WPI-CS-TR-98-23 October 1998. Revised December 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. The algorithm has to be adapted to accommodate contracting cycles in graphs occuring in our context. We show the optimality and termination of our algorithm even under that condition. 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 schema change in a data warehouse, 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 proposed 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 finding of rewritings instead of executing two separate phases of view synchronization. We give proofs that this mapping to a graph is correct and the algorithm gives the optimal solution in our framework. We also solve an occuring problem with contracting (negative-length) cycles in the weighted graph. We term our solution the *GRASP* view synchronization algorithm (for *graph-based shortest-path*) and show that the algorithm has a complexity in $O(n^3)$.

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

A preliminary version of this work has been published in the *1998 Workshop on Data Warehouses and OLAP* [KRH98]. The contributions of that work were: 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 idea of solving the mapped graph problem by a shortest-path graph algorithm.

In the current work, we have made numerous extensions to the previous version. We completed the treatment of the solution, i.e., the GRASP view synchronization algorithm. We give a proof for the correctness and applicability of the approach; we identify the problem of negative-length cycles in the mapped graph; and we solve this problem by adapting the Bellman-Ford-Algorithm. We prove the termination of the adapted Bellman-Ford-Algorithm even under the condition of contracting cycles in our graph, and we prove the optimality of the solution obtained by the adapted algorithm.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces background material on view synchronization, including a review of the CVS algorithm and the computation of QC-values and examines the complexity of the (previous) algorithm. Section 4 proposes the new GRASP view synchronization algorithm. We also prove the correctness of the new algorithm and show its implications for view synchronization. We conclude this work in Section 6.

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. In the current paper, we now present an optimization integrating both previous results[NLR98, LKNR98] with each other into one solution.

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. An overview over the architecture of the current EVE-system is given in Figure 1. Views are maintained in a data warehouse which is defined over distributed *Information Sources (IS).* Each IS is connected to our system by a *wrapper* which provides a uniform interface to *EVE.* In addition to a traditional data warehouse, we record meta knowledge about information sources in the *Meta Knowledge Base (MKB).* Using the wrappers and this information, the modules performing *View Evolution* can rewrite views (*View Synchronizer*), compute their quality and cost (*QC-Value Computation*), adapt view extents after schema changes (*View Adaptor*) and incrementally maintain views under concurrent data updates (*Concurrent View Maintainer*). A *View Knowledge Base* stores view definitions which can be changed (*synchronized*) by the modules connected.

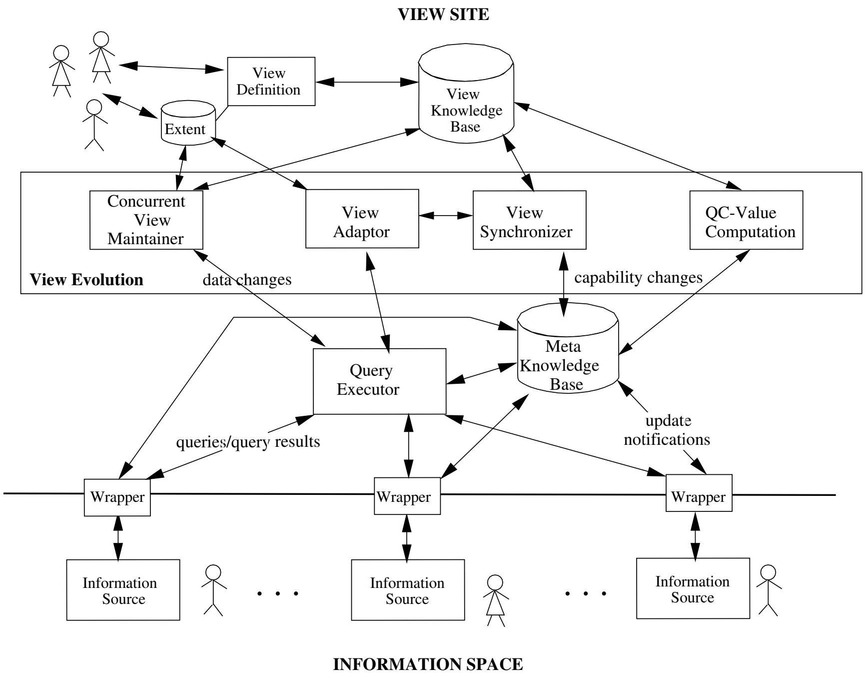

Figure 1: The *EVE* Framework: View Synchronization in an Evolving Environment.

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. E-SQL evolution preferences described in Section 3.1 are used to determine whether the adapted view is considered acceptable to the user. For the purpose of this paper, the only SQL-extension we will take a closer look at is the concept of the *view-extent* parameter $\mathcal{VE} \in \{\subseteq, \supseteq, \equiv, \approx\}$. $\mathcal{VE}$ expresses the relationship between the original and rewritten query as required by the view definer. For instance, $\mathcal{VE} =$"$\subseteq$" requires any query rewriting V_i for the current view V to compute a subset of the original view extent (i.e., $V_i \subseteq V$). The value $\mathcal{VE} =$"$\approx$" means no restrictions for the extent are given. For more details, 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:

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

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 MKB (See Figure 1).

3.2 Generating Query Rewritings: The CVS Algorithm

We briefly review the view synchronization process used by the *EVE*-system [NLR98]. Once a view is defined, *EVE* tracks schema changes in all ISs participating in this view. Once a relevant change has been discovered, *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 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]. A query rewritten by an algorithm like CVS is called a *view rewriting*, and if it fulfills certain criteria of correctness [LNR97b], it is called a *legal rewriting*. 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 5). CVS will find all possible replacements for a missing relation in a given information space using chains of joins to "reach" a candidate replacement relation. 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. The procedure of finding replacements 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 2 and 3. Assume 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*

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 2: Information Sources Content Descriptions for Example 1

