D W Q

Foundations of **D**ata **W**arehouse **Q**uality

National Technical University of Athens (NTUA)
Informatik V & Lehr- und Forschungsgebiet Theoretische Informatik (RWTH)
Institute National de Recherche en Informatique et en Automatique (INRIA)
Deutsche Forschungszentrum für künstliche Intelligenz (DFKI)
University of Rome «La Sapienza» (Uniroma)
Istituto per la Ricerca Scientifica e Tecnologica (IRST)

M. Staudt, M. Jarke

Incremental Maintenance of externally materialized views

In Proc. of the 22nd VLDB Conference, Mumbai, India, 1996.

DWQ : ESPRIT Long Term Research Project, No 22469
Contact Person : Prof. Yannis Vassiliou, National Technical University of Athens,
15773 Zographou, GREECE Tel +30-1-772-2526 FAX: +30-1-772-2527, e-mail: yv@cs.ntua.gr

Incremental Maintenance of Externally Materialized Views

Martin Staudt Matthias Jarke

RWTH Aachen, Informatik V, Ahornstr. 55, D-52056 Aachen, Germany
{staudt,jarke}@informatik.rwth-aachen.de

Abstract

With the advent of the Internet, access to database servers from autonomous clients will become more and more popular. In this paper, we propose a monitoring service that could be offered by such database servers, and present algorithms for its implementation. In contrast to published view maintenance algorithms, we do not assume that the server has access to the original materialization when computing differential view changes to be notified. We also do not assume any database capabilities on the client side and therefore compute precisely the required differentials rather than just an approximation, as is done by cache coherence techniques in homogeneous client-server databases. The method has been implemented in ConceptBase, a meta data management system supporting an Internet-based client-server architecture, and tried out in some cooperative design applications.

1 Introduction

Facilitated by the Internet, wide-area access to database servers by autonomous clients (which may or may not have local databases) is becoming more and more popular (figure 1). To reduce application programming effort, such clients demand more sophisticated services than the simple read and write transactions offered by current standards such as RDA.

Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996

An obvious candidate is a *monitoring service*. The client does not only request the initial answer to a certain query but also notifications about changes in this answer over an extended period of time, with a specified quality of service in terms of precision and actuality.

Database monitoring is not a new problem. Even in central databases, it is needed for notifying application programs or end-users about integrity violations [1], or to assist users in supervising complex processes (stock trading, power plants, ...). A more recent example is group awareness in cooperative engineering: designers working on a certain aspect of a product should be made aware of concurrent changes in requirements or by other designers. Yet another step towards the open electronic Internet market is change propagation in data warehousing [23, 10].

Traditional database systems leave the responsibility of keeping informed about updates largely to the client. Since the client cannot know what changes have happened, it must repeat queries in a polling mode. Even worse, although the server must re-compute the whole query each time, the client usually must *in addition* compute the differentials to highlight them to the end-user, thus duplicating a lot of DBMS functionalities.

Active database technology offers a partial solution by triggers that can produce effects outside the database. Distributed programming languages such as Java moreover allow the server to add certain functionalities to client programs, e.g. ensuring that they can accept change notifications and relate them to the original query. Recently, our group has also developed a 'coherency index' and corresponding extensions to distributed transaction management by which timeliness of service can be tailored to customer wishes [3].

However, the question how to compute the necessary changes is not answered by these base technologies if the view definition is reasonably complex or if views are defined on top of each other (possibly with recursion). The corresponding trigger programs become so complex that it is not conceivable they could

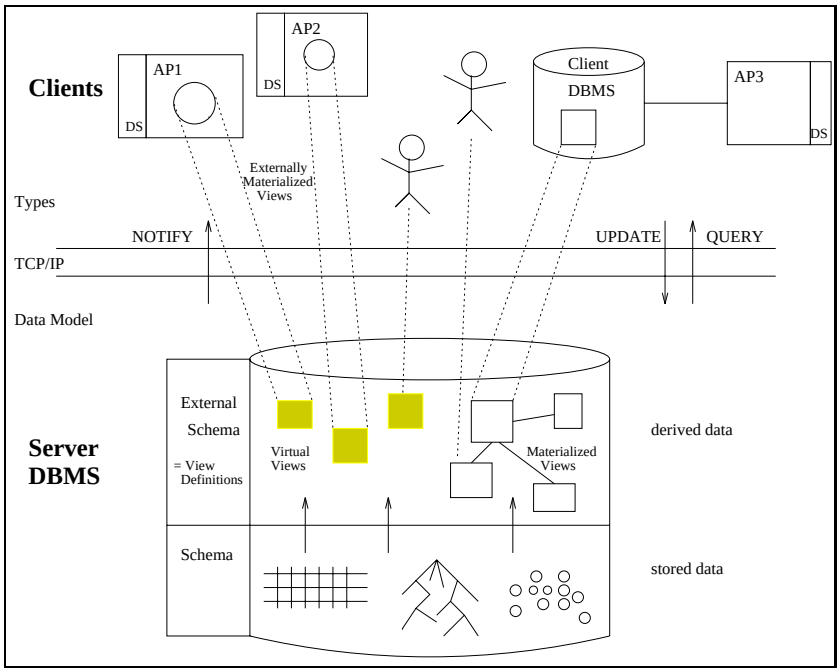

Figure 1: Client-server architecture for database applications

be written by client developers.

The published view maintenance algorithms [5] do not fully solve the monitoring problem since they assume that views are materialized logically *within* the database, either within the server or by endowing the client with DBMS capabilities plus a local database cache (AP3 in figure 1). Such a cache gives the system a lot of flexibility when to propagate changes, and with what precision [15, 8], but is easily possible only within a homogeneous environment. Moreover, it just shifts the problem from the client-server communication to the database-interface communication within the client.

In this paper, we present two related algorithms for a monitoring service. The first one, reported in section 3, assumes that the server maintains a local view materialization in addition to the external one. It achieves the precision of state-of-the-art incremental view maintenance techniques by a purely declarative rule program, rather than resorting to procedural components [6] or meta predicates [9].

This declarative approach is important because it is a prerequisite for the second algorithm in which the server only remembers the view definition to be monitored (section 4); we have not found a solution for this problem in the literature. The algorithm neither maintains a server-side materialization nor has it access to the external client-side materialization. It also eliminates the need for client-side computing of view differentials as these are precisely computed and sent by the server. The basic idea is to further rewrite the maintenance rules into trigger rules that selectively re-derive the pieces of an externally materialized view needed for computing the client-view differentials, using a standard DBMS query evaluator. As a consequence, the approach advocated here could also be used by a mediator that operates on top of a collection of distributed source databases [22].

The embedding of the algorithm in a full monitoring service is illustrated in section 5, by briefly describing its implementation and initial application experiences in ConceptBase, a meta database management system supporting a deductive object data model and operating in an Internet-based client-server architecture. Section 6 discusses related work and open issues.

