}$, delete-$\mathcal{CC}$, delete-$\mathcal{RC}$, delete-$\mathcal{AC}$}) and the view $V^{(i)}$ is currently constraint-dependent on $\mathcal{C}$.*

4.2 Backtracking in the History of a View

In order to perform view synchronization, we apply a technique that we will call *backtracking*. Recall that the application of a meta data change sequence $\mathcal{S} = \{\mathcal{MDC}_1, \mathcal{MDC}_2, \ldots, \mathcal{MDC}_n\}$ to an original view V leads to a sequence of view rewritings $\{V^{(1)}, V^{(2)}, \ldots, V^{(n)}\}$, with its history being recorded in the history graph $\mathcal{H}(V)$. The purpose of the backtracking process is to find a view V_i in $\mathcal{H}(V^{(n)})$ from which to re-apply or *redo* a changed meta data update sequence. The idea of backtracking now consists in finding a node in $\mathcal{H}(V)$ for which a certain condition C holds.

Definition 7 (backtracking) *Backtracking in the history $\mathcal{H}(V^{(n)})$ of a view rewriting $V^{(n)}$ after a meta data change $\mathcal{MDC}_{n+1}$ to a constraint $\mathcal{C}_i$ is the traversal of the graph $\mathcal{H}(V^{(n)})$, starting from node N_n through nodes $N_{n-1}, N_{n-2}, \ldots$, until the following condition is satisfied:*

1. *if $\mathcal{MDC}_{n+1}$ is an* add-constraint-*operation: $V^{(i)}$ is constraint-dependent on $\mathcal{C}_i$ or*
2. *if $\mathcal{MDC}_{n+1}$ is a* delete-constraint-*operation: $V^{(i)}$ is **not** constraint-dependent on $\mathcal{C}_i$*

The backtracking process returns the node N_i with the highest index i for which the above condition is satisfied (i.e., the most recent view that satisfies the condition).

The motivation behind this definition of backtracking is as follows: If a constraint $\mathcal{C}_i$ is added to the MKB, we are trying to determine whether this constraint has been used at some point in the history of view V. In this case, $\mathcal{C}_i$ must have been deleted from the MKB earlier and we can *cancel* the add with the delete of $\mathcal{C}_i$. Therefore, we backtrack to a view $V^{(i)}$ that used $\mathcal{C}_i$, since $V^{(i)}$ must precede the deletion of constraint $\mathcal{C}_i$. In the opposite case of a delete-constraint-operation on constraint $\mathcal{C}_i$, we look for the last view $V^{(i)}$ that was not constraint-dependent on $\mathcal{C}_i$ since we can then re-apply the remaining schema change sequence from $V^{(i)}$ on under the assumption that $\mathcal{C}_i$ was never part of the MKB.

Example 7 *Figure 7 shows backtracking in a schema change sequence. Consider the information space of Example 3. We will assume a view V defined as $V =$* Boston *with a view extent evolution parameter of $\mathcal{VE} = '\supseteq'$ (Figure 8).*

Let us now assume a meta data change add-$\mathcal{RC}_3$ (defining relation `Customer`*) followed by a meta data change* delete-$\mathcal{RC}_1$ *(defining relation* `Boston`*). Let the POC-View-Synchronization Algorithm rewrite view V into $V^{(2)} =$* MassBranch *(cf. Figure 7). This rewriting makes use of Containment Constraint $\mathcal{CC}_1 = \pi_1(\text{Boston}) \subseteq \pi_2(\text{MassBranch})$ since it uses the information provided by it to ensure the validity of the new view. By Definition 4, i.e., view rewriting $V^{(2)}$ is constraint-dependent on $\mathcal{CC}_1$.*

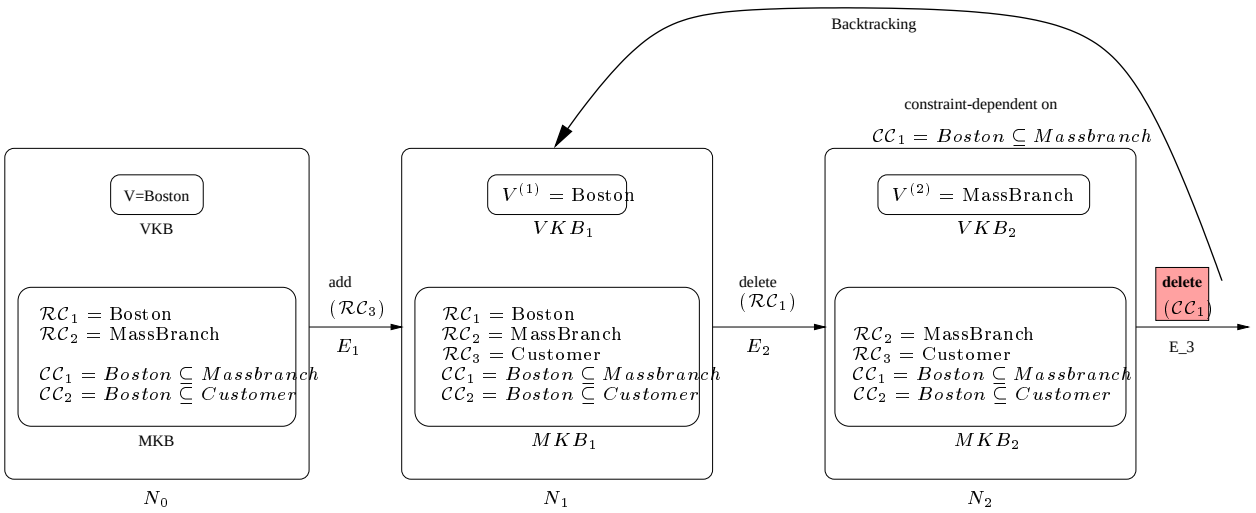

Figure 7: Backtracking in a View's History.

