Incremental View Maintenance

JOHN V. HARRISON
Department of Computer Science
University of Queensland
Brisbane, QLD, 4072 Australia
E-mail: harrison@cs.uq.oz.au

SUZANNE W. DIETRICH
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287–5406, U.S.A.
E-mail: s.dietrich@asu.edu

Abstract

A view is a derived relation that is defined in terms of other relations. If a view is retained between references, as opposed to being reconstructed each time it is required, then it is termed *materialized.* One use of materialized views is to increase the speed of query processing. Updates made to stored relations that participate, directly or indirectly, in the definition of a view can cause the materialized view to be inconsistent with its definition. Optimally, the database system should be capable of updating a materialized view *incrementally* to reflect updates to the stored relations thereby avoiding the cost of reconstructing the materialized view.

View maintenance is also an important process in an active database when rules are defined that are to be triggered by updates to derived relations, i.e., views. In this context, it is the *updates* that are of interest, rather than an updated materialized view, since the updates have the potential to trigger the active rules. Storing the materialized view is unnecessary and incurring this cost may be undesirable.

This paper presents a novel update propagation algorithm for a deductive database (or a relational database supporting the proposed SQL3 standard) that can both incrementally maintain materialized views, and also trigger active rules that are monitoring the updates to views, *without* requiring the view to be materialized. The views can be defined using the relational operators, stratified negation and recursion. In certain cases, the algorithm can perform these tasks even when the view *definition* is modified. A series of optimizations are described that increase efficiency and have been implemented in a prototype system.

1 Introduction

A view is a derived relation that is defined in terms of stored relations and other derived relations. A view provides an interface to the database that can remain constant even after modifications to the database schema are performed. If the derived relation representing a view is reconstructed each time it is referenced it is considered *virtual.* If the derived relation is retained between references then it is termed *materialized.* Materialized views can be used to increase the speed of query processing. If queries repeatedly request access to a derived relation, then it is often advantageous to materialize the view. The benefit of materializing the view is obtained from amortizing the cost of constructing the derived relation over multiple queries.

Updates made to stored relations that participate in the view's definition can invalidate the derived relation representing the view. To resolve the invalidation, the view can be recomputed after the database is updated, i.e., the view can be discarded and materialized again using the current database state. This approach can be costly if the database is updated frequently. To avoid these costs, the materialized view itself can be updated incrementally to reflect the changes made to stored relations that comprise the view's definition.

In [TOM88], the problem of updating a materialized view is decomposed into three subproblems; namely the detection of *irrelevant updates*, the detection of *autonomously computable updates* and the problem of efficiently reevaluating the view. Irrelevant updates are updates to stored relations that cannot affect a derived relation. An autonomously computable update is one where all data necessary to update the view is contained within the update and the view itself. No direct access to the stored relations is necessary.

In [BLA89], these tests are applied when an update is presented to the system. An update is first examined to determine if it is irrelevant. If the update is not irrelevant it is then tested to determine if it is autonomously computable. If both of these tests fail, the view is re-computed differentially. Blakeley et al. focus primarily on the subproblems concerning the detection of irrelevant and autonomously computable updates. The views considered are restricted to *select-project-join (SPJ)* expressions.

In this work, a different approach is proposed. Instead of recomputing the view upon the failure of both the irrelevant and autonomously computable tests, an update propagation algorithm is used to incrementally compute the necessary updates to the materialized view without forcing complete recomputation. This work complements [BLA89] in that this algorithm can be employed to avoid the third subproblem, namely the efficient recomputation of the view. In addition, optimizations to the algorithm detect irrelevant updates, hence treating the first subproblem as well.

The update propagation algorithm, which is called *Extended Propagation/Filtration*, can maintain views that are expressed using all of the relational operators, stratified negation and recursion.[1] The *EPF* algorithm is more efficient than its predecessor, known as *PF* [HAR92a]. *EPF* is also capable of computing updates to views when the view *definition* is modified. This problem has received little attention in the research community. We address the case where a view definition contains one or more union operators and an operand of one or more of the unions is either added or removed from the definition.

[1]An extension describing support for *group stratified* aggregation appears in [HAR93].