$\mathcal{JC}$	**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 3: Join Constraints for Example 1

show how to apply the CVS algorithm and find replacements under the "delete relation ***Customer****" change for this view.*

```
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 3. 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: The QC-Model

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[2]. 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 (two quality factors and three cost factors) as explained below [LKNR98].

- **Quality Factors**
  The quality of a view rewriting refers to a well defined *similarity* between an original view and its rewriting.
  - View interface—$\mathcal{DD}_{attr}$
    The *Degree of Divergence in Terms of the View Interface* determines how different the view interface of two queries are. This can be expressed numerically by counting the common and non-common attributes in both queries and computing a percentage. We will abstract from further details and refer to [LKNR98]. This quality factor is a function of the original and rewritten views and thus is independent from the *path* of joins that is used to rewrite a query. Therefore it will not be included in the further discussion.
  - View extent—$\mathcal{DD}_{ext}$ (Section 3.3.2)
    A query rewriting using information from different sources will in general produce a view extent different from the original one. Only a part of the original will be retained in the new view extent, and new tuples that haven't existed in the previous view may be added to the extent. The *Degree of Divergence in Terms of the View Extent* is determined by the relative numbers of overlapping and additional tuples in a view rewriting.
- **Cost Factors**
  Cost factors are measured for the cost of *incremental view maintenance*. We look at the (long-term) cost that future incremental view maintenance will cause after the view has been rewritten and the extent has been updated. The computation of cost is done for a single data update in one relation in the information source.
  - Number of messages—CF_M (Section 3.3.4)
    This factor refers to the number of messages that have to be exchanged between the data warehouse and information sources. One message has to be sent for each information source referenced in the query. So the number of messages is a function of the number of information sources used in the view rewriting.
  - Number of bytes—CF_T (Section 3.3.5)
    This refers to the total number of bytes that have to be transferred in our system in order to incrementally maintain the data warehouse extent after a data update in one of the sources. This number is a function of the following parameters (in the notation of [LKNR98]): selectivities of local conditions σ, sizes of relations $|R|$, join selectivities between relations js, number of relations in each information source $n_R(i)$, sizes of attributes s).
  - Number of I/Os—$CF_{I/O}$ (Section 3.3.6)
    This is the total number of I/O-operations on all information sources that are caused by the incremental

[2] 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.

view maintenance after a data update. It is a function of the number of relations visited when computing incremental updates and the join selectivities between relations (in order to determine the number of tuples retrieved).

We will now describe some necessary background for the ranking of rewritings and then go into more detail as we explain the computation of each of the quality and cost factors introduced.

3.3.1 Common Subset of Attributes

In order to present our findings on the view extent factor $\mathcal{DD}_{ext}$, we first introduce a notation for *common subset of attributes* and *common-subset-of-attributes-equivalence.*

Definition 1 Common Subset of Attributes of V with respect to V_i. *Let V and V_i be two view definitions, such that $Attr(V) \cap Attr(V_i) \neq \emptyset$. We use $V^{\pi(V_i)}$ to denote the projection of relation V on the common attributes of V and V_i. That is, $V^{\pi(V_i)} = \pi_{Attr(V) \cap Attr(V_i)} V$. Similarly, $V_i^{\pi(V)}$ is defined as $\pi_{Attr(V) \cap Attr(V_i)} V_i$.*

Besides considering the attributes preserved in the legal rewritings, the *sets of tuples* returned by the queries will also have an impact on the user's satisfaction with a view rewriting V_i. When the view interfaces of a legal rewriting V_i and the original view V are not the same, the extent preservation evaluation is done by comparing tuples on the common subset of attributes only. When the view interfaces of V_i and V are the same, the extent comparison is done as usual. When V_i and V have different view interfaces, we refer to V_i as being *common-subset-of-attributes-equal* to V, denoted by $V_i =_\pi V$, if their projections on the common subset of attributes are equal.

Other Boolean ($\subseteq_\pi$) and set ($\cap_\pi, \setminus_\pi$) operations between the view extents of V_i and V can be similarly defined on the common subset of attributes of V_i and V. We summarize these operations and their semantics in Figure 4.

Operator	Semantics
$V =_\pi V_i$	$\forall\, t \in V_i, \exists\, t_i \in V$ s.t. $t[Attr(V) \cap Attr(V_i)] = t_i[Attr(V) \cap Attr(V_i)]$ and $\forall\, t_i \in V_i, \exists\, t \in V$ s.t. $t_i[Attr(V) \cap Attr(V_i)] = t[Attr(V) \cap Attr(V_i)]$
$V_i \subseteq_\pi V$	$\forall\, t_i \in V_i, \exists\, t \in V$ such that $t_i[Attr(V) \cap Attr(V_i)] = t[Attr(V) \cap Attr(V_i)]$
$V \cap_\pi V_i$	$\{z \mid \exists\, t \in V \wedge \exists\, t_i \in V_i,\ z = t[Attr(V) \cap Attr(V_i)] = t_i[Attr(V) \cap Attr(V_i)]\}$
$V \setminus_\pi V_i$	$\{z \mid \exists\, t \in V, z = t[Attr(V) \cap Attr(V_i)] \wedge \nexists\, t_i \in V_i,\ z = t_i[Attr(V) \cap Attr(V_i)]\}$

Figure 4: Operations on the Common Subset of Attributes of V and V_i.