2 Notations and Example

2.1 Notation and Prerequisites

The algorithms in this paper are presented in a Datalog formalism [21] although they apply equally to other extended relational database languages[1]. A deductive database consists of a set of base relations EDB and a set of rules defining intensional relations IDB. These rules are of the form

$$p(\vec{X}) :- l_1(\vec{X_1}), \ldots, l_n(\vec{X_n})$$

where p is a predicate symbol corresponding to a relation in IDB, the l_i are literals of the form r_i or $\neg\, r_i$ for relations r_i and $\vec{X}$, $\vec{X_i}$ are non disjunct sequences of constants and variables. For readability, we omit variables in the following. We assume that all rules are safe: variables occuring in the rule head occur in

[1] For example, our implementation uses the deductive object language Telos [12].

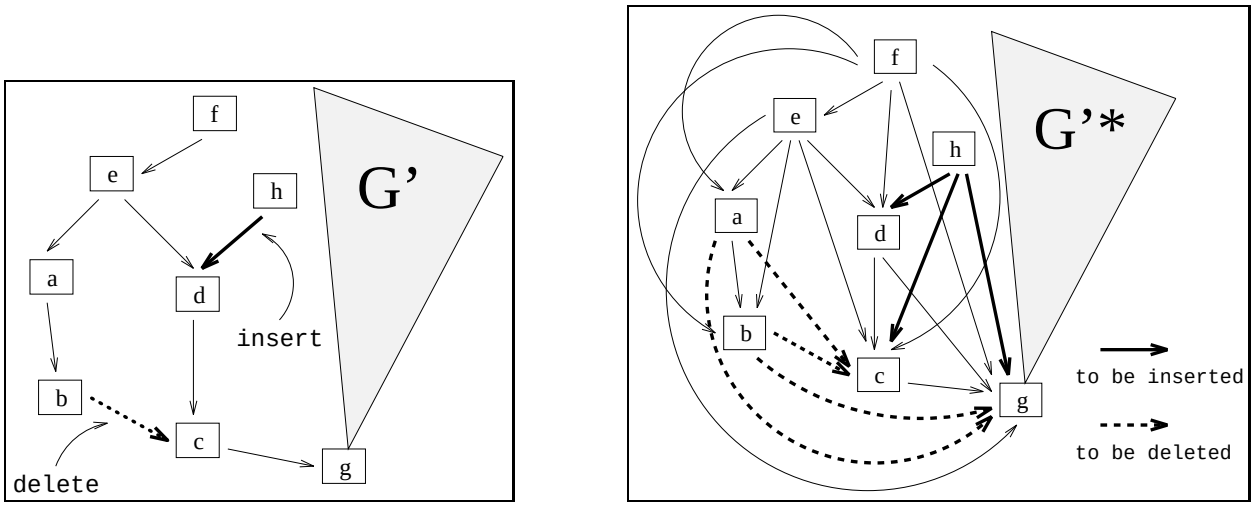

Figure 2: (a) Changes in the database (b) Changes in the view

the rule body, too, and variables appearing in negated predicates are bound through some additional positive literals in the rule body.

Intensional relations may represent views of applications; in this case, we call the derivation rules *view definitions*. Query optimization algorithms such as the magic set transformation [21] convert these view definitions into sets of *query rules* which are more suited for an efficient bottom-up query evaluation procedure. An intensional relation is *materialized* if its derived extension is stored in the database.

Update transactions on base relations have to be propagated through the rules towards the view relations in order to provide notifications to the affected client applications. The base data updates for a relation r are assumed to be available as relations r^{ins} and r^{del}. They are applied to r *after* the process of computing their consequences on derived views. In general, the set of views that has to be checked for changes is based only on a restricted subset of the IDB rules that can be determined easily, e.g. by a rule/goal graph [21].

2.2 Incremental View Maintenance: An Example

The following simple scenario illustrates the problem of maintaining views on the database managed by external client tools. An extensional database relation *edge* represents a directed graph, and a view maintained by a graph display tool contains the transitive closure of this graph. The view definition defines an intensional relation *closure* by two Datalog rules:

$R_1 : closure(x, y) :- edge(x, y).$

$R_2 : closure(x, y) :- edge(x, z), closure(z, y).$

A sample extension of *edge* is shown in figure 2 (a). It has a subgraph G' where g is the only node that is connected with nodes occuring in G'. G' is interpreted as the *rest* of relation *edge* which is not touched by update operations. G' may be a very large graph and the computation of its transitive closure G'^* is expensive.

When starting up, the graph display tool asks the database to compute the extension of $closure(x, y)$. It loads the result into its memory, transforms it into its local data structure and displays it on the screen as in figure 2 (b).

Now consider the following updates on the base relation *edge*: a `delete` operation for the tuple $\{edge(b, c)\}$; and an `insert` operation with the tuple $\{edge(h, d)\}$.

The recent literature on incremental view maintenance seems to converge on a three-step consensus procedure [9, 6] that computes view differentials for a wide range of view definitions, including negation and aggregate functions:

- **estimate the consequences of deletions**
  The deletion cuts off the relationship between nodes b and c which results in an elimination of the links $\{(a, c), (a, g), (b, c), (b, g)\}$.
- **prune those derived deletions for which alternative derivations exist**
  Despite of the cut link (b, c) e and f are still connected with c and g since there are additional corresponding paths in the graph via node d.
- **add the consequences of insertions**
  The insertion yields three new links $\{(h, d), (h, c), (h, g)\}$.

Some special cases allow for faster counting algorithms, some complex cases (e.g. duplicates) require additional treatment. While most formal results have been developed in a Datalog context, results have also been transfered to active relational databases [2], multi-databases and data warehouses [23, 10].

3 Incremental Maintenance of Materialized Views: A Declarative Solution

In this section, we present the first group of algorithms which assumes that views are in fact materialized not only in the external client but also in the server.

Our algorithm (section 3.1) follows the three-step approach just shown. However, it rewrites the original view definitions to a *purely declarative program of maintenance rules* rather than one that is mixed with procedural steps [6]. We also need less assumptions than [9] who uses meta predicates for control and additionally requires not only the old state but also the new state of all extensional *and* intensional relations to be completely available.

3.1 Generating Maintenance Rules

Generating view maintenance rules requires rewriting the original view definitions in order to incorporate differentials in the rule bodies and heads. A given view definition

$(O): p :- r_1, \ldots, r_n.$

is rewritten to a set of maintenance rules whose evaluation will compute the set of implicit insertions and deletions to be propagated. The rules are formulated in a way that mimicks the rough algorithm mentioned in the previous section.

The first subset of maintenance rules consists of the following rules ($1 \leq i \leq n$):

$(D_i): p^{del} :- r_1, \ldots, r_{i-1}, r_i^{del}, r_{i+1}, \ldots, r_n.$
$(N_1): p^{new} :- p, \neg p^{del}.$

The (D_i) rules derive all possible deletions of tuples in relation p caused by deleted tuples in body relations. If r_i is a base predicate, r_i^{del} contains the explicitly deleted facts in r_i otherwise it contains a superset of the tuples to be deleted in r_i due to deletions caused by other rules. The rule (N_1) computes those tuples that definitely remain for p in the new database state. For all base predicates among the r_i a rule (N_i') does exactly the same[2]:

$(N_i'): r_i^{new} :- r_i, \neg r_i^{del}.$

Rule R checks which tuples in p^{del} have alternative derivations on the "minimal" new database state gained so far. Those tuples are put back into p^{new} by rule (N_2).