Let us now assume another meta data change delete-$\mathcal{CC}_1$(Boston, MassBranch). *The semantic meaning of that meta data change is that the information source provider that ensured the superset-relationship between* MassBranch *and* Boston *does not provide this service anymore. Therefore, it cannot be guaranteed that the view* $V^{(1)}$ = MassBranch *will be a superset of the former view* V = Boston.

The view synchronization algorithm will backtrack in the history of V *in order to find the most recent view that is not constraint-dependent on* $\mathcal{CC}_1$*, i.e., view* $V^{(1)}$ *in our example.*

4.3 Re-applying a Meta Data Update Sequence

After backtracking to a certain view $V^{(i)}$, a meta data change sequence S' derived from the original sequence S has to be applied to $V^{(i)}$.

In the first case of *add-constraint*-operations, after view $V^{(i)}$ has been found in node N_i, we remove the meta data change $\mathcal{MDC}_{i+k} = \mathcal{MDC}_{n+1}^{-1}$ from the meta data change sequence S to obtain $S' = \{\mathcal{MDC}_{i+1}, \ldots, \mathcal{MDC}_{i+k-1}, \mathcal{MDC}_{i+k+1}, \ldots, \mathcal{MDC}_n\}$. S' is then applied to MKB_i and view $V^{(i)}$, i.e. the operation $\mathcal{MDC}_{i+k}$ will not be applied to either MKB or view. In other words, $\mathcal{MDC}_k$ (the delete-operation inverse to $\mathcal{MDC}_{n+1}$) is canceled out by $\mathcal{MDC}_{n+1}$.

In the second case of *delete-constraint*-operations, we apply the meta data change sequence $S' = \{\mathcal{MDC}_{i+1}, \mathcal{MDC}_{i+2}, \ldots \mathcal{MDC}_n\}$ on $V^{(i)}$, but on an altered MKB. The MKB is altered in the following way: the meta data change $\mathcal{MDC}_{n+1}$ is applied to MKB_i and S' is then applied assuming this altered MKB. In the case of a *delete-*$\mathcal{CC}$ or *delete-*$\mathcal{RC}$, all dependent $\mathcal{CPC}$ or $\mathcal{AC}$ are also deleted from the MKB_i and all $\mathcal{MDC}_j \in \{\text{add}-\mathcal{CPC}\}$ or $\mathcal{MDC}_j \in \{\text{add}-\mathcal{AC}\}$, respectively, with $j \geq i+1$ are also removed from $\mathcal{S}'$.

4.4 Reconstructing the History Graph

As the sequence S' is applied to MKB_i, new nodes $N_{i+1} \ldots N_j (j \leq n)$ are created and the old nodes are discarded from the graph. Lastly, a new node N_{j+1} is added to $\mathcal{H}(V)$, containing MKB_{j+1} and VKB_{j+1} and connected with N_j through an edge E_{j+1} labeled with meta data change $\mathcal{MDC}_{j+1}$.

Example 8 *This examples continues from Example 7. Figure 8 shows the re-application of the schema change sequence and the reconstruction of the history graph.*

