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), causing a data warehouse to become undefined. This process to evolve (rewrite) view queries after capability changes of ISs is referred to as view synchronization. Current view synchronization algorithms generate a potentially large number of valid solutions for the rewriting of a view query and according to our analysis in this paper have high complexity (in $O(n!)$). We 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, which can be solved with $O(n^3)$ complexity using the Bellman-Ford algorithm.

Keywords: Evolvable view environment, data warehouse, cost model, shortest path problem.

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). Besides providing simplified information access to customers without the necessary technical background, materialized views also offer higher availability and query performance.

In our recent work we study the maintenance of data warehouses defined over distributed *dynamic* information sources. 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* and which is accomplished through several view synchronization algorithms. 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 [8]. 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, which is done through the *QC-Model*[1] [5].

In this paper, we now examine the *complexity* of this view synchronization process [6, 10]. We identify that its most powerful algorithm has a high complexity (in $O(n!)$). 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, 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 phase of 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 remainder of this paper is organized as follows. Section 2 reviews related work, while Section 3 reviews EVE background, the CVS algorithm, and the QC-model. Section 4 proposes the new Optimized CVS algorithm. We conclude this work in Section 5.

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

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) [9], 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 [10, 6], 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 [8] generates a large number of alternative legal rewritings, which have to be evaluated and compared. In [5], this need was addressed by establishing a model for systematically ranking solutions for view synchronization based on the two dimensions of quality and maintenance costs.

Levy et al. [7] 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, while van den Berg et al. [11] and Agrawal et al. [1] 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. *Incremental view maintenance* has been an active area of research [3] but is limited to data level changes. We are concerned with query rewriting without equivalence and view maintenance under schema changes.

3 The View Synchronization Process

3.1 Non-Equivalent View Synchronization

In this section, we briefly review the concepts of the *Evolvable View Environment* (*EVE*) [6, 10, 8] system as needed for the remainder of this paper. *EVE* evolves views in the presence of IS capability changes.

E-SQL or *Evolvable-SQL* is an extension of SQL that allows the view definer to express preferences for view evolution [10] by defining what information is dispensable, replaceable (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 flexibility to evolve views under schema changes while preserving the user's intended semantics.

In order to enable view synchronization, our system needs 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, stored in the *Meta Knowledge Base (MKB)*). 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 $\mathcal{PC}$-*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.

3.2 The CVS Algorithm for Rewriting Queries

We briefly review the view synchronization process used by the *EVE*-system [8, 5]: Once a view is defined, *EVE* tracks schema changes in all ISs participating in this view and attempts to find replacements for missing view elements using MISD and E-SQL. The *EVE* system employs several algorithms for generating view rewritings, i.e., for achieving view synchronization [10, 8]. In this current paper, we focus on the most comprehensive algorithm thus far, which is the *Complex View Synchronization (CVS)* algorithm [8] [2]. CVS handles all common relational capability-changes (e.g., add, delete, rename). We now give an example for a rewriting generated by CVS.

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. 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. We consider the view* ***Customer-Passengers-Asia*** *in Equation 1 and show how to apply the CVS algorithm and find replacements 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,Birthday)** **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 1

$\mathcal{JC}$	Join Constraint
1	**IS1.Customer.Name = IS1.FlightRes.PName**
2	**IS2.PreferredCust.PrefName = IS2.Insurance.Holder** AND **IS2.Insurance.Amount** > 100000
3	**IS1.Customer.Name = IS3.Participant.Name**
4	**IS3.Participant.TourID = IS3.Tour.TourID**
5	**IS1.FlightRes.PName = IS2.Insurance.Holder**
6	**IS1.FlightRes.PName = IS2.PreferredCust.PrefName**

Figure 2: Join Constraints for Example 1

[2] E-SQL evolution preferences are used to determine whether the adapted view query is considered acceptable to the user. Such a rewritten query is called a *view rewriting*, and if it fulfills certain criteria of correctness [10], 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.

```
CREATE VIEW   Customer-Passengers-Asia AS
SELECT        C.Name, C.Age,
              P.Name, P.TourID
FROM          Customer C , FlightRes F,
              Participant P                       (1)
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 uses MISD join constraints 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 join constraint **JC5** 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 query (1) using this substitution is given by query (2).

```
CREATE VIEW   Customer-Passengers-Asia_1 AS
SELECT        I.Holder, AI.Age,
              P.Name, P.TourID
FROM          Insurance I, FlightRes F,
              Participant P                       (2)
WHERE         (I.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, using the QC-Model [5] as a metric (taking both the *quality* and *cost* 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 if they are not equivalent. This comparison is accomplished by assessing five different factors as explained below [5].

- **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 numerically how different the view interfaces of the two queries are.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. The factors are *Number of Messages*(CF_M) between data warehouse and information sources, *Number of Bytes Transferred*(CF_T) through the network, and *Number of I/Os*($CF_{I/O}$) at the ISs.