$(R): p^{red} :- p^{del}, r_1^{new}, \ldots, r_n^{new}.$
$(N_2): p^{new} :- p^{red}.$

The next rules propagate insertions of base relation tuples towards the intensional relations. This is done by ordinary semi-naive rewriting, i.e. by constructing rules (I_i) that join new tuples inserted into one body relation with full extensions of all others. The newly derived insertions in addition have to be put by (N_3) into the new state of p. The same must also be done for all base predicates among the r_i by rules (N_i'').

$(I_i): \quad p^{ins} :- r_1^{new}, \ldots, r_{i-1}^{new}, r_i^{ins}, r_{i+1}^{new}, \ldots, r_n^{new}.$
$(N_3): \quad p^{new} :- p^{ins}.$
$(N_i''): \quad r_i^{new} :- r_i^{ins}.$

Finally, the net insertions and deletions have to be computed by relations p^{plus} and p^{minus}. Those tuples with only *additional* derivation paths are no real (but *idle*) insertions. Tuples loosing a derivation path but still being supported by others or even newly introduced ones are no real (but *phantom*) deletions. Note, that the other two types of update abnormalities, namely idle deletions and phantom insertions are already prevented by rule (N_1) resp. may arise in case of negated body predicates only. The latter case is discussed below and can be overcome by stratification and a suited initialization of the *ins* and *del* predicates on each evaluation layer.

$(E_1): p^{plus} :- p^{ins}, \neg p.$
$(E_2): p^{minus} :- p^{del}, \neg p^{ins}, \neg p^{red}.$

Example 1 (Generating view maintenance rules)
We continue the example from section 2.2 by rewriting rule R_1 as follows:

$(D_1): closure^{del}(x,y) :- edge^{del}(x,y).$
$(N_1): closure^{new}(x,y) :- closure(x,y), \neg closure^{del}(x,y).$
$(N_1'): edge^{new}(x,y) :- edge(x,y), \neg edge^{del}(x,y).$
$(R): closure^{red}(x,y) :- closure^{del}(x,y), edge^{new}(x,y).$
$(N_2): closure^{new}(x,y) :- closure^{red}(x,y).$
$(I_1): closure^{ins}(x,y) :- edge^{ins}(x,y).$
$(N_3): closure^{new}(x,y) :- closure^{ins}(x,y).$
$(N_1''): edge^{new}(x,y) :- edge^{ins}(x,y).$
$(E_1): closure^{plus}(x,y) :- closure^{ins}(x,y), \neg closure(x,y).$
$(E_2): closure^{minus}(x,y) :- closure^{del}(x,y), \neg closure^{ins}(x,y), \neg closure^{red}(x,y).$

Transformation of rule R_2 yields the following additional rules[3]

$(D_1): closure^{del}(x,y) :- edge^{del}(x,z), closure(z,y).$
$(D_2): closure^{del}(x,y) :- edge(x,z), closure^{del}(z,y).$
$(R): closure^{red}(x,y) :- closure^{del}(x,y), edge^{new}(x,z), closure^{new}(z,y).$
$(I_1): closure^{ins}(x,y) :- edge^{ins}(x,z), closure^{new}(z,y).$
$(I_2): closure^{ins}(x,y) :- edge^{new}(x,z), closure^{ins}(z,y).$

□

Algorithm 1 summarizes the transformations discussed so far.

Algorithm 1 (Generating maintenance rules)
Input: A rule of the form $(O): p :- r_1, \ldots, r_n.$
Output: A set M of maintenance rules for (O).

[2] A corresponding rule for intensional predicates is generated when compiling one of their defining rules.

[3] The rules (N_i),(N_i'),(N_i'') and (E_i) are the same for both R_1 and R_2.

```
begin
M := ∅;
for i:= 1 to n do
       generate rules (D_i) and (I_i);
       M := M ∪ {(D_i), (I_i)} od;
for j:= 1 to n
       if r_j is base relation then do
            generate rules (N'_j) and (N''_j);
            M := M ∪ {(N'_j), (N''_j)} od
generate rules (N_1), (N_2), (R), (N_3), (E_1), (E_2);
M := M ∪ {(N_1), (N_2), (R), (N_3), (E_1), (E_2)};
return M
end
```
□

The following theorem shows that evaluating the rewritten rules generated by algorithm 1 is a sound and complete procedure for computing the differentials between the database states before and after an extensional update operation takes place.

Theorem 1 *Let S^{old} and S^{new} be the old resp. new database state concerning a given set of base data updates. Moreover, assume S_p^{old} and S_p^{new} to be the tuples belonging to a relation p, where S_p^{old} is materialized for each p as its extension. Then for each p the evaluation of the rules generated by algorithm 1 yields the exact positive and negative differentials S_p^{plus} and S_p^{minus} as extension of p^{plus} and p^{minus} such that $S^{new} = S^{old} \setminus S^{minus} \cup S^{plus}$ and $S^{minus} \subseteq S^{old} \wedge S^{old} \cap S^{plus} = \emptyset$ where $S^{plus} := \bigcup_p S_p^{plus}$ and $S^{minus} := \bigcup_p S_p^{minus}$.*

Proof. The proof is given in [17]. □

3.2 Evaluation with Access to View Caches

The maintenance rules in example 1 make obvious that it is necessary to access the old contents of *closure* before the base data update operations took place. Bottom-up evaluation approaches like [6] therefore require that the intensional relations involved are completely materialized. The view maintenance process then consists of an evaluation of the generated maintenance rules, without using the initial view definitions.

Some of the generated rules contain negated predicates in the body even though the original rules were pure Datalog rules. For evaluating these rules the predicates have to be partitioned into strata such that no two predicates in one stratum depend negatively on each other and predicates may only called negatively by predicates of a higher stratum. Note, that algorithm 1 guarantees stratifiability because the transformation itself keeps this property and the newly introduced predicates may not cause side effects with other rules. The evaluation proceeds stratum-by-stratum starting with the extensional predicates.

Example 2 (Evaluating view maintenance rules)

For our example the following strata can be obtained:
$S_0 = \{edge, closure, edge^{del}, edge^{ins}\}$
$S_1 = \{closure^{del}, edge^{new}\}$
$S_2 = \{closure^{new}, closure^{red}, closure^{ins}, closure^{plus}\}$
$S_3 = \{closure^{minus}\}$

Let $edge = \{(f,e),(e,d),(e,a),(a,b),(d,c),(b,c),(c,g)\} \cup edge'$ as in subsection 2.2 where $edge'$ (with closure $edge'^*$) represents the independent subgraph G' (G'^*, resp.). The transitive closure of *edge* is the relation *closure* as displayed in figure 2. The extensional update relations are given by $edge^{del} = \{(b,c)\}$ and $edge^{ins} = \{(h,d)\}$.