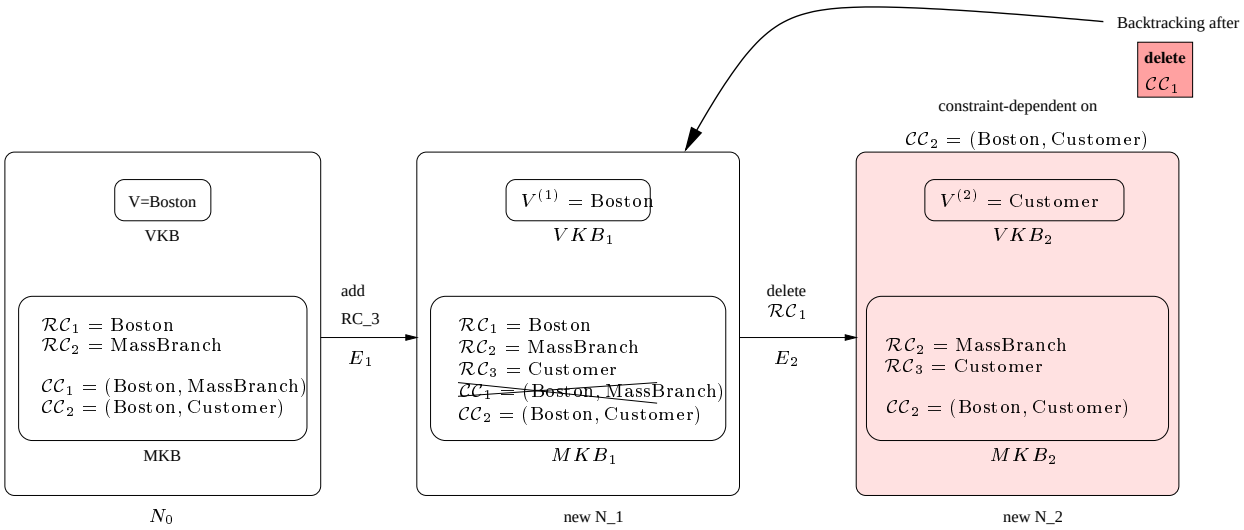

Figure 8: Reconstructing the History Graph

Remember that the view synchronization algorithm backtracked in the history of V in order to find view $V^{(1)}$. From that point in the history, all the other meta data change operations will be re-executed, i.e., the operation sequence $S = \{\text{delete-}\mathcal{RC}_1\}$ will be applied again, but now on the changed MKB without $\mathcal{CC}_1$. This will use the only remaining Containment Constraint $\mathcal{CC}_2$ and thus lead to a new synchronized view $V^{(2)} = \text{Customer}$. Node N_2 will be deleted from the history and replaced with a new node that is now also called N_2.

Figure 9 shows the complete HD-VS algorithm in detail.

4.5 Backtracking to the Beginning

In some cases, we may backtrack to the beginning of the meta data change sequence. Note that a naïve approach at history-driven view synchronization might be to *always* restart the entire view synchronization process from the beginning and apply the complete meta data change sequence after every new meta data change. However, considering that view synchronization is an expensive process [KRH98], and is only

```
ALGORITHM HD-VS
Input:  View rewriting V^(n)
        History H(V^(n))
        Meta Data Change MDC
Output: New History H(V^(j))
        View rewriting V^(j+1)

begin
  S := meta_data_change_sequence(H(V^(n)));
  i := 0; k := -1;
  if (MDC ∈ {delete-AC, delete-RC, delete-CC, delete-CPC})
    i := index of most recent view V^(i) not constraint-dependent on MDC;
  else
    i := index of most recent view V^(i)
    k := index of MDC_k = MDC^-1 constraint-dependent on MDC;
  // MDC_i occurs immediately before V^(i) is generated
  if (i > 0)
    begin
      apply MDC on MKB_i
      S' := subsequence of S from MDC_(i+1) until MDC_n;
      if (k != -1) remove MDC_k from S';
      if (MDC ∈ {delete-RC, delete-CC}) remove all dependent AC or CPC from S';
      apply S' on MKB_i and V^(i) to obtain H(V^(j)) and V^(j+1);
      return H(V^(j)) and V^(j+1);
    end
  else
    if (MDC ∈ {delete-AC, delete-RC, delete-CPC})
      begin
        apply delete-attribute or delete-relation algorithm from [NR98a]
          on V^(n) to obtain V^(n+1);
        return H(V^(n)) and V^(n+1);
      end
end.
```

Figure 9: HD-VS in Pseudocode.

defined for *one* meta data change at a time, this approach is ineffective in practice. Our algorithm minimizes the number of necessary view synchronization steps while maximizing the usefulness (quality) of the views generated. In addition, a re-application of the entire meta data change sequence would only be meaningful if that sequence would somehow be optimized before its application (since current VS algorithms can not meaningfully synchronize view under add-constraint-operations and can therefore not account for temporary absence of a constraint, i.e., pairs of delete/add-constraint operations).

4.6 Meta Data Change Operations in Detail

We now analyze details of the handling of each meta data change operation. In each case, the view synchronization algorithm will be executed only on the affected views. Table 1 gives a summary of the eight cases

whose important properties we will examine subsequently:

	delete	**add**
$\mathcal{AC}$	*delete-attribute* capability change (Sec. 2.2) or backtrack to view $V^{(k)}$ not constraint-dependent on $\mathcal{AC}$	backtrack to most recent view $V^{(k)}$ constraint-dependent on $\mathcal{AC}$
$\mathcal{RC}$	*delete-relation* capability change or backtrack to view $V^{(k)}$ not constraint-dependent on $\mathcal{RC}$	backtrack to view $V^{(k)}$ constraint-dependent on $\mathcal{RC}$
$\mathcal{CC}$	backtrack to view $V^{(k)}$ not constraint-dependent on $\mathcal{CC}$	backtrack to view $V^{(k)}$ constraint-dependent on $\mathcal{CC}$
$\mathcal{CPC}$	*delete-attribute* capability change or backtrack to view $V^{(k)}$ not constraint-dependent on $\mathcal{CPC}$	backtrack to view $V^{(k)}$ constraint-dependent on $\mathcal{CPC}$ or $\mathcal{CC}$ (see text)

Table 1: The Eight Meta Data Change Operations

4.6.1 *delete*-Operations

For the ***delete***-$\mathcal{AC}$ and ***delete***-$\mathcal{RC}$ operations, backtracking may lead to the beginning of the view's history. In this case, the *delete*-operation cannot be cancelled out with a previous add. Therefore, a single-step view synchronization algorithm (Section 2.2) has to be executed on the view $V^{(n)}$. Also, for the ***delete***-$\mathcal{CPC}$ operation, a single-step *delete-attribute* view synchronization has to be executed in the case of backtracking to the beginning.

4.6.2 *add*-Operations

The cases of *add*-$\mathcal{RC}$, *add*-$\mathcal{CC}$, and *add*-$\mathcal{AC}$ are straightforward. Let us look at an example for an *add*-$\mathcal{RC}$ improving a view rewriting $V^{(n)}$.