3.3.2 Metric of Quality: Degree of Divergence on the View Extent($\mathcal{DD}_{ext}$)

Below, we discuss the concept of a metric for information preservation for view rewritings. For this paper, we do not need details on the Degree of Divergence on the view interface ($\mathcal{DD}_{attr}(V_i)$), therefore, we will concentrate on the Degree of Divergence on the extent ($\mathcal{DD}_{ext}(V_i)$).

The view extent of a legal rewriting V_i may diverge from the original view extent of V in two aspects:

D1. The (relative) number of tuples from the original V that are not preserved in the new V_i, denoted by

$$DD_{ext\_D1}(V_i) = 1 - \frac{|V_i \cap_\pi V|}{|V^{\pi(V_i)}|} = \frac{|V \setminus_\pi V_i|}{|V^{\pi(V_i)}|} \tag{4}$$

D2. The (relative) number of surplus tuples in the new view V_i that are not in the original view V, denoted by

$$DD_{ext\_D2}(V_i) = \frac{|V_i \setminus_\pi V|}{|V_i^{\pi(V)}|} = \frac{|V_i^{\pi(V)}| - |V \cap_\pi V_i|}{|V_i^{\pi(V)}|} \tag{5}$$

with $V^{\pi(V_i)}$ and $V_i^{\pi(V)}$ as defined in Definition 1, page 7.
Intuitively, the number of tuples that are *not preserved* ($DD_{ext\_D1}(V_i)$) has to be seen in relation to the size of the original view extent $|V^{\pi(V_i)}|$, whereas the number of extra tuples coming into the new view ($DD_{ext\_D2}(V_i)$) is seen relative to the size of the *new* view extent $|V_i^{\pi(V)}|$.

The total extent divergence of V_i from V is the weighed sum of $DD_{ext\_D1}(V_i)$ and $DD_{ext\_D2}(V_i)$, denoted by $DD_{ext}(V_i)$, and is defined as follows:

$$\begin{aligned} DD_{ext}(V_i) &= \varrho_1 \cdot DD_{ext\_D1}(V_i) + \varrho_2 \cdot DD_{ext\_D2}(V_i) \\ &= \varrho_1 \cdot \frac{|V^{\pi(V_i)}| - |V_i \cap_\pi V|}{|V^{\pi(V_i)}|} + \varrho_2 \cdot \frac{|V_i^{\pi(V)}| - |V \cap_\pi V_i|}{|V_i^{\pi(V)}|} \\ &= 1 - \frac{(\varrho_1 |V_i^{\pi(V)}| + \varrho_2 |V^{\pi(V_i)}|) \cdot |V_i \cap_\pi V|}{|V^{\pi(V_i)}| \cdot |V_i^{\pi(V)}|} \end{aligned} \tag{6}$$

where $\varrho_1,\ \varrho_2$ are the trade-off parameters between $DD_{ext\_D1}(V_i)$ and $DD_{ext\_D2}(V_i)$ ($\varrho_1,\ \varrho_2 \geq 0$ and $\varrho_1 + \varrho_2 = 1$). Again, the view definer is given an opportunity to set the trade-off parameters, with the default setting being $(\varrho_1, \varrho_2) = (0.5, 0.5)$.

When the view definer sets the view-extent parameter $\mathcal{VE}$ = '$\equiv$', no view extent divergence is allowed, i.e., $DD_{ext\_D1}(V_i)$ and $DD_{ext\_D1}(V_i)$ must be both zero. When $\mathcal{VE}$ = '$\supseteq$', the first number ($DD_{ext\_D1}(V_i)$) has to be zero for a rewriting to be legal (since all objects of V must also appear in the new view V_i). Therefore, we only have to compute the second measure in order to know the QC-Value of V_i in terms of view extent information preservation. On the contrary, when $\mathcal{VE}$ = '$\subseteq$', the second measure returned is always zero. Therefore, we only need to compute the first measure. Whenever the view-extent parameter $\mathcal{VE}$ ='$\approx$', both numbers have to be computed (and for meaningful results we should have $0 < \varrho_1, \varrho_2 < 1$). For $\mathcal{VE}$ = '$\supseteq$' and $\mathcal{VE}$ = '$\subseteq$', none of the expensive set intersection operations is required. For these cases, the degree of divergence can be computed by counting the numbers of tuples in the legal rewriting V_i and the original view V (since either $V \subset_\pi V_i$ or $V_i \subset_\pi V$, so the size of the intersection is equal to the size of the smaller view extent).

3.3.3 Estimation of $|V_i|$ and $|V \cap_\pi V_i|$

On order to compute $\mathcal{DD}_{ext}(V_i)$, we need to be able to estimate $|V \cap_\pi V_i|$. The size of the new view $|V_i|$ and the size of the overlap $|V \cap_\pi V_i|$ could be measured with an SQL-Query after generation of the new view, but this would require to temporarily store the old view and is also computationally expensive. Thus, we estimate those values with the meta knowledge available in the *EVE* system. In order to estimate the size of the view $|V_i|$, we use the join selectivities of available join constraints. With the join selectivity of a join between the relations R_1 and R_2 on the attributes specified in the join constraint defined as

$$js_{R_1,R_2} = \frac{|R_1 \bowtie R_2|}{|R_1 \times R_2|}$$

and a view rewriting $V_i = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, we estimate its size as

$$|V_i| \approx |R_1| \cdot js_{R_1,R_2} \cdot |R_2| \cdot js_{R_2,\ldots} \cdot \ldots \cdot js_{\ldots,R_n} \cdot |R_n|.$$

If the view query has additional local conditions, their selectivity is multiplied with the result.

As explained in detail in [LKNR98], the size of the view overlap can be determined from the sizes of the underlying relations by

$$|V \cap_\pi V_i| = \sigma_{V,R} \cdot |R \cap_\pi S|$$

where $\sigma_{V,R}$ is the *view selectivity* of R with regard to V and indicates the percentage of tuples in relation R that contribute to the view V. This value is assumed to remain constant even when a subset of relation R is taken into account, i.e., $R_1 \subset R_2 \Rightarrow \sigma_{V,R_1} = \sigma_{V,R_2}$.

The size of the overlaps of the base relations can be estimated if $\mathcal{PC}$-constraints between the original relation and the replacing relation are available. If this is not the case, we will assume the view overlap to be equal to the smaller of the two views and compute only the size of the new view during the graph traversal, i.e., we set:

$$|V \cap V_i| := \begin{cases} |V_i^{\pi(V)}| & : \quad |V_i^{\pi(V)}| < |V^{\pi(V_i)}| \\ |V^{\pi(V_i)}| & : \quad |V^{\pi(V_i)}| < |V_i^{\pi(V)}| \end{cases} \tag{7}$$

This method makes the assumption that a good view rewriting is one whose size is similar to the size of the original view.

3.3.4 Cost Factor Based on Number of Messages Exchanged (CF_M)

