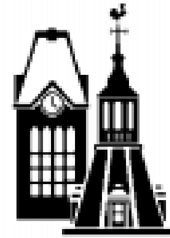
WPI-CS-TR-98-23 October 1998

Integrating the Rewriting and Ranking Phases of View Synchronization

by

Andreas Koeller
Elke A. Rundensteiner
Nabil Hachem

Computer Science Technical Report Series



WORCESTER POLYTECHNIC INSTITUTE

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

Integrating the Rewriting and Ranking Phases of View Synchronization*

Andreas Koeller, Elke A. Rundensteiner and Nabil Hachem

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{koeller|rundenst|hachem}@cs.wpi.edu

Abstract

Materialized views (data warehouses) are becoming increasingly important in the context of distributed modern environments such as the World Wide Web. Information sources (ISs) in such an environment may change their capabilities (schema). This causes a data warehouse defined by view queries over distributed sources to become undefined. Algorithms have been proposed to evolve (rewrite) view queries after capability changes of ISs by exploiting meta-descriptions about ISs and their relationships. This view rewriting process is referred to as view synchronization. View synchronization algorithms generate a potentially large number of valid solutions for the rewriting of a view query. Our analysis, as presented in this paper, shows that the most expressive algorithm for view synchronization has very high complexity (in $O(n!)$). The objective of this current work is hence to propose optimizations for this view synchronization process in order to make it practically usable. For this, we identify the expensive operations within the overall process of view synchronization and propose to reduce this complexity by representing the synchronization problem as a graph traversal problem. Once this mapping has been applied, the problem can be reduced to a single-source shortest-path problem in graphs. The solution to the view synchronization problem can thus be found by the Bellman-Ford algorithm, which has $O(n^3)$ complexity. This optimized polynomial approach towards view synchronization is now being incorporated as an optimizer module into the Evolvable View Environment (EVE) system.

Keywords: Evolvable view environment, view synchronization, optimized synchronization algorithm, data warehouse, cost model, efficiency, evolving information sources, shortest path problem.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 94–57609, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 97–32897. Dr. Rundensteiner would like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and for the IBM corporate fellowship for one of her graduate students.

1 Introduction

WWW-based information services such as data warehousing, digital libraries, data mining typically gather data from a large number of interconnected Information Sources (ISs). In order to provide efficient information access to such information services, relevant data is often retrieved from several sources, integrated as necessary, and then assembled into a *materialized view* (data warehouse) [Wid95]. Besides providing simplified information access to customers without the necessary technical background, materialized views also offer higher availability and query performance.

One important unsolved problem for these applications is that traditional view technology only supports *static, a-priori-specified* view definitions. In our recent work we study the maintenance of data warehouses defined over distributed *dynamic* information sources [LNR97a, LNR97b, RLN97]. A view can survive *schema changes* of its underlying information sources by making use of meta-information about those sources and applying algorithms for rewriting the view query; a process to which we refer to as *view synchronization*. The dynamicity of information sources in terms of not only data updates but also schema (query interface) changes motivated the development of algorithms for rewriting view definitions triggered by such schema changes [LNR97a, LNR97b]. View synchronization is in contrast to the large body of work on incremental view maintenance that addresses changes at the data but not at the schema level [ZGMHW95] and to recent work on view redefinition that again focuses on how to efficiently update the view data [GMR95].

Previous work on query optimization [vdBK94, AAS97, BLT86] and rewriting view queries [LMS95] was restricted to always requiring exact equivalence between the original and replacement queries. This condition is not likely to be always achievable in a dynamic environment such as the WWW. We have proposed a model of relaxed query semantics to allow for the rewritings of view queries that preserve different view extents and view interfaces (we call this the *quality* of the view) and result in different view maintenance costs [NLR98]. Since we will in general be able to generate a number of different such rewritings for a given situation, we need to compare rewritings with each other and with the original view to determine their desirability for a view user. To address this open issue, we have developed the *QC-Model*[1] [LKNR98] as an integrated measure of both the *quality* and *cost* dimensions of a view rewriting. Once the synchronization algorithm generates a set of possibly non-equivalent rewritings of view queries, the QC-Model is applied as a metric to each rewriting to establish a ranking among them.

In this paper, we now examine the *complexity* of this view synchronization process [LNR97b, NLR97]. We identify that its most powerful algorithm has a high complexity (in $O(n!)$). It generates an exponential (in the number of relations in the information space) number of query rewritings to which we then have to apply the QC-Model to establish a ranking among all solutions. We now reduce this complexity by mapping the problem of complex view synchronization to a polynomial complexity graph traversal problem. One key ingredient of our solution is the observation that the computation of quality and cost measures, namely the ranking of view queries,

[1] QC stands for the Quality- and Cost dimensions of the model.

previously done as a post-processing step to each view query identified by the view synchronization algorithm, can be decomposed into a stepwise computation by expressing view synchronization as a graph problem. This also allows us to integrate the query ranking phase with the finding of rewritings instead of executing two separate phases of view synchronization. We term our solution the *Optimized CVS* algorithm and show that the algorithm has a complexity in $O(n^3)$.

The contributions of this work are: the identification of a problem in view synchronization, namely the high complexity of available synchronization algorithms; the proposal of a mapping of the view synchronization problem into a graph problem; semantics for this mapping; the solution of the mapped graph problem by a shortest-path graph algorithm; a proof for the correctness and applicability of the approach.

The remainder of this paper is organized as follows. Section 2 reviews related work, while Section 3.1 introduces background material on view synchronization. Section 3 reviews the CVS algorithm and the computation of QC-values and examines the complexity of the (previous) algorithm. Section 4 proposes the new Optimized CVS algorithm. We also prove the correctness of the new algorithm and show its implications for view synchronization. We conclude this work in Section 5.