[4] long term cost of *incremental view maintenance*

Normalizing and then combining these factors yields the *QC-Value*:

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

with V_i the view rewriting with the index i; $\mathcal{DD}_{attr}$, $\mathcal{DD}_{ext}$, CF_M, CF_T, $CF_{I/O}$ 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 cost_M, cost_T, $\text{cost}_{I/O} > 0$. The unit costs can be empirically computed for a given data warehouse by a method proposed in [5], whereas the trade-off factors have to be set by the user. The QC-value evaulates a rewriting and is a real number between 0 (bad) and 1 (good). Note that 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.

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 [8]:

- All relations that are in a certain sense "connected" to the original view query (by join constraints, as defined in [5]) have to be considered for a replacement. Thus, CVS iterates through those relations in the information space.

- 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., $n'_R < n$). The algorithm generates an exponential number of possibly inefficient queries.

- For each path found, it has to be decided if the replacement conplies with E-SQL preferences. This is determined by finding appropriate $\mathcal{PC}$-constraints [8] which is of linear complexity in the number of constraints per 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 has to be computed [5]. The computation of the cost and quality factors is polynomial in the number of relations in the rewriting, but has to be executed for a exponential number of queries. Finally, the rewritings found have to be sorted by their QC-Value in order to present them to the view user.

Our analysis 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., two separate phases).

4 The Optimized View Synchronization Process

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}:

$$QC_{total}(V_i) = QC(V_i) = f(V, V_i, MKB). \quad (3)$$

with the old (V) and new view query (V_i) and the MKB.

The key to reducing the complexity of this process is to avoid the expensive operation of finding *all* paths of joins from the view to a replacing relation by finding only the *minimal* path, reducing the number of generated inefficient rewritings. To achieve this, we propose to integrate the two separate stages of query generation and query evaluation into one tightly integrated algorithm, effectively performing a cost-based search space pruning optimization. Since we will now find (optimal) paths from R to all R_i in the information space, we will obtain one rewriting only (and thus one QC-Value) for each relation R_i:

$$QC_{incr}(R_i) = f(V, R, R_i, MKB). \quad (4)$$

We term our new approach the *Optimized CVS* algorithm, having two key advantages: Only a small number (at most n_R, the number of relations considered for replacements) of queries are ever generated, and 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.

4.1 View Synchronization as a Graph Problem

Figure 3 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 $E = \{\mathcal{JC}_{R_i,R_j} | \mathcal{JC}_{R_i,R_j} \in MKB\}$. 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.

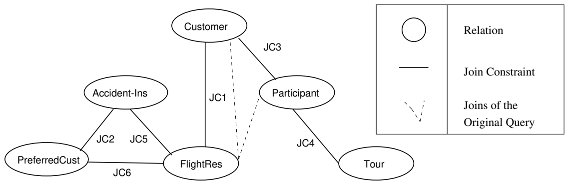

Figure 3: An Example of the Information Space Graph.

Figure 3 depicts the IS-Graph for the example information space from Section 3. Deleting *Customer* and replacing it by *PreferredCust* yields two paths (two query rewritings) from the remaining view query (*FlightRes* $\bowtie$ *Participant*) to *Accident-Ins*, namely:

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

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

In order to express our problem in terms of a graph, we integrate the QC-computation into the path finding process. We define the *length* of a join path expressing quality and cost and 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). We define meaningful semantics for the *weight* of an edge in the IS-Graph (i.e., express the QC-Value by edge weights) and 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 Augmenting the IS-Graph with 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 $qc_{E_i} = (\mathcal{DD}_{ext}(i), CF_M(i), CF_T(i), CF_{I/O}(i))$[6]. With these edge weights, we can now incrementally compute a four-tuple $qc(P_i)$ of numerical values defined on a path $P_i = (E_{l_0}, E_{l_1}, \ldots E_{l_i})$ with $E_{l_i} \in E$, which we call the *raw incremental QC-Value*, denoted by:

$$\begin{aligned} qc(P_i) &= qc_{DD_{ext}}(P_i), qc_{CF_M}(P_i), \\ &\quad qc_{CF_T}(P_i), qc_{CF_{I/O}}(P_i)). \end{aligned} \quad (5)$$

This value $qc(P_i)$ is computed as follows:

$$\begin{aligned} qc(P_0) &= qc_{E_{l_0}} \\ qc(P_k) &= (qc_{DD_{ext}}(P_{k-1}) \cdot \mathcal{DD}_{ext}(l_k), \\ &\quad qc_{CF_M}(P_{k-1}) + CF_M(l_k), \\ &\quad qc_{CF_T}(P_{k-1}) + CF_T(l_k), \\ &\quad qc_{CF_{I/O}}(P_{k-1}) + CF_{I/O}(l_k)) \end{aligned} \quad (6)$$

[5] A typical example would be a relation size.

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

for $k \geq 1$ with E_{l_k} being the k-th edge traversed in the path P_i and $l = (l_0, l_1, \ldots l_k)$ 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 [5].