S_1:
It. 1: $closure^{del} := \{(b,c),(b,g)\}$
$edge^{new} := \{(f,e),(e,d),(e,a),(a,b),(d,c),(c,g)\}$
It. 2: $closure^{del} := closure^{del} \cup \{(a,c),(a,g)\}$
It. 3: $closure^{del} := closure^{del} \cup \{(e,c),(e,g)\}$
It. 4: $closure^{del} := closure^{del} \cup \{(f,c),(f,g)\}$

S_2:
It. 1: $closure^{new} = \{(c,g),(d,g),(d,c),(e,d),(e,a),(e,b),(f,e),(f,a),(f,b),(f,d)\} \cup edge'^*$
$closure^{ins} := \{(h,d),(h,c),(h,g)\}$
It. 2: $closure^{red} = \{(e,c),(e,g)\}$
$closure^{new} := closure^{new} \cup \{(e,c),(e,g)\}$
$closure^{\mathbf{plus}} := \{(h,d),(h,c),(h,g)\}$
It. 3: $closure^{red} := closure^{red} \cup \{(f,c),(f,g)\}$
$closure^{new} := closure^{new} \cup \{(f,c),(f,g)\}$

S_3:
$closure^{\mathbf{minus}}(x,y) := \{(a,c),(a,g),(b,c),(b,g)\}$

Hence, the result is the same as in fig. 2(b). □

4 Incremental View Maintenance with Rederivation on Demand: The RoD Algorithm

We now turn to the case where the view is materialized only externally. Change propagation in this case requires partial re-derivation of the externally materialized views on demand. To achieve this, we further rewrite the maintenance rules from section 3 into a set of triggers. While section 4.1 describes these steps for standard Datalog section 4.2 shows that adding negation is not a major problem. Since view maintenance and deductive query processing can both handle aggregate functions like negation [6, 11], our approach covers a fairly large class of view definition languages. An evaluation algorithm that jointly exploits the maintenance and the query rules for view maintenance at runtime is presented in section 4.3.

4.1 Rederivation on Demand: RoD

For evaluating rules $(D_i), (R), (I_i), (E_1)$, it is necessary to access both the extensional and the intensional relations of the old database state (either directly or indirectly). If the view caches are not accessible, we need a further transformation of the maintenance rules which rederives the *relevant* parts of the intensional relations on demand.

The basic idea is similar to the supplementary magic set algorithm [21]: we need a triggering fact in each rule that enables firing *and* propagates constants. The magic set algorithms employ magic predicates for this purpose. For maintenance rules, this role can be played by the newly introduced update predicates p^{del} and p^{ins}.

4.1.1 Rewriting deletion rules

Starting with a rule

$(D_i) : p^{del} :- r_1, \ldots, r_{i-1}, r_i^{del}, r_{i+1}, \ldots, r_n.$

the insertion of tuples into r_i^{del} means instantiating the arguments of r_i^{del} and of all predicates r_j that share variables with r_i^{del}. Hence, their bindings have to be propagated from right to left towards r_1, as well as from left to right towards r_n. This propagation corresponds to a computation of joins in a given order[4]. The join results for each (D_i) are expressed as supplementary derived relations:

$(D_i^{i-1}) : sup_{i-1}^I :- r_{i-1}, r_i^{del}.$
$(D_i^{i-2}) : sup_{i-2}^I :- r_{i-2}, sup_{i-1}^I.$
...
$(D_i^1) : sup_1^I :- r_1, sup_2^I.$
$(D_i^{i+1}) : sup_{i+1}^I :- r_i^{del}, r_{i+1}.$
$(D_i^{i+2}) : sup_{i+2}^I :- sup_{i+1}^I, r_{i+2}.$
...
$(D_i^n) : sup_n^I :- sup_{n-1}^I, r_n.$

Finally, the two "streams" from r_i^{del} to r_1 and to r_n have to be joined:

(D_i^i) : $p^{del} :- sup_1^I, sup_n^I.$ (if $j \notin \{1, n\}$)
$p^{del} :- sup_n^I.$ (if $j = 1$)
$p^{del} :- sup_1^I.$ (if $j = n$)

Informally, the head arguments of each D_i^j ($j \neq i$) are those variables that do not occur further left ($j < i$) or right ($j > i$) with respect to r_i and are not needed for performing the final join. More precisely the arguments A of sup_j^I are given as follows:

if $j < i - 1$ A contains all variables v of r_j and sup_{j+1}^I with $\exists_{i<k\leq n}$ v appears in r_k or $\exists_{1\leq k<j}$ v appears in r_k or v appears in p^{del}

if $j > i + 1$ A contains all variables v of sup_{j-1}^I and r_j with $\exists_{1\leq k<i}$ v appears in r_k or $\exists_{j<k\leq n}$ v appears in r_k or v appears in p^{del}

if $j = i - 1$ A contains all variables v of r_j and r_i^{del} with $\exists_{i<k\leq n}$ v appears in r_k or $\exists_{1\leq k<i-1}$ v appears in r_k or v appears in p^{del}

if $j = i + 1$ A contains all variables v of r_i^{del} and r_j with $\exists_{1\leq k<i}$ v appears in r_k or $\exists_{i+1<k\leq n}$ v appears in r_k or v appears in p^{del}

The magic-set transformation of a rule

$$r :- g_1, \ldots, g_n$$

introduces so-called magic predicates m_g_i for each intensional body predicate g_i that a) propagate variable bindings from other body literals and b) initiate the derivation of all relevant tuples of g_i satisfying these bindings. This is done by inserting the magic predicates m_g_i as a kind of guard in each rule defining g_i, and by generating additional magic rules (with m_g_i as conclusion) that fire whenever a subset of g_i is needed for some join computation.

We assume that the original view definitions have already been transformed to query rules with the supplementary magic-set algorithm. In order to have the relevant tuples in r_j of the old database state available for the maintenance rules (D_i^j), we have to generate further magic rules $(M_{D_i}^j)$ that trigger the defining query rules for each r_j. The query rules can then derive exactly the needed set of tuples, using a standard DBMS query processor to compute the joins[5].

The right hand side of each magic rule consists of the join result gained from the preceding subgoals which is intended to be joined with r_j. The magic predicate m_r_j at the left hand side takes over only those arguments from r_j that are either constants or variables appearing and bound by the right hand side. As usual the m_r_j are adorned, i.e. marked with a pattern consisting of a sequence of b's and f's for each position in r_j depending on whether the argument is left out (f) or not (b).

$(M_{D_i}^{i+1}) : m\_r_{i+1} :- r_i^{del}$
$(M_{D_i}^{i-1}) : m\_r_{i-1} :- r_i^{del}$
$(M_{D_i}^{i-2}) : m\_r_{i-2} :- sup_{i-1}$
...
$(M_{D_i}^1) : m\_r_1 :- sup_2$
$(M_{D_i}^{i+2}) : m\_r_{i+2} :- sup_{i+1}$
...
$(M_{D_i}^n) : m\_r_n :- sup_{n-1}$

These magic rules ensure that only those parts of the (old) extensions of r_j are derived that are relevant

[4] As the example at the end of this section shows, almost no rewriting is necessary when $n = 1$ or $n = 2$. The adaption of rewriting is obvious and not discussed in detail for space reasons.

[5] Note, that for the extensional relations among the r_j magic rules are of course not required since they are directly accessible.

for the join with r_i^{del}. As a consequence evaluating the rules

$(N_1) : p^{new} :- p, \neg p^{del}.$
$(N_i') : r_i^{new} :- r_i, \neg r_i^{del}.$

yields only a partial extension of p and r_i in the new database state p^{new} and r_i^{new} respectively.