Example 9 *Assume the example MKB from Example 3 and an original view V which has undergone two meta data changes:* delete-$\mathcal{RC}$(Boston) *leading to* $V^{(1)}$ = MassBranch *constraint-dependent on* $\mathcal{CC}_1$ *and* delete-$\mathcal{RC}$(MassBranch) *leading to* $V^{(2)}$ = Customer *constraint-dependent on* $\mathcal{CC}_2$. *We assume that neither of the two containment constraints is deleted from the MKB. Subsequently, let us assume a meta data-change-operation* add-$\mathcal{RC}$(MassBranch), *i.e., relation* MathBranch *is "coming back" into the information space. Since* $\mathcal{CC}_1$ *still holds, we can intuitively just "go back" to* $V^{(1)}$ *to obtain another valid view. The above algorithm achieves just that by backtracking to the node containing V (since it V is the most recent view constraint-dependent on relation* MassBranch*) and re-applying only the* delete-$\mathcal{RC}$(Boston) *meta data change.*

Note that a *add-$\mathcal{AC}$*-Operation can improve a view definition even without a preceding delete. If a relation R has at any point been replaced by a relation S, but attribute $R.A$ has been omitted in the replacement (i.e., was not replaced by $S.B$, e.g. because $R.A$ was dispensable) then an operation *add-$\mathcal{AC}$*(S, B) can be used to obtain an improved view rewriting. For this, we need to check whether the $\mathcal{CC}$ that has been used to replace R with S now contains a $\mathcal{CPC}_i = (R.A, S.B)$ and can then simply add attribute $S.B$ to our view rewriting $V^{(n)}$.

The *add-$\mathcal{CPC}$* Operation. This operation is slightly more complicated that the other three add-Operations. Adding a $\mathcal{CPC}_i = (R.A, S.B)$ (a pair of attributes) to a Containment Constraint $\mathcal{CC}_i = \pi_1(R)\theta\pi_2(S)$ makes the latter constraint stronger in the sense that the system is now made aware of a larger overlap between R and S. Previously valid information (a certain projection of table R is a subset or superset of a projection of table S) remains valid. Therefore, adding a $\mathcal{CPC}$ can only lead to an improved query interface, never to a decrease in view quality.

If this $\mathcal{CPC}$ has earlier been deleted from the MKB and there is a view $V^{(1)}$ constraint-dependent on it, $\mathcal{CPC}$ can be canceled out (if there is an $\mathcal{AC}_i(S) = B$). This is accomplished in the usual way by backtracking in the history of $V^{(1)}$ and re-applying the sequence of meta data changes with the exception of *delete-$\mathcal{CPC}$*.

Under some circumstances, adding a $\mathcal{CPC}$ can improve a view rewriting even without a previous delete-$\mathcal{CPC}$-operation (i.e., with no view constraint-dependent on $\mathcal{CPC}$). After an *add-$\mathcal{CPC}$* -operation ($\mathcal{MDC}_{n+1}$ = *add-$\mathcal{CPC}_i$* $= (\text{Att}_1, \text{Att}_2)$ to $\mathcal{CC}_k = \pi_1(R)\theta\pi_2(S)$), we backtrack in the history of the view rewriting $V^{(n)}$ and search for a view $V^{(i)}$ constraint-dependent on $\mathcal{CC}_k$. If $V^{(n)}$ is constraint-dependent on $\mathcal{CC}_k$, we backtrack further to a view $V^{(l)}$ not constraint-dependent on $\mathcal{CC}_k$. We check if the attribute $S.\text{Att}_2$ referred to in $\mathcal{CPC}$ was used in the view $V^{(l)}$. If that is the case, and the attribute actually exists (i.e., $\mathcal{AC}(S) = \text{Att}_2$ exists in MKB_n) we can simply add $S.\text{Att}_2$ to $V^{(n)}$ to obtain an improved $V^{(n+1)}$.

5 Evaluation

5.1 Correctness

Definition 8 (valid view) *A view V is* valid *over an information space defined by MKB if all constraints on which V is constraint-dependent (Def. 4) in fact exist in the MKB.*

Definition 9 (valid meta data change) *A meta data change $\mathcal{MDC}_i$ is* valid *over an MKB if*

- $\mathcal{MDC}_i$ ***exists*** *in MKB in the case of a* delete-*constraint or*
- $\mathcal{MDC}_i$ ***does not exist*** *in MKB in the case of an* add-*constraint.*

Theorem 1 *Using the History-driven View Synchronization Algorithm, view synchronization of a valid view* $V^{(i)}$ *after a valid meta data change* $\mathcal{MDC}_i$ *that does* not *lead to backtracking will lead to a valid view* $V^{(i+1)}$.

Proof: View synchronization as introduced in this paper is an operation defined on arbitrary views $V^{(i)}$, as long as they are valid over a state MKB_i of the Meta Knowledge Base. Any meta data change that does not lead to backtracking will either do only trivial changes to the view (adds which do not affect the views or renames which are easily handled) or apply a single-step view synchronization algorithm according to Section 2.2 which has been proven to produce only valid views under any given schema change [NR98a].

Therefore, a valid meta data change that does not cause backtracking will lead to a valid view rewriting.