2 Related Work

While most prior work on database views in distributed environments has focused on view maintenance (e.g., propagating *data* changes to the view) [QW97], we have proposed algorithms for view redefinition caused by *capability* changes of ISs (called *view synchronization*), which is, to the best of our knowledge, the first solution to this problem. In [RLN97, LNR97a, LNR97b], the overall *EVE* solution framework was introduced, in particular the concept of associating evolution preferences with view specifications and algorithms for view synchronization. The Complex View Synchronization (CVS) algorithm [NLR98] generates a large number of alternative legal rewritings, thus raising the need for a way to evaluate and compare these rewritings. In [LKNR98], this need was addressed by establishing a model for systematically ranking otherwise incomparable solutions for view synchronization based on the two dimensions of quality and maintenance costs.

Much research has been done on query reformulation using materialized views. Levy et al. [LMS95] consider the problem of replacing an original query with a new expression containing materialized view definitions such that the new query is *equivalent* to the old one. To the best of our knowledge, generating such queries without *equivalence* (for example, the new reformulated query could be a subset of the original query) has not been studied.

The problem of query rewriting for *optimization* purposes has been addressed for instance by van den Berg et al. [vdBK94] and Agrawal et al. [AAS97]. They are concerned with optimizing a given query for efficient execution. View synchronization encounters a different problem, namely to select a good (but not necessarily *equivalent*) query among several possible ones. We are concerned with the problem of finding such a good query (or a small number of them) in an efficient way, taking view maintenance cost [ZGMHW95] and the new concept of view quality into account.

Incremental view maintenance has been an active area of research [CTL+96, GMS93] but is limited to changes

at the data level, whereas we are concerned with view maintenance under schema changes. Blakeley et al. [BLT86] assumed a centralized environment, while Zhuge et al. [ZGMHW95] introduce the ECA algorithm for incremental view maintenance restricted to a single IS. In Strobe [ZGMW96], they extend their findings towards multi-source information spaces. Agrawal et al. [AAS97]'s SWEEP algorithm ensures consistency of the data warehouse in a larger number of cases compared to Strobe.

3 The View Synchronization Process

3.1 Foundations of Non-Equivalent View Synchronization

In this section, we briefly review the concepts of the *Evolvable View Environment* (*EVE*) [LNR97b, RLN97, NLR98] system as needed for the remainder of this paper. *EVE* has been designed to evolve views in the presence of capability changes of information sources.

E-SQL or *Evolvable-SQL* is an extension of SQL that allows the view definer to express preferences for view evolution [LNR97b]. A user defining a view can specify what information is indispensable, what information is replaceable by similar information from other 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 flexibility to evolve a view under schema changes in a controlled way while preserving the user's intended semantics. For the purpose of this paper, we will not need details of the E-SQL extensions. The reader is referred to [LNR97b, RLN97].

In order to enable view synchronization, our system needs to be able to identify view element replacements from other ISs. MISD, our *Model for Information Source Description*, expresses relationships between ISs using constraints (e.g., agreeing data types, functional dependencies between attributes, extent overlaps between relations). These descriptions form an information pool that is critical in finding appropriate replacements for view components when view definitions become undefined. Constraints relevant for this paper are *join-constraints* and *containment-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 over the given set of join conditions with possibly another conjunction of primitive clauses satisfied. A typical join constraint is:

$$\mathcal{JC}_{\mathrm{PreferredCust,Accident-Ins}} = (\mathrm{PreferredCust.PrefName{=}Accident{-}Ins.Holder\ AND\ Accident{-}Ins.Amount{>}100000}) \quad (1)$$

A *$\mathcal{PC}$-constraint (partial/complete 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. The information described in MISD is stored in our system in the *Meta Knowledge Base (MKB)*.

3.2 The CVS Algorithm for Rewriting Queries

We briefly review the view synchronization process used by the *EVE*-system [NLR98, LKNR98]: Once a view is defined, *EVE* tracks schema changes in all ISs participating in this view. Once a relevant change has been discov-

ered, EVE attempts to find replacements for missing view elements from other ISs based on MISD descriptions and E-SQL view evolution preferences (see Section 3.1). The *EVE* system employs several algorithms for generating such evolved view rewritings under schema changes of underlying ISs, i.e., for achieving view synchronization [LNR97b, NLR98]. In this current paper, we focus on the most powerful and comprehensive algorithm used for view synchronization thus far, which is the *Complex View Synchronization (CVS)* algorithm [NLR98] [2]. CVS handles all common relational capability-changes, such as add-relation, delete-relation, add-attribute, change-relation-name, and so on. Below, we give an example for a rewriting generated by CVS for the *delete-relation* capability change.

After a *delete-relation* capability change on a relation R is detected, the CVS algorithm traverses the information space in order to find possible replacements for those attributes of R that were used by the view V (first step in Figure 3). CVS will find all possible replacements for a missing relation in a given information space[3] using chains of joins to "reach" a candidate replacement relation. This procedure is executed in two steps: Finding candidate replacement relations, and building legal view queries by joining those relations with existing ones.

Example 1 *We define an information space (Meta Knowledge Base) according to Figures 1 and 2. Assume*

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

Figure 1: Information Sources Content Descriptions for Example 1

*the view **Customer-Passengers-Asia** in Equation 2 defines (passenger,participant) pairs of passengers flying to Asia and participants to a tour in Asia that fly and start the tour at the same day, respectively. Such a view could be used to see what participants of a tour are flying to "Asia" on the same day as the tour starts. We now show how to apply the CVS algorithm and find replacements under the "delete relation **Customer**" change for this view.*

[2]E-SQL evolution preferences described in Section 3.1 are used to determine whether the adapted view is considered acceptable to the user. Such a rewritten query is called a *view rewriting*, and if it fulfills certain criteria of correctness [LNR97b], it is called a *legal rewriting*.

[3]For the current paper, we assume that we can replace all missing view elements from the same relation. Extending the optimization for multi-relation replacements is part of our current research.