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 3)[7]:

$$QC_{incr_{P_k}}(R_i) = 1 - \Big[\varrho_{quality} \cdot (\varrho_{attr} \cdot \mathcal{DD}_{attr} + \varrho_{ext} \cdot qc_{DD_{ext}}(P_k)) + \varrho_{cost} \cdot (\text{cost}_M \cdot qc_{CF_M}(P_k) + \text{cost}_T \cdot qc_{CF_T}(P_k) + \text{cost}_{I/O} \cdot qc_{CF_{I/O}}(P_k))\Big] \quad (7)$$

We can now define $QC_{incr}(R_i) = QC_{incr_P}(R_i)$ with P being the path to the replacement relation.

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 as outlined above. That is, we need to show that

$$QC_{total}(V_i) = QC_{incr}(R_k) \quad (8)$$

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 [5] (Section 3.3). Since we can apply Equation 7 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

$$qc_f(k) = f(l_0) \odot_f f(l_1) \odot_f \ldots \odot_f f(l_k) \quad (9)$$

with $f \in \{DD_{ext}, CF_M, CF_T, CF_{I/O}\}$ and operations $\odot_f$ on these values (cf. Equation 6). 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.,

$$f(l_0) \odot_f f(l_1) \odot_f f(l_2) \odot_f \ldots \odot_f f(l_k) = (\ldots((f(l_0) \odot_f f(l_1)) \odot_f f(l_2)) \ldots \odot_f f(l_k))$$

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[8] 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 $qc_{E_i} = (\mathcal{DD}_{ext}(i), CF_M(i), CF_T(i), CF_{I/O}(i))$ 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 [5] , 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 $\mathcal{DD}_{ext}(V, V_i) = f(|V|, |V_i|, |V \cap V_i|)$ for a view V and a rewriting V_i. This computation 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 $\mathcal{DD}_{ext}(k)$ component of qc_{E_i} [9]. Note that $\mathcal{DD}_{ext}$ is computed in a multiplicative way and that the weights are not necessarily larger than 1 [10]. This computation, a multiplication of rational numbers, is left-associative and can thus be incrementally computed.

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 4). With these

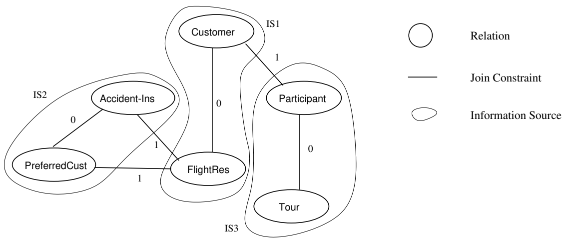

Figure 4: Assigning Weights for CF_M.

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 incrementally and this cost factor is associative.

Number of bytes—CF_T. The number of bytes transferred CF_T is computed by a sum of factors as $CF_T(k) = 2 \cdot (\sigma_{IS_{l_1}} \cdot \ldots \cdot \sigma_{IS_{l_k}})(J_{IS_{l_1}} \cdot \ldots \cdot J_{IS_{l_k}}) s_{\Delta R_{out,IS_{l_k}}}$ 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 sub-query to an IS [5]. The operation is addition (which is assosiative), with the summands dependent on the previous

[7] 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.

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

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

[10] For our solution to this problem, please refer to our TR [4].

path through the graph. In order to compute $CF_T(k)$, we need all $CF_T(i)$ for $i = 0 \ldots k-1$ for the path $(E_{l_0}, E_{l_1}, \ldots, E_{l_{k-1}})$ 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), we can perform this computation incrementally as $qc_{CF_T}(k) = qc_{CF_T}(k-1) + CF_T(l_k)$.

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 [5] (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.,

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

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 apply a shortest-path-algorithm in order to find the optimal view rewriting V_i for a given replacing relation R_k. The *Bellman-Ford*-algorithm ($\mathcal{BF}$) [2] matches the requirements identified (stability over "negative" edge-lengths and knowledge of the path "history"). $\mathcal{BF}$ 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. So in at most $O(n^3)$ operations we can compute the best "join chains" for all possible replacing relations.

5 Conclusion

View synchronization addresses an important new problem in dynamic distributed information systems [10, 6, 8, 5]. In this present paper, we show the complexity of the view synchronization process based on two separate phases (view rewriting [8] and QC-computation [5]) to be $O(n!)$ in the number of relations in the information space. This 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 was implemented and is fully functional. It has been successfully demonstrated at the IBM technology showcase during CASCON'97 [6], and will be available on our *EVE* project web page soon [11].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions, contributions, and feedback on this research. In particular, we are grateful to Anişoara Nica, Amy Lee, Xin Zhang, Yong Li and Amber Van Wyk.

References

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

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

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

[4] A. Koeller, E. A. Rundensteiner, and N. Hachem. Integrating the Rewriting and Ranking Phases of View Synchronization. Technical Report WPI-CS-TR-98-23, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[5] 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, WPI, Dept. of CS, 1998.

[6] A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proc. of IBM CASCON97*, pages 1–14, November 1997.

[7] A. Levy, A. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proc. of ACM PODS*, pages 95–104, May 1995.

[8] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proc. of EDBT*, pages 359–373, Valencia, Spain, March 1998.

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

[10] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proc. of KRDB*, pages 13.1–13.11, Athens, Greece, August 1997.

[11] 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*, pages 449–473. Morgan Kaufmann Pub., 1994.

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