4.1.2 Rewriting rederivation rules

The rederivation step (using rule R) for tuples that have an alternative derivation path starts with the existing new partial extensions p^{new} and r_j^{new}.

$(R) : p^{red} :- p^{del}, r_1^{new}, \ldots, r_n^{new}.$
$(N_2) : p^{new} :- p^{red}.$

However, it may happen that the join between tuples of the overestimate p^{del} requires materialization of additional tuples from the old states of p and the r_i. Therefore, we need a triggering mechanism which propagates bindings from p^{del} to the r_i^{new} predicates. This can be reached by rewriting (R) as follows:

$(R^1) : sup_1^{II} :- p^{del}, r_1^{new}.$
$(R^2) : sup_2^{II} :- sup_1^{II}, r_2^{new}.$
...
$(R^{n-1}) : sup_{n-1}^{II} :- sup_{n-2}^{II}, r_{n-1}^{new}.$
$(R^n) : p^{red} :- sup_{n-1}^{II}, r_n^{new}.$

Again we have to ensure that all tuples needed to join sup_{j-1}^{II} and r_j^{new} are available in r_j^{new}. This can be done by rederiving the candidate tuples from the old database state, i.e. in r_j. This derivation is initiated by corresponding magic rules (M_R^j):

$(M_R^1) : m\_r_1 :- p^{del}.$
$(M_R^2) : m\_r_2 :- sup_1^{II}.$
...
$(M_R^n) : m\_r_n :- sup_{n-1}^{II}$

Rules (N_1) and (N_i') then copy all tuples from r_j that definitely remain in the new database state into r_j^{new}. The overestimate of deletions r_j^{del} used by these rules which leads to potentially missing tuples[6] for r_j^{new} is corrected by rule (N_2) generated for the rules defining r_j.

4.1.3 Rewriting insertion rules

The next step of rewriting the maintenance rules concerns the rule set (I_i) which incrementally propagates the insertions. As already mentioned the rewriting from the view definition (O) to the set (I_i) is exactly standard semi-naive rewriting and has now to be supplemented by corresponding magic-set transformations that are essentially the same as the transformations for the rule set (D_i) with the difference that the body literals now access the new database state.

$(I_i^{i-1}) : sup_{i-1}^{III} :- r_{i-1}^{new}, r_i^{ins}.$
$(I_i^{i-2}) : sup_{i-2}^{III} :- r_{i-2}^{new}, sup_{i-1}^{III}.$
...
$(I_i^1) : sup_1^{III} :- r_1^{new}, sup_2^{III}.$
$(I_i^{i+1}) : sup_{i+1}^{III} :- r_i^{ins}, r_{i+1}^{new}.$
$(I_i^{i+2}) : sup_{i+2}^{III} :- sup_{i+1}^{III}, r_{i+2}^{new}.$
...
$(I_i^n) : sup_n^{III} :- sup_{n-1}^{III}, r_n^{new}.$
$(I_i^i) : p^{ins} :- sup_1^{III}, sup_n^{III}.$

Since some necessary tuples may be not available in the partially new states r_i^{new}, we have again to rederive additional parts of the old state through suited magic rules $(M_{I_i}^j)$:

$(M_{I_i}^{i+1}) : m\_r_{i+1} :- r_i^{ins}$
$(M_{I_i}^{i-1}) : m\_r_{i-1} :- r_i^{ins}$
$(M_{I_i}^{i-2}) : m\_r_{i-2} :- sup_{i-1}^{III}$
...
$(M_{I_i}^1) : m\_r_1 :- sup_2^{III}$
$(M_{I_i}^{i+2}) : m\_r_{i+2} :- sup_{i+1}^{III}$
...
$(M_{I_i}^n) : m\_r_n :- sup_{n-1}^{III}$

The rules (N_1) and (N_2) for each r_i then ensure that only those tuples from the rederived set go into r_i^{new} that were not deleted or at least were rederived.

4.1.4 Rewriting idle checks

The last rule to be looked at in order to provide on-demand access to the old database state is (E_1) which computes the net insertions as difference between p^{ins} and p. This rule checks whether a tuple derived as to be inserted into p^{new} in fact was not already in p before, i.e. is not an idle insertion. The check can simply be realized by adding the following magic rule:

$(M_{E_1}) : m\_p :- p^{ins}.$

This rule guarantees that all idle insertions are caught by trying to rederive them in the old database state.

Based on these transformation steps, the complete transformation procedure of a rule into a set of maintenance rules is given by algorithm 2.

Algorithm 2 (The RoD algorithm)
Input: A rule of the form $(O) : p :- r_1, \ldots, r_n$.
Output: A set M' of maintenance rules for (O).

```
begin
generate from (O) a rule set M by algorithm 1;
M' := M;
for i:= 1 to n do
 for j:= 1 to n do
   generate from (D_i) ∈ M' rules (D_i^j) and (M_{D_i}^j);
   generate from (I_i) ∈ M' rules (I_i^j) and (M_{I_i}^j);
   M' := M' ∪ {(D_i^j),(M_{D_i}^j),(I_i^j),(M_{I_i}^j)} od;
 M' := M' \ {(D_i),(I_i)} od;
for i:= 1 to n do
 generate from (R) ∈ M rules (R^i) and (M_R^i);
 M' := M' ∪ {(R^i),(M_R^i)} od;
M' := M' \ {(R)};
generate from (E_1) ∈ M' (M_{E_1});
M' := M' ∪ {(M_{E_1})};
return M'
end                                                    □
```

[6] Of course only if a rule for r_j exists.

maintenance rules generated from R_1:

$(D_1) \Rightarrow closure^{del}(x,y) :- edge^{del}(x,y).$
$(R) \Rightarrow closure^{red}(x,y) :- closure^{del}(x,y).$
$(I_1) \Rightarrow closure^{ins}(x,y) :- edge^{ins}(x,y).$
$(E_1) \Rightarrow closure^{plus}(x,y) :- closure^{ins}(x,y), \neg closure(x,y).$
$m\_closure^{bb}(x,y) :- closure^{ins}(x,y).$

maintenance rules generated from R_2:

$(D_1) \Rightarrow closure^{del}(x,y) :- edge^{del}(x,z), closure(z,y).$
$m\_closure^{bf}(z) :- edge^{del}(x,z).$
$(D_2) \Rightarrow closure^{del}(x,y) :- edge(x,z), closure^{del}(z,y).$
$(R) \Rightarrow closure^{red}(x,y) :- sup_1^{II}(x,y,z), closure^{new}(z,y).$
$sup_1^{II}(x,y,z) :- closure^{del}(x,y), edge^{new}(x,z).$
$m\_closure^{bb}(z,y) :- sup_1^{II}(x,y,z).$
$(I_1) \Rightarrow closure^{ins}(x,y) :- edge^{ins}(x,z), closure^{new}(z,y).$
$m\_closure^{bf}(z) :- edge^{ins}(x,z).$
$(I_2) \Rightarrow closure^{ins}(x,y) :- edge^{new}(x,z), closure^{ins}(z,y).$

relevant query rules for R_1:

$closure(x,y) :- m\_closure^{bf}(x), edge(x,y).$
$closure(x,y) :- m\_closure^{bb}(x,y), edge(x,y).$

relevant query rules for R_2:

$closure(x,y) :- sup_1(x,z), closure(z,y).$
$sup_1(x,z) :- m\_closure^{bf}(x), edge(x,z).$
$m\_closure^{bf}(z) :- sup_1(x,z).$
$closure(x,y) :- sup_1'(x,y,z), closure(z,y).$
$sup_1'(x,y,z) :- m\_closure^{bb}(x,y), edge(x,z).$
$m\_closure^{bb}(z,y) :- sup_1'(x,y,z).$

Figure 3: RoD applied to the graph example

The following theorem states that evaluating the rewritten rules leads to an exact computation of the difference between two subsequent database states.