$\mathcal{JC}$	**Join Constraint**
JC1	**IS1.Customer.Name = IS1.FlightRes.PName**
JC2	**IS2.PreferredCust.PrefName = IS2.Accident−Ins.Holder** AND **IS2.Accident−Ins.Amount** > 100000
JC3	**IS1.Customer.Name = IS3.Participant.Participant**
JC4	**IS3.Participant.TourID = IS3.Tour.TourID**
JC5	**IS1.FlightRes.PName = IS2.Accident−Ins.Holder**
JC6	**IS1.FlightRes.PName = IS2.PreferredCust.PrefName**

Figure 2: Join Constraints for Example 1

```
CREATE VIEW   Customer-Passengers-Asia AS
SELECT        C.Name, C.Age, P.Participant, P.TourID
FROM          Customer C , FlightRes F, Participant P                    (2)
WHERE         (C.Name = F.PName) AND (F.Dest = 'Asia')
              AND (P.StartDate = F.Date) AND (P.Location = 'Asia')
```

The CVS algorithm traverses the information space and constructs chains of joins (according to join constraints given in MISD) 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 **AccidentIns.Age** in relation **Accident−Ins** and join the new table with **FlightRes** using join constraint **JC5** from Figure 2. Then, all view elements (i.e., attributes, WHERE-clauses) that depend on the old relation are replaced by view elements using the new relation. A possible rewriting of query (2) using this substitution is given by query (3).

```
CREATE VIEW   Customer-Passengers-Asia_1 AS
SELECT        AI.Holder, AI.Age, P.Participant, P.TourID
FROM          Accident−Ins AI, FlightRes F, Participant P                (3)
WHERE         (AI.Holder = F.PName) AND (F.Dest = 'Asia')
              AND (P.StartDate = F.Date) AND (P.Location = 'Asia')
```

3.3 Ranking Query Rewritings

Once the CVS algorithm enumerates all possible query rewritings, we need to select one of them. For this, the QC-Model [LKNR98] is used as a metric to compare different legal rewritings by taking both the *quality* and *cost* of a query rewriting into account. 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[4]. With these two dimensions, the QC-Model can *compare* different view queries with each other, even

[4]Cost in this context refers to the long term cost of *incremental view maintenance*, i.e., the cost that is caused by a data update in an underlying information source, and not the immediate and one-time cost caused by updating the view extent after a capability change and the application of our algorithms for fixing the schema.

if they are not equivalent. This comparison is accomplished by assessing five different factors (two quality factors and three cost factors) as explained below [LKNR98].

- **Quality Factors:** Quality refers to the *similarity* (vs. divergence) between an original view and its rewriting.
  - The *Degree of Divergence in Terms of the View Interface* ($\mathcal{DD}_{attr}$) determines how different the view interfaces of the two queries are (expressed numerically by counting the common and non-common attributes in both queries and computing a percentage).
  - The *Degree of Divergence in Terms of the View Extent* ($\mathcal{DD}_{ext}$) 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.
  - *Number of Messages* between data warehouse and information sources—CF_M
  - *Number of Bytes Transferred* through the network—CF_T
  - *Number of I/Os* at the information sources—$CF_{I/O}$

These five factors are normalized and then combined by multiplying them with *trade-off factors* and adding them to a common metric called the *QC-Value*:

$$QC(V_i) = 1 - \Big[\varrho_{quality} \cdot (\varrho_{attr} \cdot \mathcal{DD}_{attr} + \varrho_{ext} \cdot \mathcal{DD}_{ext}) + \varrho_{cost} \cdot (\text{cost}_M \cdot CF_M + \text{cost}_T \cdot CF_T + \text{cost}_{I/O} \cdot CF_{I/O})\Big] \quad (4)$$

with V_i being the view rewriting with the index i; $\mathcal{DD}_{attr}, \mathcal{DD}_{ext}, \mathcal{CF}_{\mathcal{M}}, \mathcal{CF}_{\mathcal{T}}, \mathcal{CF}_{\mathcal{I/O}}$ as defined above; the *trade-off factors* $\varrho_{attr}, \varrho_{ext} \geq 0$; $\varrho_{attr} + \varrho_{ext} = 1$; $\varrho_{quality}, \varrho_{cost} \geq 0$; $\varrho_{quality} + \varrho_{cost} = 1$; and the unit costs $\text{cost}_M, \text{cost}_T, \text{cost}_{I/O} > 0$. The unit costs can be empirically computed for a given data warehouse by a method proposed in [LKNR99], whereas the trade-off factors have to be set according to a user's preferences. This computation returns a numerical value between 0 and 1 that determines the QC-Value of a query rewriting. Since a low degree of divergence as well as low cost are semantically "good", we subtract the factors from 1 and obtain a QC-Value in which 1 describes a perfect query rewriting and 0 a query essentially useless to the user. In the view synchronization process discussed so far, all query rewritings have to be generated first in order to be compared by their QC-Values (cf. Figure 3, second step).

3.4 Complexity of the View Synchronization Process

The basic principle underlying CVS is that a missing view element (relation or attribute) can be replaced by a new element that is connected to the rest of the view query by a *chain* (or *path*) of joins. Finding a replacement for

an attribute involves iterating through the complete information space and finding all possible replacements (i.e., finding a relation containing a replacement and then all possible paths of join constraints between the original and the replacing relations). For each of these paths of joins through the information space, a number of conditions outlined below have also to be met. We will now give a graph-oriented description of this CVS process more formally described in [NLR98]:

- All relations that are in a certain sense "connected" to the original view query (by join constraints, as defined in [LKNR98]) have to be considered for a replacement. Thus, CVS iterates through those relations in the information space. That is, the remaining steps in CVS (explained below) have to be executed up to n_R times.
- For each relation that is considered for replacement, all possible paths of joins that lead from the deleted relation to this relation in the information space have to be found (R-replacement). Since an exhaustive search through the graph is necessary here, this is of complexity $O(n'_R!)$ with n'_R being the number of relations considered (i.e., $n'_R < n$, first step in Figure 3). The algorithm generates an exponential number of possibly inefficient queries.
- For each path found, it has to be decided if the replacement is valid (i.e., satisfies the users' expectations with respect to extent containment specified in E-SQL). This is determined by finding appropriate $\mathcal{PC}$-constraints [NLR97]. The finding of these constraints is of linear complexity in the number of constraints per relation if they are stored in an efficient way (indexed by relation). We also have to check if new WHERE-clauses introduced by the replacement not contradict WHERE-clauses already in the query which is of low polynomial complexity in the number of WHERE-clauses.
- After all (i.e., exponentially many in the number of relations) possible view rewritings have been generated, the QC-Value for each rewriting has to be computed (second step in Figure 3, cf. Section 3.3 and [LKNR98]). The computation of the cost and quality factors involves the estimation of intermediate relation sizes, the computation of local costs, and a summation of the results. This computation is polynomial in the number of relations in the rewriting, but has to be executed for a exponential number of queries, which gives it exponential complexity. Finally, the rewritings found have to be sorted by their QC-Value (third step in Figure 3) in order to present them to the view user.

Our graph oriented description reveals that the most expensive operation in the algorithm above involves finding *all paths* of joins. Also, it is inefficient to generate view rewritings first and then compute QC-Values for each rewriting (i.e., performing view synchronizationin two separate phases).

4 The New Optimized View Synchronization Process

To recall from Section 3.4, the current view synchronization process generates all possible query rewritings V_i for an original query V by considering all paths leading through the information space between the to-be-replaced

relation R and the replacement relation R_i. It then applies the QC-Model to each rewriting V_i and recomputes the QC-Value for this rewriting. We will refer to this computation as QC_{total}:

$$QC_{total}(V_i) = QC(V_i) = f(V, V_i, MKB). \tag{5}$$

with the old view query V, the new view query V_i, and the MKB.

The key to reducing the complexity of the view synchronization process is avoiding the expensive operation of finding *all* paths of joins from the view to a replacing relation by finding only the *minimal* path. This then would reduce the number of generated inefficient rewritings that later would be discarded anyways. To achieve this, we propose to integrate the two separate stages of first query generation and then query evaluation into one tightly integrated algorithm, effectively performing a cost-based search space pruning optimization. Since we will now find (optimal) paths from a deleted relation R to all other relations R_i in the information space that may serve as potential replacements of R in the view query, we will obtain one rewriting only (and thus one QC-Value) for each relation R_i. So we have:

$$QC_{incr}(R_i) = f(V, R, R_i, MKB). \tag{6}$$

Figure 4 contrasts our new approach, referred to as the *Optimized CVS* algorithm, with the original process

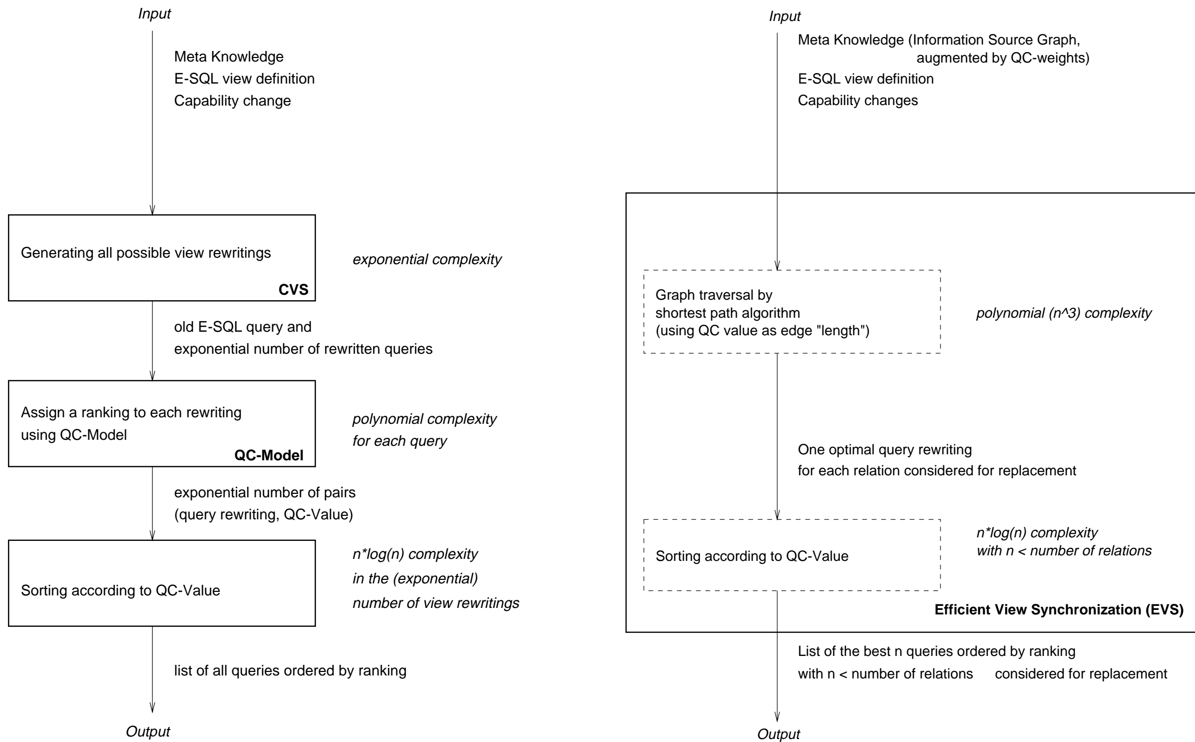


Figure 3: View Synchronization as a Two-Stage Computation Process

Figure 4: View Synchronization as One Integrated Process

in Figure 3, showing its two key advantages: With the new algorithm, only a small number (at most n_R, the number of relations considered for replacements) of queries are ever generated. Furthermore, the QC-Values of

the queries do not have to be computed *after* generating the queries, but they are already determined *during* the rewriting construction process itself.