The number of messages exchanged between the information space and the view site for a single base data update, denoted as CF_M, is in the range $[0, 2m]$ (with m denoting the number of information sources involved in the view). To be more specific:

$$CF_M = \begin{cases} 0 & \text{if } m = 1 \text{ and } n_1 = 0 \\ 2 & \text{if } m = 1 \text{ and } n_1 > 0 \\ 2 \cdot (m-1) & \text{if } m > 1 \text{ and } n_1 = 0 \\ 2 \cdot m & \text{otherwise} \end{cases}$$

n_1 is the number of relations in the update-generating IS *besides* the relation where the update occured. The best case $CF_M = 0$ occurs when there is only one relation in the update-generating IS referred to in V $(n_1 = 0)$. Then, no query needs to be sent at all.

3.3.5 Cost Factor Based on Bytes of Data Transferred (CF_T)

Considering an information space consisting of n relations $R_1, \ldots, R_n$ in m information sources $IS_1, \ldots, IS_m$, we can derive a general computation for the number of bytes transferred (assuming the distributed incremental view maintenance algorithm described in [LKNR98]). This computation assumes that one inserted/deleted tuple is sent from IS_1 to the view site, which is the initial delta relation (the first line in Equation 8). Then this delta relation is sent down to IS_1 to join with other relations in IS_1 referred to in the view query (the first term in the second line), and the resulting new delta relation (the second term in the second line) is sent back to the view site. The same process iterates through all ISs referred to in the view to build up the delta relation that contains the tuples affected by the data update. In summary, the number of bytes transferred can thus be approximated by:

$$
\begin{aligned}
CF_T \quad = \quad & \underbrace{s_{R_{1,0}}}_{update\ notification} \\
+ \quad & \underbrace{s_{R_{1,0}}}_{\Delta R_{in,IS_1}} + \underbrace{\sigma_{IS_1} \cdot J_{IS_1} \cdot s_{\Delta R_{out,IS_1}}}_{\Delta R_{out,IS_1}} \\
+ \quad & \underbrace{\sigma_{IS_1} \cdot J_{IS_1} \cdot s_{\Delta R_{out,IS_1}}}_{\Delta R_{in,IS_2}} + \underbrace{\sigma_{IS_1}\sigma_{IS_2} \cdot J_{IS_1}J_{IS_2} \cdot s_{\Delta R_{out,IS_2}}}_{\Delta R_{out,IS_2}} + \cdots \\
+ \quad & \underbrace{(\sigma_{IS_1} \cdot \ldots \cdot \sigma_{IS_{m-1}})(J_{IS_1} \cdot \ldots \cdot J_{IS_{m-1}}) s_{\Delta R_{out,IS_{m-1}}}}_{\Delta R_{in,IS_m}} + \underbrace{(\sigma_{IS_1} \cdot \ldots \cdot \sigma_{IS_m})(J_{IS_1} \cdot \ldots \cdot J_{IS_m}) s_{\Delta R_{out,IS_m}}}_{\Delta R_{out,IS_m}}
\end{aligned}
\tag{8}
$$

where s_R is the size (sum of the length of attributes in bytes) of the relation R or intermediate query result R, σ_{IS_i} is the average selectivity for the selection conditions used in the single-site query to source IS_i [3], and J_{IS_i} is the estimated size of the resulting join relation $R_{i,1} \bowtie R_{i,2} \bowtie \cdots \bowtie R_{i,n_i}$ returned by the source IS_i. That is, $J_{IS_i} \approx js^{n_i} \cdot |R_{i,1}| \cdot |R_{i,2}| \cdot \ldots \cdot |R_{i,n_i}|$, with js^{n_i} being the average join selectivity for this IS, raised to the power of the number of relations in this IS.

3.3.6 Cost Factor Based on I/O ($CF_{I/O}$)

We use the total number of estimated input/output operations (block accesses) performed by local ISs in order to process incremental view maintenance for each legal rewriting as a criterion to rank the legal rewritings. Let $CF_{I/O}(IS_i)$ be the number of estimated I/Os at the information source IS_i. $CF_{I/O}(IS_i)$ is the sum of the I/Os of the relations that reside at source IS_i, i.e., incorporating the I/O-costs of all relations at IS_i. Then the total number of I/Os, denoted as $CF_{I/O}$, is the sum of the I/Os at all m sources, i.e.,

$$CF_{I/O} = \sum_{i=1}^{m} CF_{I/O}(IS_i). \tag{9}$$

One way to estimate the number of blocks accessed in order to retrieve a tuple from a database is a formula given by Yao [Yao77]. With

$$B' = \left\lfloor \frac{|R_i|}{bfr_{R_i}} \right\rfloor,$$

we obtain

$$CF_{IO}(i) = 1 + B' - \left[1 \Big/ \binom{|R_i|}{\beta} \right] \sum_{k=1}^{B'} \binom{k \cdot bfr_{R_i}}{\beta} \tag{10}$$

with bfr_{R_i} the number of bytes per physical block in the information source for relation R_i (blocking factor). This computation yields the number of blocks accessed in a database storing a relation of size $|R_i|$ when searching for β tuples. β will be an input parameter computed as $\beta = |R_1| \cdot js_{R_1,R_2} \cdot |R_2| \cdot \ldots \cdot |R_n - 1| \cdot js_{R_{n-1},R_n} \cdot |R_n|$ for a join $R_1 \bowtie \ldots \bowtie R_n$.

3.3.7 Combining the Five Factors

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*:

[3]Since we assume there is one local condition for each relation residing in IS_i, $\sigma_{IS_i} = \sigma_{R_{i,1}} \cdot \sigma_{R_{i,2}} \cdot \ldots \cdot \sigma_{R_{i,n_i}}$ for $1 \le i \le m$.

$$QC(V_i) = 1 - \Big[\varrho_{quality} \cdot (\varrho_{attr} \cdot \mathcal{DD}_{attr} + \varrho_{ext} \cdot \mathcal{DD}_{ext}) + \varrho_{cost} \cdot (\mathrm{cost}_M \cdot CF_M + \mathrm{cost}_T \cdot CF_T + \mathrm{cost}_{I/O} \cdot CF_{I/O})\Big] \tag{11}$$

with V_i being 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 $\mathrm{cost}_M, \mathrm{cost}_T, \mathrm{cost}_{I/O} > 0$. The unit costs can be empirically computed for a given data warehouse by a method proposed in [LKNR98], 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 5, 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 connecting the original remaining view definition with 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 all those n_R 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 remaining view query after deletion 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., $n'_R < n_R$, first step in Figure 5). 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 5, 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. This gives the algorithm exponential complexity. Finally, the rewritings found have to be sorted by their QC-Value (third step in Figure 5) in order to present them to the view user.