Theorem 2 *Let S^{old}, S^{new}, S^{plus},S^{minus} be defined as in Theorem 1, but now assume that only the extensions of base data relations are available. Then for each relation p the evaluation of the rules generated by algorithm 2 in combination with the magic-set rewritten original (query) rules yields the exact positive and negative differentials S_p^{plus} and S_p^{minus} as extension of p^{plus} and p^{minus}. During the evaluation only those tuples from S_p^{old} resp. S_p^{new} are (re)derived in p and p^{new} that are indispensable for determining the differentials.*

Proof. The proof is given in [17]. □

Example 3 (Applying RoD)

Figure 3 shows the resulting rule set of applying RoD to rules R_1 and R_2, and relevant query rules. Since both original rules have only one or two subgoals, the transformation of (D_i),(R) and (I_i) generates only a few additional magic rules. The other rules remain unchanged. As mentioned above, the magic predicates are adorned with variable bindings. □

4.2 RoD with Negation

The algorithm RoD needs only to be changed slightly to allow negation in view definitions. In addition, the evaluation has to be performed with a slightly changed control structure that respects stratification of the original rule set. The first change to RoD concerns the effect of updates for predicates r_i that **occur negatively** in a rule (O):

$$(O) : p :- r_1, \ldots, \neg r_i, \ldots, r_n.$$

Insertions into r_i lead to possible deletions of the rule head p. Deletions from r_i may allow now tuples for p to be derivable that were prevented before by the existence of certain tuples in r_i.

A consequence of this observation is a modified generation of the basic maintenance rules (D_i) and (I_i) in algorithm 1 for those r_i that occur negatively in (O):

$(D_i) : p^{del} :- r_1, \ldots, r_{i-1}, r_i^{ins}, r_{i+1}, \ldots, r_n.$
$(I_i) : p^{ins} :- r_1^{new}, \ldots, r_{i-1}^{new}, r_i^{del}, r_{i+1}^{new}, \ldots, r_n^{new}.$

In the generated rules (D_j) , (I_j) with $j \neq i$ as well as in (R) the predicate r resp. r^{new} keeps its negative sign. All other rules handle negated predicates without respecting their signs.

From an evaluation point of view, the magic set rewriting with supplementary relations fixes the execution order of joins, in a manner determined by the formulation of the original rules. Due to the commutativity of the AND operator, the ordering of literals and thus the execution order is in principle free and could be re-organized according to some cost-based optimization. This freedom becomes restricted in the presence of negation, as negation leads to a combination of joins and set differences in the implementation.

Safeness of rules restricts variables occuring in negated literals to be bound in a positive predicate. Only then, there always exists some order for join computation such that joins with relations refered to by negated predicates can be performed straightforward by standard set difference. Hence, $r(\vec{X},\vec{Y}) \bowtie \neg\, s(\vec{Y})$ is evaluated by $(\prod_{\vec{Y}} r(\vec{X},\vec{Y}) \backslash s(\vec{Y})) \bowtie r(\vec{X},\vec{Y})$, i.e. r is projected onto the columns of s, the difference between the result and s is computed and joined with r. Of course, the arguments of s must be a subset from those of r.

In our approach, the ordering is determined for the transformation of (D_i) and (I_i) by the position of the body predicate r_i respectively its delta variant r_i^{del} and r_i^{ins}. From there the join sequence is built up to the left *and* right. As in the general case, negated predicates have to be moved such that the transformation doesn't destroy the applicability of joins. In our case, we could simply move all negated predicates to the right. After processing the positive literals[7] and joining both sequences together (rules (D_i^i) and (I_i^i)), the negative literals are processed from left to right by corresponding joins with ordinary set difference[8].

The third specific aspect for handling negation is **evaluation control**. The solution is straightforward:

[7] Note, that we don't have to deal with negated delta predicates.

[8] Another solution would be to give up the idea of propagating bindings starting from the delta literal to both sides and to move the delta literal to the beginning of the body literal sequence as it is already when transforming rule R. Then we have the same situation as with standard magic-set.

the original rule set is partitioned into strata with respect to their head predicates: if a rule calls a predicate negatively then the defining rules for that predicate belong to a lower stratum. The view maintenance process then starts with the lowest stratum and computes the net insertions p^{plus} and net deletions p^{minus} of its defined predicates p. At the next layer, these predicates are handled like extensional predicates and their update relations p^{ins} and p^{del} are initialized with the just derived net updates. One effect of this initialization is the prevention of phantom insertions. Each layer only has to deal with the maintenance rules generated for its own rules. One accompanying additional step for RoD is therefore to generate rules (N_i') and (N_i'') not only for extensional body predicates but also for predicates belonging to lower strata.

4.3 Evaluation without Access to View Caches

The layered view maintenance process and its interplay with query evaluation for rederiving the relevant parts of the old database state are summarized by algorithm 3. This algorithm employs the notion of an *environment* ENV as a mapping from predicate symbols to sets of tuples that represent a partial database state. This state is the subset of the overall EDB and IDB contents which needs to be looked at for maintaining the views of interest. The rule sets M and Q denote the rewritten rules used for view maintenance and query evaluation, respectively. Recall that we assume Q to be derived from the original rules by standard supplementary magic-set rewriting (with respect to all possible binding patterns) such that the link between both rule sets is provided by the magic rules generated by RoD. Whereas EDB and IDB denote the extensional and intensional predicates as before, SUP contains all delta, supplementary and magic predicates introduced for M and Q. Each $p \in EDB \cup IDB$ has a unique stratum number $S(p)$ between 0 and some constant m. In addition we define $M^{(n)}$ as the set of rules $r \in M$ such that r is based on an original rule defining a predicate $p \in EDB \cup IDB$ with $S(p) = n$.

The algorithm assumes $EVAL$ to be a fixpoint evaluator that works on a stratified set of rules and an environment with initializations for the relations involved. $EVAL$ respects the changes of the environment produced during the preceding evaluation in a semi-naive manner and returns its with additional tuples inserted for certain relations. When called for evaluating the maintenance rules $EVAL$ only has to process the subset $M^{(i)}$ for a given stratum i of the original rule set. The second call with Q, however, is exactly as for evaluating an arbitrary query but now on a partially materialized intensional database state. The new tuples for the magic predicates generated in the first call of $EVAL$ denote queries that have to be answered in order to continue the maintenance process. These queries may have to rederive tuples for predicates from lower strata that were not rederived before since they were not needed. Therefore, $EVAL$ is called with the complete set of query rules Q.