4.1 Expressing View Synchronization as a Graph Problem

Figure 5 shows an example of mapping our view synchronization problem to a graph representation $G(N, E)$ that we call the *Information Space Graph (IS-Graph)*. We map the relations from the MKB into vertices in N and the join constraints into edges in E, i.e., $N = \{R_i | R_i \in MKB\}$ and $E = \{\mathcal{JC}_{\mathcal{R}\rangle,\mathcal{R}|} | \mathcal{JC}_{\mathcal{R}\rangle,\mathcal{R}|} \in \mathcal{MKB}\}$. Given that mapping, a path through the IS-Graph (from the view query to an end vertex R_i) represents exactly one possible query rewriting V_i that uses the relation R_i for a replacement for missing view elements. Hence, all paths between the vertices representing the original query and each other (reachable) vertex have to be considered as possible replacements.

Figure 5 depicts the IS-Graph for the example information space from Section 3. Assume the relation *Customer* is deleted and we try to use *PreferredCust* as its replacement. In this case, we have two paths (two query rewritings) from the remaining view query (*FlightRes* $\bowtie$ *Participant*) to *Accident-Ins*, namely:

$$\textit{FlightRes} \bowtie_{JC_6} \textit{PreferredCust}$$

$$\text{and} \quad \textit{FlightRes} \bowtie_{JC_5} \textit{Accident-Ins} \bowtie_{JC_2} \textit{PreferredCust}$$

In order to select one of these paths (rewritings), we now apply the QC-Model to compute a numerical measure

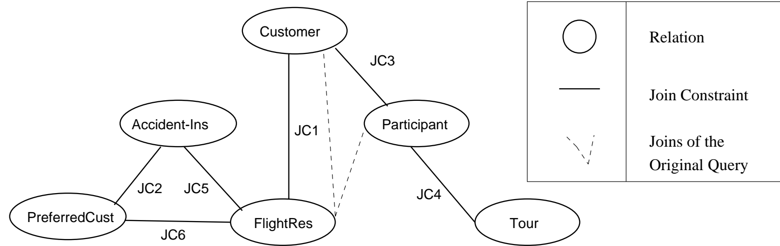


Figure 5: An Example of the Information Space Graph.

of "desirability" (its QC-Value) for each query rewriting.

In order to map the view synchronization problem into a graph problem, we need to integrate the QC-computation into the path finding process. For this, we define a measure for the *length* of a join path that reflects the semantics associated with each path in terms of its quality and cost to serve as replacement chain for a query rewriting. We will assure that the *shortest* path between two relations in this weighted graph will give us the expected result (i.e., the best query rewriting for the given replacement). To accomplish this, we first need to define meaningful semantics for the *weight* of an edge in the IS-Graph (i.e., express the QC-Value by edge weights). We also need to show that computing the QC-Value for a query *incrementally* along a path using these weights (i.e., computing QC_{incr}) yields the same results as computing the QC-Value at once (i.e., computing

QC_{total}).

4.1.1 Augmentation of the IS-Graph with QC-Edge Weights

We label the edges in our graph with parameters that are used to determine the values of the quality and cost factors. A value that is a property of a relation N_R (a vertex in the graph)[5] instead of a pair of relations (i.e., an edge) will be attached to edges adjacent to N_R. The label for an edge E_i is a four-tuple $qc_{E_i} = (\mathcal{DD}_{ext}(\rangle), \mathcal{CF}_{\mathcal{M}}(\rangle), \mathcal{CF}_{\mathcal{T}}(\rangle), \mathcal{CF}_{\mathcal{I/O}}(\rangle))$[6]. With these edge weights, we can now incrementally compute a four-tuple $qc(P_i)$ of numerical values defined on a path $P_i = (E_{l_0}, E_{l_1}, \dots E_{l_i})$ with $E_{l_i} \in E$, which we call the *raw incremental QC-Value*, denoted by:

$$qc(P_i) = (qc_{DD_{ext}}(P_i), qc_{CF_M}(P_i), qc_{CF_T}(P_i), qc_{CF_{I/O}}(P_i)). \tag{7}$$

This raw incremental QC-Value $qc(P_i)$ is computed as follows:

$$\begin{aligned} qc(P_0) &= qc_{E_{l_0}} \\ qc(P_k) &= (qc_{DD_{ext}}(P_k), qc_{CF_M}(P_k), qc_{CF_T}(P_k), qc_{CF_{I/O}}(P_k)) \\ &= (qc_{DD_{ext}}(P_{k-1}) \cdot \mathcal{DD}_{ext}(\updownarrow_{\|}),\ \amalg\rfloor_{\mathcal{CF}_{\mathcal{M}}}(\mathcal{P}_{\|-\infty}) + \mathcal{CF}_{\mathcal{M}}(\updownarrow_{\|}), \\ &\quad\ qc_{CF_T}(P_{k-1}) + CF_T(l_k),\ qc_{CF_{I/O}}(P_{k-1}) + CF_{I/O}(l_k), \end{aligned} \tag{8}$$

for $k \geq 1$ with E_{l_k} being the k-th edge traversed in the path P_i and $l = (l_0, l_1, \dots l_k)$ the sequence of the indices of the edges E_{l_i} that have been traversed for this computation. We motivate these computations (multiplication for $qc_{DD_{ext}}(P_i)$ and addition for the other factors) in our technical report on the QC-Model [LKNR98] (see also Section 4.1.2.