As can be seen from the above considerations, 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 of the exponentially many different rewritings (i.e., performing view synchronization in two separate phases).

4 Mapping View Synchronization to a Weighted Graph Problem

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 relations remaining in the query after deleting R and the replacement relation R_k. 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}:

$$QC_{total}(V_i) = QC(V_i) = f(V, V_i, MKB). \tag{12}$$

with the old view query V, the new view query V_i, and the MKB [4].

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 the relations in an original query to all other relations R_k 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_k. So we have:

$$QC_{incr}(R_k) = f(V, R, R_k, MKB). \tag{13}$$

For the purpose of introducing the mapping of the algorithm to a graph problem, we will also use the notation $QC_{incr_P}(R_k)$ for the incremental QC-Value of a view rewriting using R_k which does not necessarily use the *optimal* path but rather path P.

Figure 6 contrasts our new approach, referred to as the *Graph-Based Shortest-Path (GRASP)* view synchronization algorithm, with the original process in Figure 5, 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 7 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

[4]The notation $y = f(x)$ denotes that there exists a function from domain x to domain y.

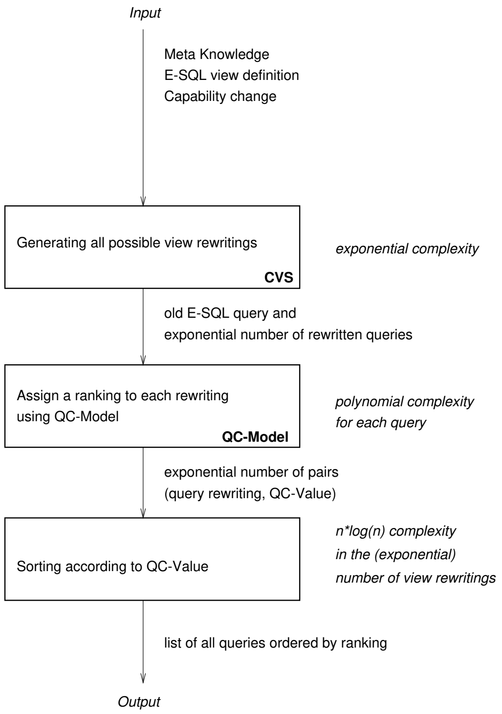

Figure 5: View Synchronization as a Multi-Stage Computation Process

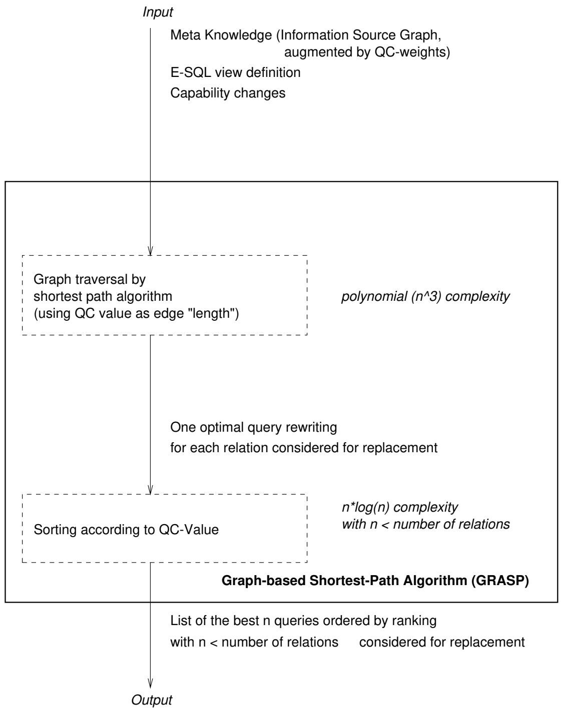

Figure 6: View Synchronization as Integrated Process

mapping, a path through the IS-Graph (from the view query to an end vertex R_k) represents exactly one possible query rewriting V_i that uses the relation R_k for a replacement for missing view elements from R. Hence, all paths between the vertices representing the original query and any other (reachable) vertex have to be considered as possible replacements.

Figure 7 depicts the IS-Graph for the example information space from Section 3. Assume the query in Equation 2. 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* ⋈ *Participant*) to *Accident-Ins*, namely:

$$\begin{array}{ll} & \textit{FlightRes} \bowtie_{JC_6} \textit{PreferredCust} \\ \text{and} & \textit{FlightRes} \bowtie_{JC_5} \textit{Accident-Ins} \bowtie_{JC_2} \textit{PreferredCust} \end{array}$$

Those two paths are depicted in Figure 7 by thick lines. 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 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

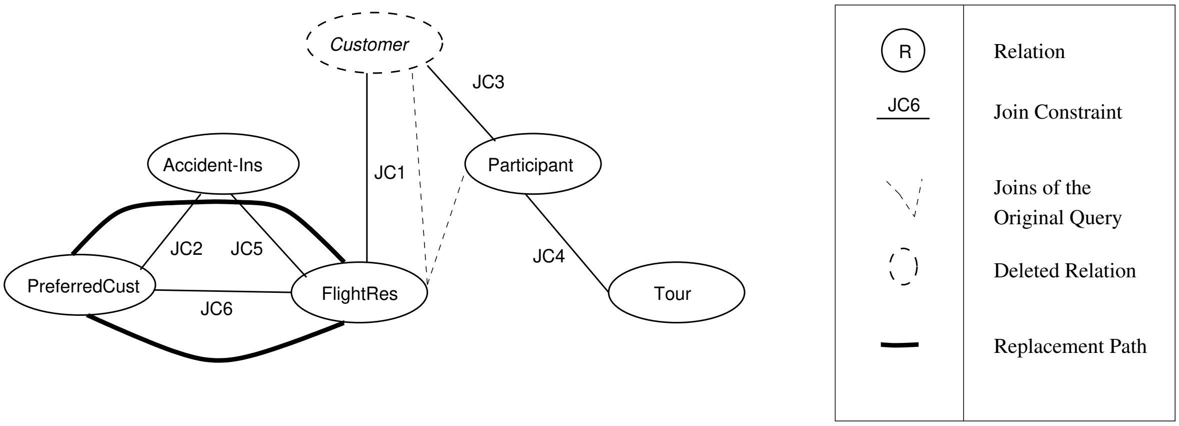

Figure 7: The Information Space Graph for Example 1.

QC_{total}).

4.2 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 (Section 3.3). 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:

$$qc(P_i) = (qc_{DD_{ext}}(P_i), qc_{CF_M}(P_i), qc_{CF_T}(P_i), qc_{CF_{I/O}}(P_i)). \tag{14}$$

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

$$\begin{aligned} qc(P_0) &= qc_{E_{l_0}} \\ qc(P_k) &= (qc_{DD_{ext}}(P_k), qc_{CF_M}(P_k), qc_{CF_T}(P_k), qc_{CF_{I/O}}(P_k)) \\ &= (qc_{DD_{ext}}(P_{k-1}) \cdot \mathcal{DD}_{ext}(l_k),\ qc_{CF_M}(P_{k-1}) + CF_M(l_k), \\ &\quad qc_{CF_T}(P_{k-1}) + CF_T(l_k),\ qc_{CF_{I/O}}(P_{k-1}) + CF_{I/O}(l_k), \end{aligned} \tag{15}$$

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 Section 4.3 (see also our technical report on the QC-Model [LKNR98]).

At any vertex N_k in the path, we compute the intermediate QC-Value $QC_{incr_{P_k}}(R_k)$ for sub-path P_k traversed for the replacement relation R_k by putting together the 5 (4) factors computed for the sub-path as in Equations 15

[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.

(see also Equation 11):

$$\begin{aligned} QC_{incr_{P_k}}(R_k) &= 1 - \Big[\varrho_{quality} \cdot (\varrho_{attr} \cdot \mathcal{DD}_{attr} + \varrho_{ext} \cdot qc_{DD_{ext}}(P_k)) + \\ &\quad \varrho_{cost} \cdot (\text{cost}_M \cdot qc_{CF_M}(P_k) + \text{cost}_T \cdot qc_{CF_T}(P_k) + \text{cost}_{I/O} \cdot qc_{CF_{I/O}}(P_k))\Big] \,. \end{aligned} \tag{16}$$

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.

If the replacing relation R_k is reached (i.e., the last edge E_k of path P_k ends in R_k), we have computed the incremental QC-Value for replacing relation R_k. We will denote the current (complete) path to R_k by P. The incremental QC-Value for this particular path P can now be defined as:

$$QC_{incr_P}(R_k) = QC_{incr_{P_k}}(R_k)$$

with P_k being the complete path to the replacement relation R_k.

4.3 Semantics of Incremental Computation

We now need to show that the complete computation of the QC-Value for a 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

$$QC_{total}(V_i) = QC_{incr_P}(R_k) \tag{17}$$

for a view V_i that uses R_k as a replacement and joins it with the existing query through path P. Previously, the QC-Value was computed by the quality and cost formulas given in [LKNR98] (Section 3.3). Since we can apply Equation 16 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 $qc_{op}(k)$ as

$$qc_{op}(k) = f(l_0) \odot_{op} f(l_1) \odot_{op} \ldots \odot_{op} f(l_k) \tag{18}$$

with $op \in \{DD_{ext}, CF_M, CF_T, CF_{I/O}\}$ and operations $\odot_{op}$ on these values (cf. Equation 15). Since this computes the QC-Value from left to right for each factor f, $\odot_{op}$ has to be shown to be left-associative, i.e.,

$$f(l_0) \odot_{op} f(l_1) \odot_{op} f(l_2) \odot_{op} \ldots \odot_{op} f(l_k) = (\ldots((f(l_0) \odot_{op} f(l_1)) \odot_{op} f(l_2)) \ldots \odot_{op} f(l_k)) \tag{19}$$