This algorithm is unique because the updates to the view can be computed even when the view is *not* materialized. This characteristic allows *EPF* to be applied to the problem of *complex event detection* in an active database system[ROS89, CHA91, DIE92]. In an active database system, a user can define rules that are triggered by updates to stored relations. More sophistication is obtained if the user can define rules which are triggered by updates to *derived* relations, i.e., views. In this context, it is the updates that are of interest as they may trigger an active rule. Storing the materialized view is unnecessary and incurring this cost may be undesirable. The *EPF* algorithm can be employed to detect updates to derived relations, even those recursively defined, *without* having to incur the cost of materializing the derived relation.

The remaining sections of this paper are outlined below. Section 2 introduces notation and terminology. Section 3 describes the *EPF* algorithm and its support for stratified negation and view definition updates. Section 4 presents a series of optimizations that have improved the performance of our prototype. Finally, we compare the *EPF* algorithm to related algorithms, including the recently proposed *DRed* algorithm [GUP93].

2 Basic Concepts and Notation

The view maintenance approach described here relies on a procedure for computing the difference between two consecutive database states. This difference represents the changes that must be made to the initial database to obtain the updated database.

Assume that a database DB consists of a set of extensionally defined relations (EDB) and a set of intensionally defined relations (IDB). Let relation $P \in$ IDB and be defined by the predicate p. Let $\mathcal{U}_e$ be a set of updates to the EDB. The database state before $\mathcal{U}_e$ is performed is referred to as *old.* The database state after $\mathcal{U}_e$ is performed is referred to as *new.* Let the function *mat(IDB_Pred,DB_State)* compute an IDB relation defined by the predicate *IDB_Pred* using the EDB indicated by *DB_State* and the IDB. Let the difference between the materialization of an IDB relation in the old state and the materization of the same relation in the new state be termed the "delta set" (abbreviated Δset) for the relation. A delta set can be viewed as the updates that must be made to the old relation to obtain the new relation.

The notation ΔP represents the Δset for IDB relation P. A Δset consists of two distinct (possibly empty) subsets. The first, labeled ΔP_{add}, consists of tuples that must be added to the *old* relation to obtain the *new* relation. The second, labeled ΔP_{rem}, consists of tuples that must be removed from the *old* relation to obtain the *new* relation. These concepts are formalized using the definitions below, which assume that the predicates that define the IDB relations are not updated. Later, we address updates to the definitions of the IDB relations.

Definition. Let EDB_{Old} refer to an arbitrary EDB before a set of updates $\mathcal{U}_e$ are performed to the EDB relations. Let EDB_{New} refer to the same EDB after $\mathcal{U}_e$ are performed. Let p denote a predicate representing an arbitrary IDB relation P. Since we defer our discussion of IDB updates until a later section, let $IDB_{New} = IDB_{Old}$.

$$\Delta P_{rem} = mat(p, Old) - mat(p, New) \qquad DB_{Old} = EDB_{Old} \cup IDB_{Old}$$
$$\Delta P_{add} = mat(p, New) - mat(p, Old) \qquad DB_{New} = EDB_{New} \cup IDB_{New}$$
$$\Delta P = \{\Delta P_{rem}, \Delta P_{add}\}$$

□

The view maintenance approach described here compute the updates, i.e., Δsets, to IDB relations that the user has requested to be materialized. Clearly, using the definition above as an algorithm would result in a very inefficient implementation since it would require materializing IDB relations twice, i.e., once in each database state. Instead, an *incremental* approach is employed. An update propagation algorithm can be used for this purpose and is the subject of the next section.

3 Update Propagation

This section describes the *EPF* algorithm. For clarity, we initially present a version that computes updates to IDB relations defined in terms of safe, recursive Datalog without negation. We then describe the integration of a memoing feature and the extensions that support stratified negation and modifications to the view definition.

The dependency graph [ULL88] $\mathcal{DG}$ for a Datalog program $\mathcal{D}$ can be used to determine the IDB relations defined by $\mathcal{D}$ that may have been updated as a result of the updates to the EDB relations. An IDB relation I may have been updated as a result of updates to the EDB relations if the predicate defining I, namely i, depends, directly or indirectly, on one or more of the set of updated EDB relations $\mathcal{E}_u$.