At any vertex N_k in the path, we compute the intermediate QC-Value $QC_{incr_{P_k}}(R_i)$ for original view V and deleted relation R for sub-path P_k traversed for the replacement relation R_i (see also Equation 4):

$$\begin{aligned} QC_{incr_{P_k}}(R_i) &= 1 - \Big[\varrho_{quality} \cdot (\varrho_{attr} \cdot \mathcal{DD}_{attr} + \varrho_{ext} \cdot \amalg\rfloor_{\mathcal{DD}_{\rceil\S\sqcup}}(\mathcal{P}_{\|})) + \\ &\quad \varrho_{cost} \cdot (\text{cost}_M \cdot qc_{CF_M}(P_k) + \text{cost}_T \cdot qc_{CF_T}(P_k) + \text{cost}_{I/O} \cdot qc_{CF_{I/O}}(P_k))\Big]. \end{aligned} \tag{9}$$

Note that $\mathcal{DD}_{attr}$ depends solely on the two relations that mark the end points of the path and therefore is independent of P_k. This explains why we can work with a four- instead of a five-tuple.

The incremental QC-Value for the complete replacement path for R_i, can now be defined as:

$$QC_{incr}(R_i) = QC_{incr_P}(R_i)$$

with P being the complete path to the replacement relation.

---

[5]E.g., a relation size.

[6]The factor $\mathcal{DD}_{attr}$ does not have to be taken into consideration, as explained later in this section

4.1.2 Semantics of Incremental Computation

We now need to show that the computation of the QC-Value for a complete query is equivalent to the incrementally computed QC-Value along the replacement path using the method outlined above. That is, we need to show that

$$QC_{total}(V_i) = QC_{incr}(R_k) \tag{10}$$

for the "best" V_i according to our QC-Model that uses R_k as a replacement. Previously, the QC-Value was computed by the quality and cost formulas given in [LKNR98] (Section 3.3). Since we can apply Equation 9 at any point in the incremental computation, we have to show that the incremental computation of the five factors that contribute to the QC-Value yields the same final result as the total computation. So we must express the computation of each factor in an incremental way, i.e., our approach is to find a way to compute $qc_f(k)$ as

$$qc_f(k) = f(l_0) \odot_f f(l_1) \odot_f \ldots \odot_f f(l_k) \tag{11}$$

with $f \in \{DD_{ext}, CF_M, CF_T, CF_{I/O}\}$ and operations $\odot_f$ on these values (cf. Equation 8). Since this computes the QC-Value from left to right for each factor f, $\odot_f$ has to be shown to be left-associative, i.e.,

$$f(l_0) \odot_f f(l_1) \odot_f f(l_2) \odot_f \ldots \odot_f f(l_k) = (\ldots((f(l_0) \odot_f f(l_1)) \odot_f f(l_2)) \ldots \odot_f f(l_k)) \tag{12}$$

for our path $P_i = (E_{l_0}, E_{l_1}, \ldots, E_{l_i})$. If we can compute all factors incrementally and all four operations $\odot_f$ are left-associative, then the incremental computation of $QC_{incr}(i)$ will deliver the same result as the total computation, i.e., $QC_{incr_n}(R_k) = QC_{total}(V_i)$ for a path with $n+1$ edges[7] that leads to R_k and a view rewriting V_i that uses R_k as replacement relation for R. In order to show these required characteristics for each factor, we now describe the construction of the labels $qc_{E_i} = (\mathcal{DD}_{ext}(\rangle), \mathcal{CF}_{\mathcal{M}}(\rangle), \mathcal{CF}_{\mathcal{T}}(\rangle), \mathcal{CF}_{\mathcal{I/O}}(\rangle))$ for the edges E_i in the IS-Graph.

View interface—$\mathcal{DD}_{attr}$. This quality factor is a function of the original and rewritten view definition only, i.e., it is independent of the *path* of joins that is used to rewrite a query. Therefore this factor does not need to be included in this discussion.

View extent—$\mathcal{DD}_{ext}$. Due to limited space, the derivation of the total computation for $\mathcal{DD}_{ext}$ cannot be repeated here. In [LKNR98] , we compute the size of view extents $|V_i|$ and overlaps $|V_i \cap V_j|$ by multiplying sizes of relations $|R_i|$ used in the view V_i and selectivities js_{R_i,R_j} of the joins between them. The Degree of Divergence is then computed as $\mathcal{DD}_{\rceil\S\sqcup}(\mathcal{V}, \mathcal{V}_\rangle) = \{(|\mathcal{V}|, |\mathcal{V}_\rangle|, |\mathcal{V} \cap \mathcal{V}_\rangle|)$ for a view V and a rewriting V_i. This computation of $|V_i|$ and $|V \cap V_i|$ by multiplying constant factors can be executed incrementally as $qc_{DD_{ext}}(k) = f(qc_{DD_{ext}}(k-1), jc_{R_{l_{k-1}}, R_{l_k}}, |R_{l_k}|)$. We associate $|R_i|$ and jc_{R_i,R_j}, respectively, with the edge E_k between vertices R_i and R_j, together representing the $\mathcal{DD}_{\rceil\S\sqcup}(\|)$ component of qc_{E_i} [8].

[7] We start counting QC_{incr_k} with 0.

[8] The size of the original view $|V|$ is known beforehand and does not have to be computed during the incremental QC-computation.

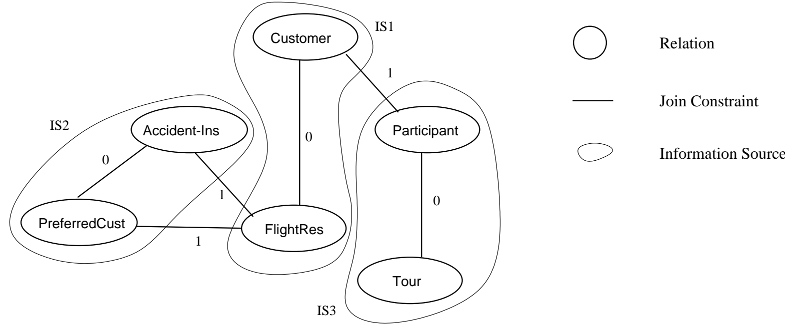


Figure 6: Assigning Weights for CF_M.