for our path $P_k = (E_{l_0}, E_{l_1}, \ldots, E_{l_k})$. If we can compute all factors incrementally and all four operations $\odot_{op}$ are left-associative, then the incremental computation of $QC_{incr}(i)$ will deliver the same result as the total computation, i.e., $QC_{incr_P}(R_k) = QC_{total}(V_i)$ for a path P that leads to R_k and a view rewriting V_i that uses R_k as replacement relation for R and uses P. In order to show these required characteristics for each factor, we now describe and analyze 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.

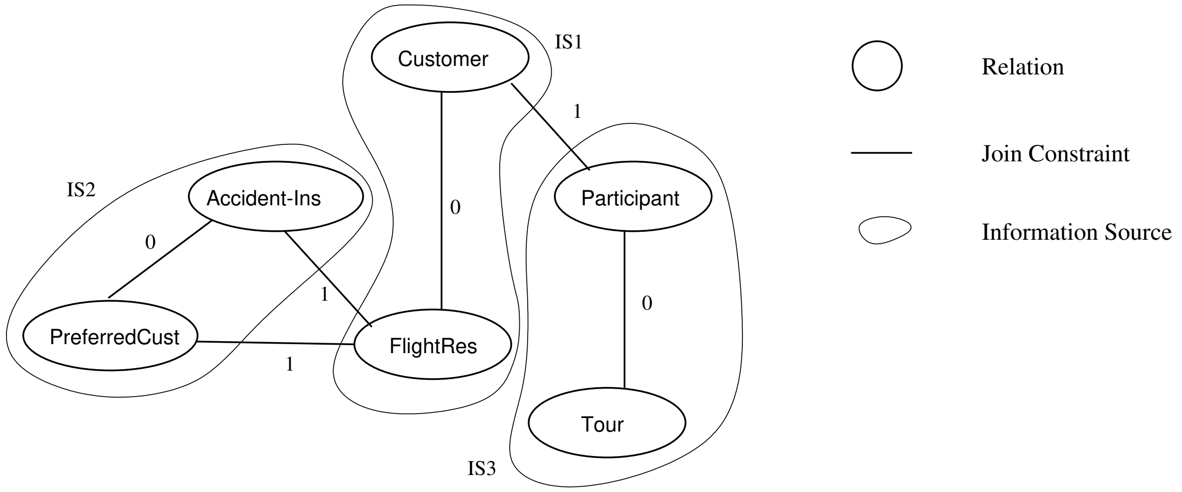

Figure 8: Assigning Weights for CF_M.

View extent—$\mathcal{DD}_{ext}$. As introduced in section 3.3.2, we compute the size of view extents $|V_i|$ and overlaps $|V \cap V_i|$ 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 using Equation 6. 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 $\mathcal{DD}_{ext}(k)$ component of qc_{E_k}. The size of the original view $|V|$ is known beforehand and does not have to be computed during the incremental QC-computation. 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.

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. This indicates the requirement that a shortest-path algorithm for this problem must be *stable under "negative" edge lengths*, as we further discuss for our solution in Section 5.

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. In some cases, additional $\mathcal{PC}$-constraints can be used to determine the satisfaction of the view extent constraint. Whenever such constraints are not available, we will assign a value of ∞ to $\mathcal{DD}_{ext}$ to terminate the current path.

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 8). In order to calculate the number of messages for a given query, we can then add up the edge weights along the path. Every time a new information source is included in the query, the length of the path will increase by 1, every time the next relation is in the same information source as the previous one, the path length will remain the same. Adding 1 to the complete path length will exactly give the number of messages in the query as defined in Section 3.3.4.

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. Equation 8):

$$CF_T(k) = 2 \cdot (\sigma_{IS_{l_1}} \cdot \ldots \cdot \sigma_{IS_{l_k}})(J_{IS_{l_1}} \cdot \ldots \cdot J_{IS_{l_k}})s_{\Delta R_{out,IS_{l_k}}} \tag{20}$$

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 $s_{\Delta R_{out,IS_{l_k}}}$ the size (sum of the lengths of attributes in bytes) of a sub-query to an IS [LKNR98]. The operation is addition, with the summands (from Equation 8) dependent on the previous 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 than computing sub-paths at random and adding the results), we can perform this computation incrementally as $qc_{CF_T}(k) = qc_{CF_T}(k-1) + CF_T(l_k)$. In Section 5, we confirm that our solution meets this requirement (*knowledge of path "history"*). 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 (cf. Equation 10). The I/O-cost depends on the number of tuples that have to be retrieved from the current relation for an incremental update. Given this parameter, we can compute the number of block accesses in the current database and add this number to the previously computed I/O-cost. 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.

4.4 Equivalence of Incremental and Total Computation

Theorem 1 *For each view rewriting V_i that uses R_k as the replacement relation and joins the original query with R_k using path P in the IS-Graph the following equation holds:*

$$QC_{total}(V_i) = QC_{incr_P}(R_k) \tag{21}$$

Proof: 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. □

Lemma 1 *Each path P to a relation R_k in the IS-Graph is a potential solution for the problem of generating a view rewriting using R_k.*

Proof: Each path through the IS-Graph corresponds to a sequence of WHERE-conditions in a SQL-query. That sequence can be used to generate the view V_i for the total computation.

5 Finding the Best View Rewriting: The Shortest Path Approach

In order to find the optimal view rewriting for a given replacement relation, we need to optimize, i.e., *maximize* the QC-Value. Thus, we need to find a path P through the IS-Graph such that

$$QC_{incr_P}(R_k) = max(QC_{incr_{P*}}(R_k)) \tag{22}$$

for P^* the set of all possible paths leading to relation R_k.

5.1 Using the Bellman-Ford Algorithm (Assumption of No Negative Cycles)

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. 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] on the input parameters G (the graph), w (the set of weights on the edges) and s (the source vertex). This single source would be our original view query V which can be abstracted as one node in the IS-Graph.

We give a pseudo-code representation in Figure 9. $w(u,v) \in w$ denotes the weight of edge (u,v), $d[v]$ denotes the distance value currently kept with vertex v. The algorithm keeps distances (sums of edge weights) from the source node s for each node and updates those value as it iterates through the vertices and edges. Those distances are initialized with ∞ in Step (1). The operation $\textsc{Relax}(u,v,w)$ updates the distance value of v to $d[v] \leftarrow d[u] + w(u,v)$ iff $d[v] > d[u] + w(u,v)$. Step (2) iterates through each edge of the graph $|V[G]|$ times and thus finds the shortest paths to all nodes from node s iff there are no negative length cycles in the graph. Step (3) checks if the edge weights and distances are consistent with each other. If they are not, the graph must contain a negative length cycle. The cost function of the original algorithm is $cost(P_k) = \sum_{i=0}^{k-1} w(E_i, E_{i+1})$ for a path $P_k = (E_0, E_1, \ldots, E_k)$. $\mathcal{BF}$ is stable under negative length edge weights, as long as the graph does not contain a negative-length (contracting) cycle.