Definition. Let e_u represent the predicate defining an arbitrary EDB relation E_u where $E_u \in \mathcal{E}_u$. Let $\Rightarrow$ be a path in $\mathcal{DG}$. An IDB relation I, defined by the predicate i, is termed a *candidate* for update if:

$$e_u \Rightarrow i \in \mathcal{DG}$$

An IDB relation is said to be *unaffected* if it is not a candidate. □

A subset (not necessarily proper) of the rules that define a candidate relation can contribute changes after updates to the base relations are introduced.

Definition. A rule defining a candidate predicate that contains one or more literals in the rule body corresponding to either a candidate predicate or an updated EDB relation is termed a *candidate rule*. A rule is termed *unaffected* if it is not a candidate.

The *EPF* algorithm computes the updates for all candidate relations. This is accomplished by iterating a *propagation* phase followed by a *filtration* phase. These phases are discussed below.

3.1 Propagation and Filtration

During the propagation phase, candidate rules are evaluated when the relations that correspond to subgoals are updated. The evaluation is constrained using bindings taken from these updates. The result of the evaluation is a set of tuples representing possible updates to the candidate relation.

Definition. The set of tuples generated for an IDB relation as a result of a propagation phase is termed an *approximation*. Each tuple in the approximation is termed a *potential IDB update*.

To obtain an approximation from a candidate rule, a query consisting of the literals appearing in the rule body is invoked over either DB_{Old} or DB_{New}. Consider a rule r defining an IDB relation P where both additions and removals have been identified for a relation L corresponding to a literal l appearing in the body of r. To propagate the additions to L to the IDB relation P, the rule body is evaluated using DB_{New}. The evaluation is constrained using bindings from ΔL_{add}. The result of this evaluation is a relation whose schema contains all of the variables that occur in the rule body. The relation is projected onto the set of attributes corresponding to the set of variables that appear in the head literal. A similar procedure is performed to process the removals, however, the rule body is evaluated using DB_{Old} and the evaluation is constrained using bindings from ΔL_{rem}. The result of the projection is the approximation for P.

Note that for both additions and removals the subgoal representing the literal l can be removed from the query for efficiency, since the updates to L used to constrain the query bind all variables appearing in subgoal l. This forms a query that tests a tuple that has already been determined to be an actual IDB update and, therefore, represents redundant computation.

If the rule body contains several literals, each representing either a candidate relation or an updated base relation, then a separate query is issued for each literal. In addition, a separate query is issued for the additions to, and the removals from, each relation. In the worst case situation, where a rule defining a predicate p has k subgoals each corresponding to a relation where both addition and removal updates have been identified, $2k$ queries would be issued during the propagation phase to obtain all potential updates for p. In our implementation, a multiple query optimizer identifies common subexpressions in the queries, which all involve the same set of predicates, thereby significantly reducing the actual computation performed.

The propagation phase propagates the changes to the extensional relations up through the rules and identifies potential changes to the intensional relations. Potential changes are filtered to identify actual changes. For example, a potential addition represents a derivation of a tuple t. If t is provable in the database state before the updates, then the potential addition is filtered and is *not* reflected as an actual change to the database. Similarly, a potential removal represents the deletion of a derivation for a tuple t and if t is still provable in the database state after the updates, then the filter phase does not identify the potential removal as an actual removal.

Thus, the filtration phase of *PF* refines the approximation of potential updates to IDB relations identified during the propagation phase. Potential IDB updates that cannot be proven are removed from the approximation. Each potential addition is posed as a query to the database using DB_{Old}. Each potential removal is posed as a query to the database using DB_{New}. Tuples returned as a result of the query do not represent a change in the database state so they are deleted from the approximation.

Definition. A potential IDB removal is termed *disqualified* if it is provable in DB_{New}. A potential IDB addition is disqualified if it is provable in DB_{Old}. A potential IDB update that is not disqualified is termed an *actual IDB update.*

Actual IDB updates are saved in global Δsets and are available for use in subsequent invocations of the algorithm, including recursive ones. The Δsets facilitate duplicate elimination. They are also used to reduce the size of subsequent approximations, as described in the next section.

3.2 The EPF Algorithm

The *EPF* basic algorithm consists of three procedures, which are given in Figure 1. Let $\mathcal{E}_u$ represent the set of updated base relations. Let $\Delta\mathcal{E}_u$ be the set of Δsets corresponding to $\mathcal{E}_u$. If E is an updated base relation then $\Delta E \in \Delta\mathcal{E}_u$. To process the updates, the system calls procedure *process_updates* to initiate propagation of the updates to the stored relations. This procedure calls procedure *select_propagation_rules* to identify the rules that are to be used for propagation. For clarity of the algorithm, a *for* loop is used to iterate over all of the candidate rules so that each can be considered for use for a propagation phase. A more efficient method, which is employed in our prototype, uses the dependency graph to identify the applicable rules.