If the E-SQL view specification requires the new view to be a superset or subset of the original one, the join selectivity alone is not sufficient for determining if the view rewriting V_i is legal. The selectivity of $\mathcal{PC}$-constraints has to be used in place of the join selectivity. In this case, $\mathcal{PC}$-constraints between each pair of relations (R_i, R_j) on the path to the replacing relation are necessary. Whenever such constraints are not available, we will assign a value of ∞ to $\mathcal{DD}_{ext}$.

Note that $\mathcal{DD}_{ext}$ is computed in a multiplicative way and that the weights are not necessarily larger than 1. For a shortest-path algorithm, this means that the "length" of the complete path does not necessarily increase with the inclusion of a new edge, but that it could also decrease[9]. This computation, a multiplication of rational numbers, is left-associative. This supports that the Degree of Divergence can in fact be incrementally computed as we traverse a path from a deleted relation to its replacement.

Number of messages—CF_M. To compute the number of messages CF_M exchanged between the data warehouse and the underlying information sources incrementally, we assign 1 to an edge if the two relations that it connects are in different information sources and a 0 otherwise ($CF_M(k) = \{0, 1\}$, cf. Figure 6). With these edge weights, an incremental computation as $qc_{CF_M}(k) = qc_{CF_M}(k-1) + CF_M(l_k)$ is possible. The operation is additive, i.e., we can compute an intermediate value for CF_M by adding the current value to a previously computed intermediate value. Since addition is associative, this cost factor is associative.

Number of bytes—CF_T. The number of bytes transferred CF_T is computed by a sum of factors:

$$CF_T(k) = 2 \cdot (\sigma_{IS_{l_1}} \cdot \ldots \cdot \sigma_{IS_{l_k}})(J_{IS_{l_1}} \cdot \ldots \cdot J_{IS_{l_k}}) s_{\Delta R_{out,IS_{l_k}}} \tag{13}$$

with k an index denoting the sequence number of the relation in the path for which CF_T is currently computed, σ_{IS_i} the selectivity of the selection conditions for IS_i, J_{IS_i} the estimated size of a joined relation (computed from join selectivities and relation sizes), and $s_{\Delta R_{out,IS_k}}$ the size (sum of the lengths of attributes in bytes) of a

[9]This indicates the requirement that a shortest-path algorithm for this problem must be able to handle "negative" edge lengths, as we further discussed for our solution in Section 4.2.1.

sub-query to an IS [LKNR98]. The operation is addition, with the summands (from Equation 13) dependent on the previous path through the graph. In order to compute $CF_T(k)$, we need all $CF_T(i)$ for $i = 0 \dots k-1$ for the path $(E_{l_0}, E_{l_1}, \dots, E_{l_{k-1}})$ that led from the starting node to the current edge. If we compute path lengths from the starting node (rather that computing sub-paths at random and adding the results)[10], we can perform this computation incrementally as $qc_{CF_T}(k) = qc_{CF_T}(k-1) + CF_T(l_k)$. This addition is again associative.

Number of I/Os—$CF_{I/O}$. Similarly to the previous case, we compute $CF_{I/O}$ as a sum of several factors depending on the current information source and the relations included in the join [LKNR98] (since the I/O-cost depends on the number of tuples that have to be retrieved from the current relation for an incremental update). So we have $qc_{CF_{I/O}}(k) = qc_{CF_{I/O}}(k-1) + CF_{I/O}(l_k)$. Due to addition, associativity is given.

4.1.3 Equivalence of Incremental and Total Computation

Since we have shown all parameters to be computable in an incremental way and left-associative, this assures that it is possible to compute the QC-Value incrementally, i.e.,

$$QC_{total}(V_i) = QC_{incr}(R_k) \tag{14}$$

for the "best" view rewriting V_i ("best" according to the QC-Model) that uses R_k as the replacement relation.

4.2 Finding the Best View Rewriting: The Shortest Path Approach

We can now to apply a shortest-path-algorithm in order to find the optimal view rewriting V_i for a given replacing relation R_k. One algorithm that matches the requirements identified in the previous section (stability over "negative" edge-lengths and knowledge of the path "history") is the *Bellman-Ford*-algorithm ($\mathcal{BF}$) for the single-source shortest path in a graph [CLR90]. This single source would be our original view query V which can be abstracted as one node in the IS-Graph.

On a graph with n vertices, $\mathcal{BF}$ loops n times over all edges and applies an *edge relaxation* in each iteration. This relaxation step determines whether applying the current edge would decrease the *distance* of a vertex from the original vertex and updates this distance parameter (kept with that vertex) accordingly. $\mathcal{BF}$ finds the *best* path between a source vertex and all other vertices. It returns an ordered set of vertices for a given "destination" vertex R_i, which, in our mapping, represents the "chain" of joins to a relation R_i used to rewrite the given query V after a capability change.

The complexity of $\mathcal{BF}$ is $O(|V| \cdot |E|)$. For a fully connected graph, this is $O(n^3)$ with n being the number of relations considered. For a sparsely connected graph[11], the complexity of this operation is $O(n^2)$. So in at most $O(n^3)$ operations we can compute the best "join chains" for all possible replacing relations. Since $\mathcal{BF}$ computes

[10] In Section 4.2, we confirm that our solution meets this requirement.

[11] A graph $G = (V, E)$ is *sparsely connected* if $|E| \in O(|V|)$.

the shortest paths for all reachable destination relations R_i in the IS-Graph, all optimal view rewritings V_i (one for each replacement relation R_i) can be directly constructed after the algorithm finishes.