□

Theorem 2 *Assume a meta data change sequence* $\mathcal{S} = \{\mathcal{MDC}_1, \mathcal{MDC}_2, \ldots, \mathcal{MDC}_{n+1}\}$ *that led to a view rewriting* $V^{(n+1)}$. *Backtracking in the history of* $V^{(n+1)}$ *to view rewriting* $V^{(i)}$ *caused by* $\mathcal{MDC}_{n+1}$ *and applying a changed sequence of meta data changes* $\mathcal{S}' = \{\mathcal{MDC}_{i+1}, \mathcal{MDC}_{i+2}, \ldots, \mathcal{MDC}_m\}$ *as described in Section 4 on the view* $V^{(i)}$ *defined over* MKB_i *will always lead only to valid rewritings* $V^{(i+1)} \ldots V^{(n+1)}$.

Proof: Sequence $\mathcal{S}$ consists of meta data changes that led to valid view rewritings $V^{(1)}, V^{(2)}, \ldots, V^{(n)}$ (i.e., all $\mathcal{MDC}$ in $\mathcal{S}$ are valid). An intermediate $\mathcal{S}'$ is constructed as $\mathcal{S}' := \{\mathcal{MDC}_{i+1}, \mathcal{MDC}_{i+3}, \ldots, \mathcal{MDC}_{n+1}\}$. Then, all meta data changes that are conflicting with $\mathcal{MDC}_{n+1}$ (cf. Section 4.3) are removed from $\mathcal{S}'$. For a *delete-*$\mathcal{CC}$-Operation those are all *add-*$\mathcal{CPC}$- and *delete-*$\mathcal{CPC}$-Operations that are related by the *con* function (Def. 1), for a *delete-*$\mathcal{RC}$-Operation those are all *add-*$\mathcal{AC}$- and *delete-*$\mathcal{AC}$-Operations that are related by the *rel* function. After a meta data change $\mathcal{MDC}_{n+1}$, according to Section 4, there are then two ways of applying $\mathcal{S}'$ to a view $V^{(i)}$:

1. After add-constraint-operations:

   In this case, $\mathcal{S}'$ is first updated by removing $\mathcal{MDC}_k = \mathcal{MDC}_{n+1}^{-1}$ with $(k \geq i)$, the inverse of the current meta data change (cf. Section 4), then all dependent operations ($\mathcal{AC}$ or $\mathcal{CPC}$ changes) are removed and finally $\mathcal{S}'$ is applied on MKB_i. Since the combination of the two inverse operations $\mathcal{MDC}_k$ and $\mathcal{MDC}_{n+1}$ and the removal of dependent constraints does not overall change the MKB, the rest of S' will consist of only valid meta data changes and therefore lead to valid views (due to Theorem 1).

2. After delete-constraint-operations:

   In this case, $\mathcal{MDC}_{n+1}$ is first applied to MKB_i, which is possible since $V^{(i)}$ is a valid view constraint-dependent on constraint($\mathcal{MDC}_{n+1}$) and therefore MKB_i must contain constraint($\mathcal{MDC}_{n+1}$). Then $\mathcal{S}'$ is applied on MKB_i and $V^{(i)}$. With Theorem 1, this sequence of valid meta data changes will then lead to a sequence of valid views.

□

Since $\mathcal{S}'$ creates valid views, the new history $\mathcal{H}(V^{(n+1)})$ will contain only valid views and the valid meta data changes representing the edges of the history graph form the new and valid sequence $\mathcal{S} := \{\mathcal{MDC}_1, \mathcal{MDC}_2, \ldots, \mathcal{MDC}_i, \mathcal{S}'\}$.

5.2 Improving Views after Meta Data Changes

In order to evaluate the usefulness of our approach, we have to assess if this view synchronization process produces views that better satisfy users' needs. The previous approach at view synchronization (Section 2.2) takes only *delete*-capability-changes into account and ignores adds, which causes views to deteriorate over time and become increasingly different from their originals.

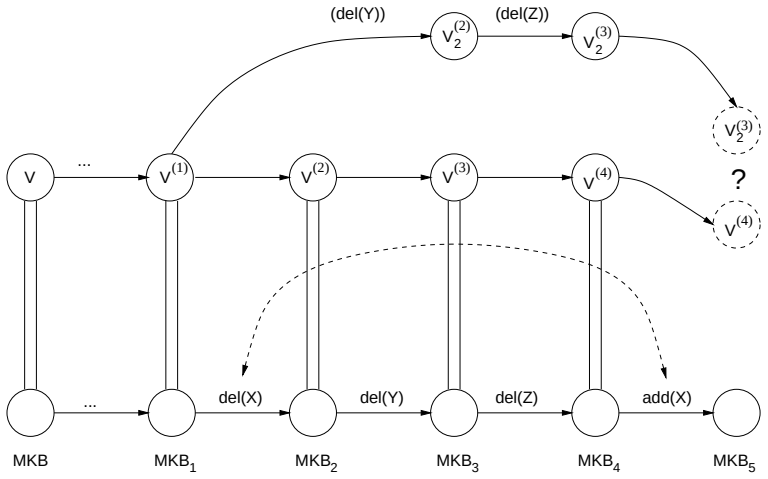

Figure 10: A Meta Data Change Sequence

Figure 10 illustrates this issue by an example of a meta data change sequence that uses backtracking. Previous approaches [RLN97, NLR98, NR98b, KRH98] would simply ignore the add-capability change. Here instead, we also consider the cancellation of the *del-X*-operation, and subsequently re-applying *del-Y* and *del-Z* will lead to a better view $V_2^{(3)}$ than simply ignoring the *add-X* operation, obtaining $V^{(4)}$. For this, we would have to compare the quality of those two alternate rewritings to the user. Quality is measured by the QC-value (Section 2.3, fully defined in [LKNR99b]) so that in the example, we would have to evaluate if $Q(V_2^{(3)}) > Q(V^{(4)})$ [4]. Generalized, the question has to be answered if the quality $Q(V^{(n)})$ of a view rewriting will improve when the history-based algorithm described in this paper instead of a one-step algorithm is applied.