After the rules are identified, the procedure *propagate_filter* is called to perform propagation and filtration. The function *query_appr(Query, Updates, State)* is called by the procedure to compute the approximation. It issues a query consisting of a conjunction of literals over the database state indicated by *State.* Bindings ex-

```
Procedure process_updates
begin
    For each $\Delta E \in \Delta\mathcal{E}_u$ do begin
        $e$ = pred-arity($E$);
        if $\Delta E_{rem} \neq \emptyset$ then select_propagation_rules($e$, $\Delta E_{rem}$);
        if $\Delta E_{add} \neq \emptyset$ then select_propagation_rules($e$, $\Delta E_{add}$);
    end;
end;

Procedure select_propagation_rules($q$, $\mathcal{U}_{Type}$)
begin
    For each candidate rule: $p \leftarrow \mathcal{Q}$ do
        For each $q_i \in \mathcal{Q}$ | pred-arity($q_i$) = pred-arity($q$) do
            propagate_filter($p \leftarrow \{\mathcal{Q} - q_i\}$, $\mathcal{U}_{Type}$);
end;

Procedure propagate_filter( $p \leftarrow \mathcal{Q}'$ , $\mathcal{U}_{Type}$)
begin
1)  if Type = add then begin
2)      Prop-State = new;   Filter-State = old end
3)  else begin /* Type = rem */
4)      Prop-State = old;   Filter-State = new end;
5)  $\mathcal{A} = \pi_{atts(p)}$ (query_appr($\mathcal{Q}'$, $\mathcal{U}_{Type}$, Prop-State));
6)  $\mathcal{A}_{Red} = \mathcal{A} - \Delta P_{Type}$
7)  $\mathcal{D}$ = query_disq($p$, $\mathcal{A}_{Red}$, Filter-State);
8)  $\mathcal{U}^P_{Type} = \mathcal{A}_{Red} - \mathcal{D}$;
9)  if $\mathcal{U}^P_{Type} \neq \emptyset$ then begin
10)     $\Delta P_{Type} = \Delta P_{Type} \cup \mathcal{U}^P_{Type}$;
11)     select_propagation_rules($p$, $\mathcal{U}^P_{Type}$);
    end;
end;
```

Figure 1: Basic EPF Algorithm

tracted from *Updates* are used to constrain the evaluation of *Query*. The function *query_disq(Pred,Approx,State)* is called by procedure *propagate_filter* to filter the approximation. It poses a set of tuples *Approx* each with predicate symbol *Pred*, as a query in the database state represented by *State*. Each type of query can be evaluated using any strategy that is sound, complete and terminates for recursive Datalog programs.

Lines 1-4 of the *propagate_filter* procedure select the correct database state for propagation and filtration based on the type of update that is being propagated. The propagation phase is implemented by line 5. Updates that have already been discovered are removed from the approximation at line 6. The filtration phase is implemented at line 7. Newly discovered updates are identified at line 8. The Δset is updated at line 10. If any actual IDB updates are identified that were not previously known, then the algorithm is recursively called to propagate them at line 11.

Note that the EPF algorithm computes Δsets, Not DB_{New}. State DB_{New} becomes available immediately when the updates to the EDB relations are submitted. Newly inserted tuple are ignored when querying the database when access to DB_{Old} is required. Tuples tagged for removal are ignored when querying the database when access to DB_{New} is required. The EPF algorithm issues queries to each state.

The *EPF* algorithm never accesses the materialized view to compute the Δsets. Therefore, it can be used to monitor changes to derived relations *without* having to materialize the derived relation. This capability is useful in an active database where a user may define triggers to be activated by changes to derived data. Changes to the derived data can be computed incrementally without having to incur the costs of either materializing or storing the derived relation.

It is possible that a potential IDB update p may appear in more than one approximation as a result of multiple recursive invocations of the *EPF* algorithm. Filtration need only be performed once to determine if p is disqualified, or alternatively, if p represents an actual IDB update. To avoid redundant computation, *EPF* performs a check to ensure that p is only filtered once. The check that avoids filtering IDB updates that already appear in a Δset occurs at line 6 in procedure *propagate_filter*.