```
BELLMAN-FORD (G, w, s)
(1) INITIALIZE-SINGLE-SOURCE(G, s)
(2) for i ← 1 to |V[G]| − 1 do
(2a)    for each edge (u, v) ∈ E[G]
            do RELAX (u, v, w)
(3) for each edge (u, v) ∈ E[G]
        do if d[v] > d[u] + w(u, v)
             then return FALSE
(4) return TRUE
```

Figure 9: The Bellman-Ford Algorithm [CLR90].

The cost function we use is

$$cost(P_k) = 1 - QC_{incr_{P_k}}(R_k) \tag{23}$$

for a sub-path P_k and the QC-Value computed according to Equation 16, since $\mathcal{BF}$ *minimizes* path lengths. The construction of edge labels has been described in Section 4.2. With this cost function, the edge relaxation step works as follows: In order to compute $d[v]$, or $cost(P_k)$ for a path P_k ending in v, we use $d[u] = cost(P_{k-1})$ for the path ending in u and apply the weights on edge (u,v) according to Equations 15 and 16. We will denote this operation as $d[v] \leftarrow d[u] \odot w(u,v)$ for the purpose of simplicity in demonstrating the algorithm.

$\mathcal{BF}$ 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_k, which, in our mapping, represents the "chain" of joins to a relation R_k used to rewrite the given query V after a capability change.

Whenever an edge relaxation step is applied, a path from the starting node to the current edge is known. Along this path, the required parameters such as the current size of the join (needed for the computation of CF_T

and $CF_{I/O}$, cf. Section 4.3) can be computed.

5.2 Complexity of $\mathcal{BF}$

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

5.3 Extending $\mathcal{BF}$ for "Negative Length" Cycles

Since the computation of our QC-Value can potentially lead to "contracting" edge lengths (meaning that the value of the cost function decreases with an edge inclusion, i.e., $cost(P_2) < cost(P_1)$ for $P_1 = (E_0, E_1, \cdots, E_n)$ and $P_2 = (E_0, E_1, \cdots, E_n, E_n + 1)$) it is necessary to look at the possibility of contracting cycles, usually referred to as *negative-length* cycles, in our algorithm. A graph containing an infinite negative-length cycle has no shortest path. In order to show that the problem of infinite negative length cycles can be solved in our context, we take a closer look at the computation of this value.

Out of the five factors contributing to the QC-Value (cf. Section 4.3), one factor ($\mathcal{DD}_{attr}$) is not affected by the join path, and three factors will increase with each new edge being incorporated into a path:

- CF_M, the number of messages between data warehouse and information sources will increase (by 1) if the next relation to be joined resides in an information source different from the previous relation. Otherwise, this factor will remain constant.
- CF_T, the number of bytes transferred between data warehouse and sources for an incremental update will always increase or remain constant since for each new relation included in the join a new term will be added to the computation (Equation 8), and this term will be non-negative.
- $CF_{I/O}$, the number of I/O-Operations in the local sources will also increase with each new relation, since that new relation has to be retrieved from some information source (and causes additional I/O operations).

Thus, the cost factors ($CF_M, CF_T, CF_{I/O}$) will always be increasing or at least remain constant with each new relation added to the query. Only the remaining quality factor $\mathcal{DD}_{ext}$ could possibly be contracting, meaning that the (incrementally computed) Degree of Divergence of the View Extent could be reduced with the inclusion of another relation into the join path (since the operation executed here is multiplication, and the factors could be potentially be smaller than 1, although not negative, cf. Section 3.3.2).

Theorem 2 *The IS-Graph used for the computation of $QC_{incr}(R_k)$ does not contain infinitely contracting cycles.*

Proof: We first note that the value $\mathcal{DD}_{ext}$ is computed as a ratio between integer numbers, multiplied with a constant (cf. Equation 6). Since we are in an integer domain and relations have finite size, this value cannot asymptotically approach a limit (e.g., 0). For example, in the case of a decreasing $\mathcal{DD}_{ext}$, this value will be 0 after a finite number of iterations.

We now consider a join path $P = (E_0, E_1, \ldots, E_n)$ that contains a sub-path $P_{sub} = (E_k, E_{k+1}, \ldots, E_{k+m})$ such that $0 \leq k < k+m \leq n$ and $P_1 = (E_0, E_1, \ldots, E_{k-1}) + P_{sub}$, $P_2 = P_1 + P_{sub}$ with + an operation to concatenate

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

paths. A negative-length or *contracting* cycle then occurs if $cost(P_2) < cost(P_1)$. In our context, P_{sub} is a join of relations, which is itself a relation. We can therefore define: $R_C := R_k \bowtie \ldots \bowtie R_{k+m}$ Expressed in terms of our relational environment, a contracting cycle then occurs if the QC-Value for a join $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_C$ is smaller (worse) than the QC-Value of the same join after the second inclusion of the relations in the cycle, i.e., $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_C \bowtie R_C$.

Depending on the join selectivity js_{R_C,R_C}, the incrementally computed new view containing R_C will either shrink, remain constant, or grow with each inclusion of R_C. Note here that, although the join selectivity is a pre-computed or estimated constant, the size of actual joins may not be exactly determinable using an estimation through join selectivities. Especially the case of constantly joining a relation with itself (in a loop) can change the join selectivity with each loop. Nonetheless, our model assumes join selectivities to be constant in order to give estimations for the QC-Value. Depending on the join selectivity, we consider these three cases:

1. Decreasing intermediate size—$js_{R_C,R_C} < 1/R_C$
   With decreasing size of the intermediate relation, the constant join with R_C will eventually lead to $|V_i| = 0$ (see above), i.e., to a view rewriting with an empty extent. This clearly has $\mathcal{DD}_{ext} = 1$, so further inclusions of R_C will not improve this quality factor.

2. Constant size—$js_{R_C,R_C} = 1/R_C$
   It is possible that the intermediate view extent does not change with a new inclusion of R_C into the join. In this case, the factor $\mathcal{DD}_{ext}$ remains constant, and the increasing three factors CF_M, CF_T, and $CF_{I/O}$ will lead to an increasing value of $cost(P_k)$.

3. Increasing size—$js_{R_C,R_C} > 1/R_C$
   If the estimated intermediate view extent grows with every new join with R_C, both $|V_i|$ and $|V \cap V_i|$ can either grow or shrink, so the QC-Value could be affected in both directions (increasing or decreasing). Note that $|V \cap V_i|$ has an upper bound, namely $|V \cap V_i| < |V|$. For a growing V_i, the intermediate value for $|V \cap V_i|$ will eventually approach this maximum, whereas the size $|V_i|$ will keep growing (cf. also Equation 7). A asymptotic approach of $|V \cap V_i|$ towards $|V_i|$ is not possible, since both number are integers, so that the conditions for asymptotic behavior are not given. This situation, by our definition stated in Equation 6, eventually leads to an increasing value for $\mathcal{DD}_{ext}$ and will therefore not infinitely improve the QC-Value.

□

We can now say that infinitely contracting cycles are not possible in our model even though negative length (contracting) cycles are possible. An algorithm finding the shortest path in the IS-Graph must be capable of following such cycles for a finite number of loops and include paths with such cycles in the set of solutions.

We will now describe the extensions necessary for $\mathcal{BF}$ to accommodate negative length cycles. As mentioned above, the Bellman-Ford algorithm $\mathcal{BF}$ as given in the literature [CLR90] runs on a graph with v vertices and e edges, iterates through all e edges v times and applies an edge-relaxation step in each loop (cf. Step 2 in Figure 9). With each vertex v, an estimated *distance* $d[v]$ is kept which is an upper bound for the actual cost of a path from the start node to v. After $v \cdot e$ edge relaxations, all distances are correct except in the case of negative length cycles. If such a cycle occurs, at least one of the edges (u, v) will not have the property $d[v] \leq d[u] + w(u, v)$ with $w(u, v)$ the weight of edge (u, v) (Step 3). If this is detected, the original $\mathcal{BF}$ terminates.

In this case, some distance labels on nodes will include distances obtained by following negative length cycles which are not necessarily optimal. In order to adapt the algorithm to our needs, we will in this case apply another e edge relaxation steps (one for each edge), i.e., rerun the inner loop (Step 2a) of $\mathcal{BF}$ once. The effect of this procedure is to follow another loop through the negative-length cycle. After that step, we will check the graph again for optimality (Step 3) and if it is not optimal, repeat the procedure. A pseudo-code representation of this new algorithm $\mathcal{BF}$' is given in Figure 10.