[4] We refer here only to the quality part of the QC-Value, not the cost part.

Theorem 3 *Let $V^{(n-1)}$ be a view rewriting of view V obtained after a meta data change sequence $\mathcal{S} = \{\mathcal{MDC}_1, \mathcal{MDC}_2, \ldots, \mathcal{MDC}_{n-1}\}$, and $\mathcal{MDC}_n \in \{\text{add}-\mathcal{CC}, \text{add}-\mathcal{CPC}, \text{add}-\mathcal{AC}, \text{add}-\mathcal{RC}\}$, the next meta data change. Then, a view rewriting $V^{(n_2)}$ obtained through the history-based view synchronization will have a higher or equal quality $Q(V_2^{(n)})$ than a rewriting V^n obtained with previous view synchronization algorithms as described in Section 2.2, i.e., $Q(V_2^{(n)}) \geq Q(V^{(n)})$.*

Proof: Assume a meta data change sequence $\mathcal{S} = \{\text{delete}-X, S', \text{add}-X\}$ with $|S| = n$ and S' a sequence that does not contain add$-X$ or delete$-X$ for some constraint X. Note that a comparison is only meaningful for a $X \in \{\mathcal{AC}, \mathcal{RC}\}$, since $\mathcal{CC}$ and $\mathcal{CPC}$ changes are not handled by the one-step approach. Applying $\mathcal{S}$ on a view V leads to a number of view rewritings. The one-step view synchronizer described in Section 2.2 would apply the POC-algorithm after each schema change, *including* $\mathcal{MDC}_1 = \text{delete}-X$ to obtain a sequence of rewritings $V^{(1)}, \ldots, V^{(n)}$. HD-VS *cancels* $\mathcal{MDC}_1$ with $\mathcal{MDC}_n$ and therefore applies only S' (reverting back to POC for some or all steps in S') in order to obtain $V_2^{(1)}, \ldots, V_2^{(n-2)}$. We prove that HD-VS performs equal or better, i.e., that $Q(V_2^{(n-2)}) \geq Q(V^{(n)})$ holds.

$\mathcal{MDC}_1$ is the only operation that leads to a rewriting in only one algorithm (the one-step approach) but not in the other. Therefore, in the one-step algorithm, $\mathcal{MDC}_1 = \text{delete}-X$ could lead to a view rewriting $V^{(1)}$ from which the algorithm (based on POC) eventually arrives at a view $V^{(n)}$ whose quality is higher than the best $V_2^{(n-2)}$ that can be found by HD-VS (using POC at its different steps) without applying $\mathcal{MDC}_1$.

For a view synchronization, POC replaces a relation or attribute X with exactly one relation or attribute Y or drops X if no replacement is found [NR98b]. If X is replaced by Y, a containment constraint $\mathcal{CC}_i = X\theta Y$ must have been used to justify this replacement (by definition of POC). There are now three cases for the remaining meta data sequence S':

1. $|S'| = 0$. In this case, HD-VS does not execute any view synchronization operation, while the one-step approach just executes $\mathcal{MDC}_1$. Since each delete-operation (including $\mathcal{MDC}_1$) under POC decreases the quality of the view (as stated in Section 2.2), $Q(V_2^{(n-2)}) \geq Q(V^{(n)})$ holds.

2. If S' does not contain a delete$-Y$-operation, then no operation in S' is dependent on delete$-X$. In this case, both algorithms perform the exact same rewritings on the remaining components of the view specification V (besides X), with the exception that POC applies $\mathcal{MDC}_1$ which decreases the quality. Thus $Q(V_2^{(n-2)}) \geq Q(V^{(n)})$.

3. If S' contains a delete$-Y$ Operation, POC drops or replaces Y with Z given a $\mathcal{CC}_j = Y\theta Z$. If the MKB contains both $\mathcal{CC}_i$ and $\mathcal{CC}_j$, a third containment constraint $\mathcal{CC}_k = X\theta Z$ can be derived due to the transitivity of containment constraints. In particular, HD-VS can also generate a rewriting $V^{(n-2)}$

by doing the replacement of X to utilize Y as done by POC, or, while POC would drop X, HD-VS may not. If X is dropped by POC, the resulting rewriting must be worse than any rewriting that replaces X with another element. Therefore, with the previous argument, the resulting $V^{(n)}$ must also be worse than $V_2^{(n-2)}$.

□

Theorem 4 *The HD-VS algorithm achieves* $Q(V_2^{(n)}) \geq Q(V^{(n)})$ *even after* delete-*capability changes.*

Proof: In the case of delete-capability changes (delete$-\mathcal{CC}$, delete$-\mathcal{CPC}$, delete$-\mathcal{AC}$, delete$-\mathcal{RC}$) the HD-VS algorithm produces the exact same views as the one-step POC-based algorithm in all comparable cases, namely for *delete-*$\mathcal{AC}$ and *delete-*$\mathcal{RC}$ [5]. Therefore, the HD-VS algorithm will never perform worse than the one-step approach, but in many cases better. □

6 Implementation

In the *EVE*-Project [RLN97, NLR98, NR98b, KRH98], a prototype dynamic data warehouse has been developed [RKZ+99]. The *EVE*-System (built in Java over JDBC-capable SQL-databases) currently performs (one-step) view synchronization using the POC algorithm. An extension of this system for history-driven view synchronization is relatively straightforward since most of the necessary modules already exist. Figure 11 shows the *EVE*-Architecture augmented by the necessary changes for the History-Driven View Synchronization Algorithm.