Let p be a potential IDB update of type *Type*. Let $\mathcal{A}_{Type}$ be an approximation containing tuples of type *Type*. If $p \in \Delta\text{P}_{Type}$ then filtering p more than once represents redundant computation since p will be discarded later as a duplicate. These concepts are formalized below.

Proposition 1. *If there exists p such that $p \in \mathcal{A}_{rem}$ and $p \in \Delta\text{P}_{rem}$ then p is a duplicate. Alternatively, if there exists p such that $p \in \mathcal{A}_{add}$ and $p \in \Delta\text{P}_{add}$ then p is a duplicate.*

Proof (sketch). *As can be observed from the algorithm, no tuple can be a member of ΔP_{rem} or ΔP_{add} without having appeared in an approximation and not having been disqualified.* □

The check that avoids filtering disqualified IDB updates more than once is described in a later section.

Example

The following example illustrates the propagation and filtration performed by the basic unoptimized EPF algorithm on a simple example. The program and base data are given in Figure 2. The EDB relation *edge*, which is derived from an example appearing in [GUP93a], is represented as a graph. The only update is the removal of *edge(b,d)*.

The *EPF* algorithm will invoke procedure *select_propagation_rules* twice (once from procedure *process_updates* and once from procedure *propagate_filter*. This will result in procedure *propagate_filter* being called three times. Each invocation of procedure *propagate_filter* will be represented (in order of call) by an entry in the table in Figure 3. Each entry indicates the rule used for propagation (Rule), the updates being processed using the rule body ($\mathcal{U}$), the approximation computed by propagation ($\mathcal{A}$), the disqualified tuples identified by filtration ($\mathcal{D}$) and the actual IDB updates obtained from the approximation ($\mathcal{U}^P$).

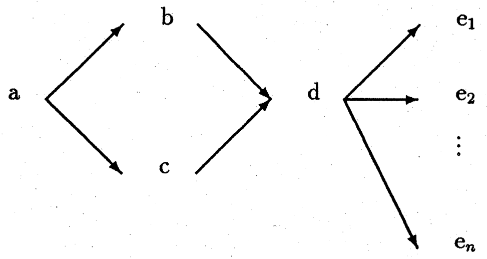
The Δset computed by the algorithm for relation TC is the set of actual IDB updates identified by each invocation. Note that at no time are tuples of the form: $tc(a,e_i)$ ever considered potentially deleted. Also note that if the graph of the transitive closure relation was comprised of several connected components, the algorithm would only examine tuples representing edges in the single connected component where the update occurred.

3.3 Increasing Performance using Memoing

The *EPF* algorithm may issue many queries during the propagation and filtration phases to compute the Δsets. The worst case involves a rule that has n subgoals where each subgoal corresponds to a relation that has been updated as a result of both additions and removals. In this situation, $2n$ queries would be issued, where n queries would be issued in the *new* database state and n queries would be issued in the *old* database state. Each of these queries will involve essentially the same set of predicates. A call made to the same predicate by different queries represents a common subquery if the arguments of one call will unify with the arguments of another.

The invocation of multiple related queries motivates the development of a multiple query optimization (MQO) algorithm to increase the efficiency of propagation and filtration. As described in [DIE87], memoing inherently implements the MQO task of *common subexpression identification* [CHA86, PAR88]. Each occurrence of a literal defining an IDB predicate represents the subexpressions defined by the conjunction of literals in the bodies of the rules defining the predicate.

Our prototype employs a top-down recursive query evaluation strategy known as *EQ*¬*[HAR92c]. This strategy, which is an enhanced version of the *ET** strat-



tc(X,Y) ← edge(X,Y).
tc(X,Y) ← tc(X,W), edge(W,Y).