Algorithm 3 (View maintenance procedure)
Input: 1. A set of query rules Q
2. A set of maintenance rules M
3. A set of view names $V = \{v_1, \ldots, v_n\}$
4. A set of base data changes

Output: changes to the views in V

```
begin
for r ∈ IDB ∪ SUP do
   initialize ENV[r] := ∅ od;
for r ∈ EDB do
   initialize ENV[r^del] and ENV[r^ins] with the
    base data changes;
   initialize ENV[r] with the old extension
    of r od;
for i := 1 to m do
   repeat
      OLD := ENV;
      ENV := EVAL(ENV, M^(i));
      ENV := EVAL(ENV, Q)
   until ENV = OLD;
   for r ∈ IDB with S(r) = i do
      ENV[r^ins] := ENV[r^plus];
      ENV[r^del] := ENV[r^minus] od od;
MOD := [(ENV[v_1^plus], ENV[v_1^minus]), ...,
        (ENV[v_n^plus], ENV[v_n^minus])];
return(MOD)
end
```
□

Example 4 (View maintenance for the graph example)

If we apply algorithm 3 to the graph example in order to maintain *closure*, the rules shown in figure 3 and the base data updates ($edge^{del}(b,c)$,$edge^{ins}(h,d)$) serve as input. Since no negation occurs in the original rules, the evaluation enters the `repeat/until` loop only once. Query rules can be evaluated without stratification, too. In contrast to the evaluation trace for the maintenance rules in example 2, *closure* now does not belong to stratum S_0 but to S_1 together with the new predicates sup_1^{II} and $m\_closure^{bf}$. $m\_closure^{bb}$ goes into S_2 since it depends on $closure^{ins}$. Without demonstrating the complete evaluation it can be stated that exactly the relevant part of *closure* that represents the complement of subgraph G' in section 2.2 is rederived. □

5 View Monitoring in ConceptBase

ConceptBase [7] is a deductive object manager for meta data management which supports the O-Telos

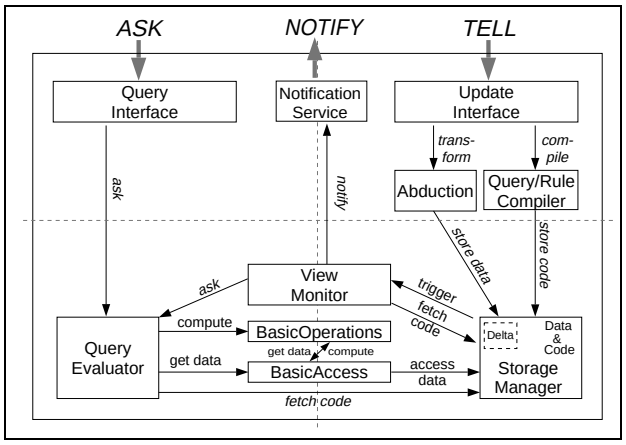

Figure 4: The ConceptBase server architecture

object model [12]. Textual and graphical user interface tools are linked to ConceptBase servers as clients over the Internet. We therefore experienced the problems addressed in this paper since the first uses of ConceptBase as an Internet-based cooperative modeling tool in the late 1980's.

An O-Telos object base is semantically equivalent to a deductive database (Datalog with negation) which includes a predefined set of rules and integrity constraints coding the object structure. The surface language syntax is frame-based or uses semantic networks. Rules and integrity constraints are included as first-order formulas defined over a basic set of predicates describing the abstraction principles of instantiation, specialization and attribution.

The ConceptBase server architecture [7] is shown in figure 4. The `RoD` algorithm 2 is part of the ConceptBase `Query/Rule-Compiler` component while algorithm 3 constitutes the `ViewMonitor`.

Figure 5 demonstrates the incremental view maintenance process for an example database of software modules, with three classes `Module`,`Procedure` and `OperatingSystem`. Procedures are `defined_in` modules and modules `import` procedures; they may `depend_on` particular operating systems.

Views and queries in ConceptBase are specified as classes of derived data (keywords `QueryClass`, `View`) with necessary *and* sufficient membership constraints [19]. A view `ModuleDependency` links (`based_on`) modules with those other modules from which they directly or indirectly (via transitive closure) import procedures. A second view `IllegalOS` maintains violations of an integrity constraint. It describes incompatible `based_on` relationships which contain procedures that have a `depend_on` link to different operating systems.

The `Query/Rule-Compiler` of ConceptBase maps both view definitions to intensional relations *mod_dep* and *ill_os* defined by deductive rules as follows:

$$mod\_dep(x,y) \;:-\; module(x), module(y), procedure(p) \\ import(x,p), defined\_in(p,y).$$

$$mod\_dep(x,y) \;:-\; mod\_dep(x,z), mod\_dep(z,y).$$

$$ill\_os(x,y,z) \;:-\; mod\_dep(x,y), mod\_dep(x,z), \\ op\_sys(o_1), op\_sys(o_2), procedure(p_1), \\ procedure(p_2), unequal(o_1,o_2), \\ defined\_in(p_1,y), \\ defined\_in(p_2,z), \\ depend\_on(o_1), depend\_on(o_2).$$

The `ViewMonitor` works on such an internal representation of views, queries and rules and provides notification messages to those applications affected indirectly by updates of others. Integrity views like `IllegalOS` do not necessarily lead to rejections of updates if their extension becomes non-empty.

In figure 5, the left graph browser application displays the contents of the view `ModuleDependency` based on the current extensional database state (displayed by the browser at the top). The right graph-browser window shows the extension of `IllegalOS`.

A check-in of some source module had the effect of introducing a new `import` link between module `A` and procedure `time` which leads to an operating system conflict with `directory` defined in module `C` on which `A` is based, too. This update also induced a `based_on` link between `A` and `E` in `ModuleDependency`. Both changes have already been notified to the graphical displays (number tag 2).

A second update is caused by a new module `D` which imports a procedure `diff` from `A` (number tag 3). Both externally materialized views have to be updated by inserting `based_on` links between `D` and the other modules into `ModuleDependency`, and by inserting `D` together with `A` and `E` into `IllegalOS`.

To get a feeling for the performance impact of the approach, the following table compares response times for complete recomputation of views and incremental maintenance with `RoD` with respect to a database that contained descriptions of 283 modules with 1630 exported procedures. The update operation was an newly inserted import link from a module to a procedure. The results indicate significant advantages for `RoD` in complex, recursive view definitions and almost no difference for simple ones.