The only new module needed for this algorithm is the *history manager*. It combines the data stores of VKB and MKB since HD-VS builds graphs whose nodes are pairs (VKB, MKB). The history manager is responsible for keeping all necessary historical states of the VKB and MKB as well as for providing operations to support backtracking, reapplication, and reconstruction of the history.

A HD-VS view synchronizer wraps the old one-step view synchronizer (e.g., running POC) and uses it when necessary. It also makes use of the QC-Computation module in order to decide about the quality of the generated views, and interacts with the history-manager during synchronization.

7 Related Work

Materialized views over distributed information sources have been explored for a number of years. First work focused on questions of materialized view maintenance under data updates in the sources [BLT86,

[5] Note that deletion of $\mathcal{CC}$ and $\mathcal{CPC}$ is not handled in the one-step approach, so a comparison is not meaningful.

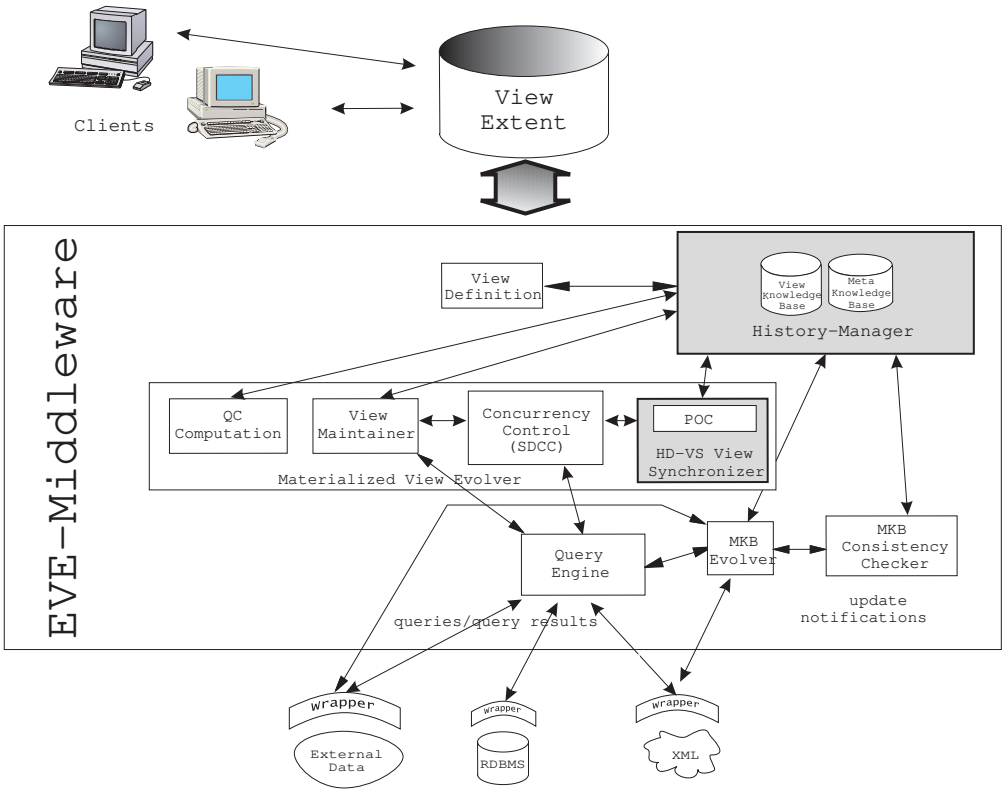

Figure 11: Augmenting the *EVE*-Architecture for HD-VS.

GM95, QW97]. More recently, questions of optimizing view queries given varying parameters or capabilities of underlying sources have also been explored. Generally, work in this area assumes that the rewritten view query computes a view extent *equivalent* to the original one. Prominent approaches that deal with equivalent query rewriting include work by Jarke et al. [JK84], van den Berg et al. [vdBK94], or Agrawal et al. [AASY97]. Important work on integrating heterogeneous sources in one view using a common semistructured data model (OEM) has been done in the TSIMMIS-project [GMHI+95].

Some work has been done on rewriting queries using materialized views [LRU96, RSU95, LMS95, SDJL96, RU96]. This work is relevant to the *EVE* project, although it generally deals with rewriting queries into *equivalent ones* using underlying views.

Work on rewriting queries using views (e.g., [LMSS93, LS92]) is used in subsequent work by Levy et al. which is closely related to ours in terms of its goal, but not the approach taken. In [LSK95], 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 [RLN97]. 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 an elegant solution to

a subset of problems that we also solve. It is nevertheless 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 non-equivalent rewrites of queries in the case of the possible unavailability of one of the sources. With the help of a quality measure (QC-Value [LKNR99b]), we can also decide which of a number of given information sources provides the *best* answer to a query.