```
ADAPTED-BELLMAN-FORD (G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i ← 1 to |V[G]| − 1 do
  for each edge (u, v) ∈ E[G]
     do RELAX (u, v, w)
  do
    neg ← FALSE
    for each edge (u, v) ∈ E[G]
      do if d[v] > d[u] ⊙ w(u, v)
        then neg ← TRUE
    if(neg)
      then for each edge (u, v) ∈ E[G]
          do RELAX (u, v, w)
  while (neg)
  return
```

Figure 10: The adapted Bellman-Ford Algorithm $\mathcal{BF}$'.

Lemma 2 *$\mathcal{BF}$' will always terminate on the IS-Graph after a finite number of additional edge relaxation steps.*

Proof: The proof follows from Theorem 2 since infinitely contracting cycles are not possible. □

Although we can find optimal paths containing negative length cycles in this way, it is not likely that a real-world application has such cycles. The semantics of a negative-length cycle in our domain would be a view that could *better* be rewritten by a join of the form $R \bowtie S \bowtie S \bowtie T$ than by a join $R \bowtie S \bowtie T$. Clearly, this is not very likely for meaningful data. Nevertheless, it is possible to create examples that show that such a cycle is possible and will in fact lead to an optimal view rewriting that contains a finite number of cycles through a subset of nodes. The worst-case complexity of our adapted $\mathcal{BF}$ algorithm will increase slightly, since $2 \cdot |E|$ steps have to be executed for each additional loop through the IS-Graph. The complexity of $\mathcal{BF}$' is therefore:

$$c(\mathcal{BF}') = c(\mathcal{BF}) + l \cdot (2|E|)$$

with l some (small) finite integer which is determined by join selectivities and relation sizes in the IS-Graph.

6 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 and '98 conferences [LNR97a] as well as during EDBT'98, and is available on our *EVE* project web page under `http://davis.wpi.edu/dsrg/EVE`. 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.

[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.

[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.

[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.

[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.

[Yao77] S. B. Yao. An Attribute Based Model for Database Access Cost Analysis. *ACM Transactions on Database Systems (TODS)*, 2(1):45–67, March 1977.

[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.