	View	recomp.	incr.
1	All modules M is based on together with all directly or indirectly imported procedures	25 sec	4.5 sec
2	All procedures with a given name prefix imported by M	0.3 sec	0.25 sec
3	All procedures imported by M with less than 10 lines of code	4.5 sec	0.3 sec
4	All procedures with a given name imported by M directly or indirectly	4.4 sec	0.5 sec

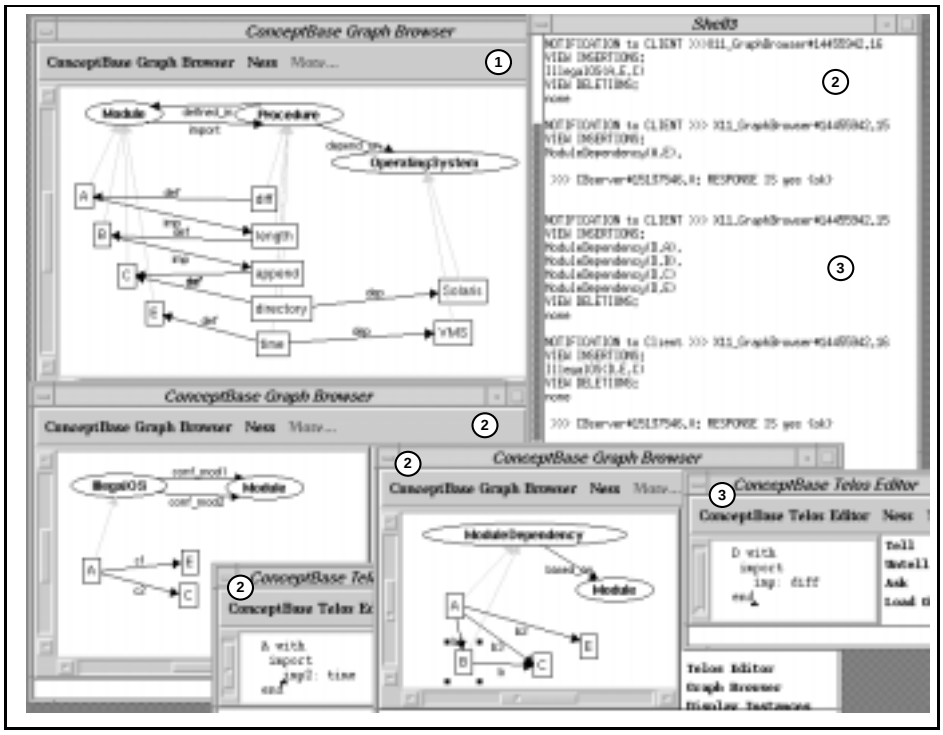

Figure 5: View monitoring in ConceptBase: an example

Another case where RoD proved very beneficial is the parallel maintenance of *many* views or integrity constraints, e.g. in design applications. As already suggested a decade ago [16], presenting violations in integrity views [19] is preferable to just rejecting updates not only because of better explanation but also because it enables multiple levels of integrity enforcement. For example, in a commercial application of ConceptBase [14], a business process analysis is monitored by more than 80 integrity views. While all of them need to be monitored continuously, they are reacted to at different periods in the analysis process. Corrections in one view may indirectly correct other violations, or cause new ones. In such a setting, non-incremental view maintenance may become prohibitively expensive.

6 Discussion and Outlook

Since the early papers [20, 1, 13, 15], the incremental maintenance of views has received a lot of attention in database research. The recent survey [5] is organized mostly according to the amount of information available to the maintenance tool. In the case of *full information* which we discussed in section 3, the base data, the materialized view, and the derivation rule can all be used.

Among the many conceivable cases of *partial information*, interest has focused on maintenance at the client side. In *self-maintainable views*, the view definition is so simple that all the consequences of a base-data change can be locally computed by the client, without accessing the base data [4].

Driven by our goal of offering a standard monitoring service for non-database clients, we mainly focused on a solution for the opposite case : *The maintenance tool has access to the base data and the view definition but not to the materialized view.* Though this may at first sound contradictory to the idea of materialization and has therefore hardly been studied, we argued that there will be many uses of such a service, including quite traditional ones such as integrity checking or user interface management.

Our solution extends known algorithms for the case of full information by a magic-set like rewriting of the generated maintenance rules such that the relevant parts of the externally materialized views can be rederived in the database on demand. As a prerequisite to the magic set transformation, the known algorithms had to be changed slightly such that they generate a pure stratified Datalog program of maintenance rules. We showed that during the evaluation phase the magic maintenance predicates created by our approach interoperate nicely with the magic query evaluation rules created when initially computing the external view materialization.

Finally, we summarized the implementation of our approach in ConceptBase, a deductive object manager for meta data management. Based on this implementation, some practical experience has been gained with flexible integrity maintenance in design applications.

Developing the view maintenance algorithms and linking them to existing base technology for databases and distributed systems is only one step on the way towards the effective monitoring service we envision. It is obvious that there is a space-time trade-off between the solutions presented in sections 3 and 4 which needs to be investigated quantitatively, including different options *how* to materialize: fully, by view indexes, with or without intermediate results, etc.

Secondly, the additional transformations required for complex-object views such as required for representing a whole graph as a view need to be integrated; for this purpose, a two-way transformation between Telos and a subset of C++ has been developed [18].

Finally, as mentioned earlier, we plan to combine the logical view maintenance approach discussed here to the work on relaxed coherency in replicated databases by which we can tailor quality of service.

Acknowledgements
The work described in this paper was partly supported by the Commission of the European Communities under ESPRIT BRA 6810 (Compulog 2). The authors would like to thank Christoph Quix and René Soiron for implementing the algorithms in ConceptBase.

References

[1] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *TODS*, 4(3):368–382, September 1979.

[2] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[3] R. Gallersdörfer and Nicola M. Improving performance of replicated databases through relaxed coherency. In *Proc. Int. Conf. on VLDB 1995*, 1995.

[4] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data integration using self-maintainable views. Technical Report 113880-941101-32, AT&T Bell Laboratories, November 1994.

[5] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering*, 18(2), June 1995.

[6] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Conference on Management of Data*, 1993.

[7] M. Jarke, R. Gallersdörfer, M. Jeusfeld, M. Staudt, and S Eherer. ConceptBase: A deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167 –192, March 1995.

[8] A.M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5:35–47, January 1996.

[9] V. Kuechenhoff. On the efficient computation of the difference between consecutive database states. In *Intl. Conf. on Deductive and Object-Oriented Databases*, 1991.

[10] J.J. Lu, Moerkotte G., J. Schue, and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 340–351, 1995.

[11] I. Mumick, H. Pirahesh, and R. Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, August 1990.

[12] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, Oktober 1990.

[13] J. M. Nicolas and K. Yazdanian. An outline of BD-GEN: A deductive DBMS. In *Proceedings of the triannual IFIP Conf 83, Mason(ed), N-H*, 1983.

[14] H.W. Nissen, M.A. Jeusfeld, M. Jarke, G.V. Zemanek, and H. Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, March 1996.

[15] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS. *IEEE Computer*, 19(12), December 1986.

[16] E. Simon. *Conception et Realisation d'un sous Systeme d'Intgerite dans un SGBD Relationnel*. PhD thesis, Universite de Paris VI, 1986.

[17] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. Technical Report AIB-95-13, RWTH Aachen, 1995.

[18] M. Staudt, M. Jarke, and C. Quix. Change notification for externally materialized application program views. Technical report, RWTH Aachen, March 1996. submitted for publication.

[19] M. Staudt, H.W. Nissen, and M.A. Jeusfeld. Query by class, rule and concept. *Applied Intelligence*, 4(2):133–156, 1994.

[20] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the ACM SIGMOD conference*, San Jose, CA, June 1975.

[21] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1/2*. Computer Science Press, Rockville, MD, 1989.

[22] G. Wiederhold and M. Genesereth. The basis for mediation. In *Proceedings of the 3rd Int. Conf. on Cooperative Information Systems*, pages 140–157, May 1995.

[23] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *1995 ACM SIGMOD International Conference on Management of Data*, 1995.