Arens et al. [AKS96] and the SoftBot project [EW94] provide similar approaches as Levy which solve similar problems. Although addressing different issues, SIMS' process of finding relevant information sources for a query 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 "links" among data sources that are described by action schemas. While related to our view synchronization algorithm CVS [NLR98], the SoftBot planning process also relies on discovering connections among information sources when very different source description languages are used. Neither SIMS nor SoftBot address the particular problem of evolution under capability changes of participating external information sources.

CoBase by Chu et al. [CMB93] relates to our work in that they also use the notion of relaxation of the *query extent*, similar to our E-SQL approach [RLN97]. Chu 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'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 [LKNR99a, LKNR99b] (of which our algorithms would normally generate several) in order to help a view synchronization algorithm to find trade-offs among query rewritings.

Since in our approach we are using the concept of versioning meta-knowledge in a database system, work in versioning is also relevant in that it could be used as an implementation basis for our history graph. One

example of repository work is Microsoft's Repository [BBPS99] which now also supports versioning.

8 Conclusion

In this paper, we present an approach at view synchronization under meta data updates. With our algorithm, a view that is defined over a given set of relations and attributes and other meta knowledge that is defined in a Meta Knowledge Base (MKB) can be synchronized with *additions and deletions* of all elements in that MKB. Previous approaches [RLN97, NR98b, NR98a] at this problem could only adapt views to *deletions* in the information, but not to additions, so that, for example, temporary unavailability of information sources could not be accounted for. The main contributions of this work are (1) formal foundations of view synchronization taking the history of view changes into account, including the precise definition of meta data changes, (2) the investigation of the effects of meta data changes on meta data (i.e. dependencies between meta-data entities), (3) the definition of the notion of *history* of views and meta data, leading to versioning of both, (4) the definition of a *backtracking* algorithm making use of the view's history for future view rewritings, and (5) a proof of correctness and assessment of the improvements of that algorithm for view synchronization over previous approaches.

The technology presented in this paper builds upon previously describes view synchronization algorithms and uses them in particular cases. It is more powerful that the previous approaches, yet simple to implement given an implementation of single-step view synchronization. This technology can be applied to views over distributed information sources, especially as those sources are independent and prone to change their schemas or query capabilities. We are extending our current *EVE*-Demo software [RKZ+99] to incorporate the results obtained in this paper in a prototype data warehouse system.

References

[AASY97] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

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

[BBPS99] T. Bergstraesser, P. A. Bernstein, S. Pal, and D. Shutt. Versions and workspaces in Microsoft repositories. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 28, 2 of *ACM SIGMOD Record*, pages 532–533, New York, June 1999. ACM Press.

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

[CMB93] W. W. Chu, M. A. Merzbacher, and L. Berkovich. The Design and Implementation of CoBase. *SIGMOD Record*, 22(2):517–522, June 1993.

[EW94] O. Etzioni and D. Weld. A Softbot-based interface to the Internet. *Communications of the ACM*, 37(7):72–76, July 1994.

[GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2):3–19, 1995.

[GMHI+95] H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, pages 111–152, 1984.

[KRH98] A. Koeller, E. A. Rundensteiner, and N. Hachem. Integrating the Rewriting and Ranking Phases of View Synchronization. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'98)*, November 1998.

[LKNR99a] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 255, March, Sydney, Australia 1999.

[LKNR99b] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Non-Equivalent Query Rewritings. In *International Database Conference*, Hong Kong, July 1999.

[LMS95] A. Levy, A. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 95–104, May 1995.

[LMSS93] A. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query reachability and satisfiability in datalog extensions. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–122, Washington, DC, 25–28 May 1993.

[LNR97] 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 Institute, Dept. of Computer Science, 1997.

[LRU96] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external processors. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–237, Montreal, Canada, 3–5 June 1996.

[LS92] A. Levy and Y. Sagiv. Constraints and Redundancy in Datalog. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2–4, 1992, San Diego, CA*, pages 67–80, 1992.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems - Special Issue on Networked Information Discovery and Retrieval*, 5(2):121–143, 1995.

[NLR97] 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 Institute, Dept. of Computer Science, 1997.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[NR98a] A. Nica and E. A. Rundensteiner. The POC and SPOC Algorithms: View Rewriting using Containment Constraints in *EVE*. Technical Report WPI-CS-TR-98-3, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[NR98b] A. Nica and E. A. Rundensteiner. Using Containment Information for View Evolution in Dynamic Distributed Environments. In *Proceedings of International Workshop on Data Warehouse Design and OLAP Technology (DWDOT'98)*, Vienna, Austria, August 1998.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*, pages 393–400, 1997.

[RKZ+99] E. A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li. Evolvable View Environment. In *Proceedings of SIGMOD'99 Demo Session*, pages 553–555, May/June 1999.

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

[RSU95] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering Queries Using Templates With Binding Patterns. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 105–112, May 1995.

[RU96] A. Rajaraman and J. Ullman. Integrating Information by Outerjoins and Full Disjunctions. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 238–248, 1996.

[SDJL96] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering Queries with Aggregation Using Views. In *International Conference on Very Large Data Bases*, pages 318–329, 1996.

[vdBK94] C. A. van den Berg and M. Kersten. An Analysis of a Dynamic Query Optimization Schema for Different Data Distributions. In J. C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*, chapter 15, pages 449–473. Morgan Kaufmann Pub., 1994.

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.

[ZR99] X. Zhang and E. A. Rundensteiner. The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks. In *International Database Engineering and Application Symposium*, pages 206–214, Montreal, Canada, August, 1999.