4.2.1 "Negative Length" Cycles

Since one of our QC-Model factors ($\mathcal{DD}_{ext}$) can potentially lead to "contracting" edge lengths (meaning that $QC_{incr_k}(R_i) < QC_{incr_{k-1}}(R_i)$ for a given k) it is necessary to look at the possibility of contracting cycles, usually referred to as *negative-length* cycles, in our algorithm. In order to show that the problem of negative length cycles does not occur in our context, we take a closer look at the computation of the quality factor $\mathcal{DD}_{ext}$. Intuitively, a good quality of a view rewriting is given if the extent of the new view V_i is "close" to the extent of the old view V. The quality, i.e., the Degree of Divergence of Extents $\mathcal{DD}_{ext}$, is defined as a weighted sum of the relative amounts of missing tuples $\mathcal{DD}_{]\S\sqcup_\mathcal{D}\infty}(\mathcal{V}_\rangle)$ and surplus tuples $\mathcal{DD}_{]\S\sqcup_\mathcal{D}\in}(\mathcal{V}_\rangle)$ between the old and the new view [LKNR98]. The objective for obtaining a small Degree of Divergence is to minimize the two ratios:

$$\mathcal{DD}_{]\S\sqcup_\mathcal{D}\infty}(\mathcal{V}_\rangle) = \frac{|\mathcal{V} \setminus \mathcal{V}_\rangle|}{\mathcal{V}} \quad and \quad \mathcal{DD}_{]\S\sqcup_\mathcal{D}\in}(\mathcal{V}_\rangle) = \frac{|\mathcal{V}_\rangle \setminus \mathcal{V}|}{\mathcal{V}_\rangle}$$

since the Degree of Divergence, computed as $\mathcal{DD}_{ext}(\mathcal{V}_\rangle) = \varrho_\infty \cdot \mathcal{DD}_{]\S\sqcup_\mathcal{D}\infty}(\mathcal{V}_\rangle) + \varrho_\in \cdot \mathcal{DD}_{]\S\sqcup_\mathcal{D}\in}(\mathcal{V}_\rangle)$ with $\varrho_1, \varrho_2 \geq 0$ and $\varrho_1 + \varrho_2 = 1$, has to be minimized.

Note that this value cannot become negative. If a contracting cycle occurs in the IS-Graph, this can be caused only by the Degree of Divergence being lowered with each loop through the cycle (since this is the only one of the five cost factors that possibly could decrease the QC-Value, while the other factors except $\mathcal{DD}_{\dashv\sqcup\sqcup\nabla}$ are guaranteed to increase the QC-Value with each addition of an edge). This would mean that the quality of the view rewriting becomes *better* with the inclusion of another relation and join in the view query, which is unlikely. Also, since the Degree of Divergence has the lower limit of 0 and we have a discrete domain (the number of tuples in a relation is a natural number, i.e., the function $f : \{qc_{DD_{ext}}(l_i)\} \rightarrow \mathcal{DD}_{]\S\sqcup}$ cannot be asymptotic), $\mathcal{DD}_{]\S\sqcup}$ can not be contracting infinitely. After a finite number of loops through this contracting cycle, the Degree of Divergence cannot become lower (we would have a "perfect" rewriting, preserving all attributes and tuples from the original query). The remaining factors CF_M, CF_T and $CF_{I/O}$ have been increasing all along, which causes $\mathcal{DD}_{]\S\sqcup}$ to increase. For this reason, a shortest path algorithm would not get "caught" in an infinite loop for our problem. We note that the original $\mathcal{BF}$ (well-known from the literature on algorithms [CLR90]) does not pursue such loops through the graph, but rather terminates that particular path. If one would want to make use of those cycle-based solutions, unlikely to occur in a real situation, slight adaptations to the algorithm would be needed.

5 Conclusion

View synchronization addresses an important new problem in dynamic distributed information systems [RLN97, LNR97a, NLR97, LNR97b, NLR98, LKNR98]. In this present paper, we show the complexity of the view synchronization process based on two separate phases (view rewriting [NLR98] and QC-computation [LKNR98])

to be $O(n!)$ in the number of relations in the information space. This implies that the original view synchronization process is too inefficient to be practically viable for very large information spaces. In this paper, we developed a solution based on integrating the two phases into one process that is capable of discarding inferior solutions without first having to enumerate them. This reduces the complexity to polynomial ($O(n^3)$), making view synchronization now efficient even for large information systems.

A prototype of the *EVE*-system has been implemented and is fully functional. It has been successfully demonstrated at the IBM technology showcase during the CASCON '97 conference [LNR97a], and will be available on our *EVE* project web page soon [12]. A new view synchronization optimizer designed based on the basic concepts outlined in this paper is currently being implemented in the *EVE*-system. This new implementation of EVE will allow us to efficiently handle schema evolution in a distributed data warehousing environment, which is a significant improvement over current technology handling only data updates at the underlying information sources [GMR95, vdBK94, AAS97, BLT86].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular, we are grateful to Xin Zhang, Yong Li, and Amber Van Wyk for implementing several of the major components of the EVE system, and Anişoara Nica and Amy Lee for their work on the foundations of view synchronization and their valuable suggestions and discussions in the current context.

References

[AAS97] D. Agrawal, A. El Abbadi, and A. Singh. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

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

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Cambridge, The MIT Press, 1990.

[CTL+96] L.S. Colby, T.Griffin, L.Libkin, I.S.Mumick, and H.Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinition. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 211–222, 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[LKNR98] 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, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[LKNR99] 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*, Accepted as poster paper, 1999.

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

[LNR97a] A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proceedings of IBM Centre for Advanced Studies Conference CASCON97, Best Paper Award*, pages 1–14, November 1997.

[12] `http://davis.wpi.edu/dsrg/EVE`

[LNR97b] 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.

[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.

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

[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.

[vdBK94] C. A. van den Berg and M.L. 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.

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

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

[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*, December 1996.