Figure 2: Sample Graph

Rule:	*tc(X,Y) ← edge(X,Y).*
$\mathcal{U}$:	{*edge(b,d)*}
$\mathcal{A}$:	{*tc(b,d)*}
$\mathcal{D}$:	{ ∅ }
$\mathcal{U}^P$:	{*tc(b,d)*}

Rule:	*tc(X,Y) ← tc(X,W), edge(W,Y).*
$\mathcal{U}$:	{*tc(b,d)*}
$\mathcal{A}$:	{*tc(b,e$_1$), tc(b,e$_2$), ..., tc(b,e$_n$)*}
$\mathcal{D}$:	{ ∅ }
$\mathcal{U}^P$:	{*tc(b,e$_1$), tc(b,e$_2$), ..., tc(b,e$_n$)*}

Rule:	*tc(X,Y) ← tc(X,W), edge(W,Y).*
$\mathcal{U}$:	{*edge(b,d)*}
$\mathcal{A}$:	{*tc(a,d)*}
$\mathcal{D}$:	{*tc(a,d)* }
$\mathcal{U}^P$:	{∅}

Figure 3: Trace of Basic *EPF* Algorithm

egy [DIE87], utilizes memoing to insure completeness and to facilitate the implementation of MQO. The $EQ^{*}\neg$ algorithm detects *completed* calls, which have an extension that is complete. Completed calls that are detected by the algorithm are not recomputed. Instead, the answers for the call, which are retained in the extension table, are returned to the caller. If a call is made to a predicate that is not subsumed by an earlier call, the call is tagged complete after evaluation and the results are made available to subsequent callers.

The MQO optimization benefits both the propagation and filtration phases of the *PF* algorithm. For example, memoed tuples obtained from DB_{New}, when computing potential additions during a propagation phase, are used to reduce the effort required to filter potential removals in a subsequent filtration phase. Alternatively, memoed tuples obtained from DB_{Old}, when computing potential removals during a propagation phase, are used to reduce the effort required to filter potential additions in a subsequent filtration phase. Tuples stored in a memo table for unaffected predicates, obtained during a propagation or filtration phase, can be utilized regardless of the state in which the query is posed.

Regardless of the type of update, memoing can be used to ensure that disqualified updates are filtered exactly once. All that is required is to check the extension table for p, which is partitioned into those tuples derived from the each state. If p appears in an approximation computed using the *new* state and it also appears in the extension table computed using the *old* state, then p is disqualified. Alternatively, if p appears in an approximation computed using the *old* state and it also appears in the extension table computed using the *new* state, then p is disqualified. This concept is formalized below.

Let ΘP_{State} represent the (possibly empty) extension table for relation P that holds answers computed in state *State*.

Proposition 2. *If there exists p such that $p \in \mathcal{A}_{rem}$ and $p \in \Theta P_{New}$ then p is disqualified. Alternatively, if there exists p such that $p \in \mathcal{A}_{add}$ and $p \in \Theta P_{Old}$ then p is disqualified.*

Proof (sketch). *If a removal appears in the memo table derived using DB_{New} then the removal is provable in that state. This implies that the removal is disqualified. Alternatively, if an addition appears in the memo table derived using DB_{Old} then the addition is provable in that state. This implies that the addition is disqualified.* □

Proposition 3. *Algorithm EPF will incur the cost of rederiving a potential IDB update exactly once.*

Proof (sketch). *As implied by proposition 1, if* p *represents an actual IDB update, then EPF will attempt to rederive p during filtration exactly once. As implied by proposition 2, if* p *represents a disqualified update, then EPF will rederive p exactly once. Since all tuples that appear in an approximation must be either actual or disqualified IDB updates, no more than one attempt will be made to rederive any tuple appearing in an approximation.* □

```
Procedure select_propagation_rules($q$, $\mathcal{U}_{Type}$)
begin
    For each candidate rule: $p \leftarrow \mathcal{Q}$ do
        For each $q_i \in \mathcal{Q}$ | pred-arity($q_i$) = pred-arity($q$) do begin
            if negated($q_i$) then
                if Type = rem then Type = add else Type = rem
            propagate_filter($p \leftarrow \{\mathcal{Q}$ - $q_i\}$, $\mathcal{U}_{Type}$);
        end;
end;
```

Figure 4: Modification to support stratified negation

3.4 Negation

An extension to EPF provides support for stratified negation. Assume a rule defining an IDB relation P contains a negated literal $\neg l$ that is the only literal in the rule representing an updated relation R_U. Additions to R_U, labeled as Δl_{add}, can only have the effect of generating potential removals for P. This is because tuples for P formed with bindings that occur in tuples of Δl_{add} are no longer provable with the rule. Therefore, algorithm EPF processes additions to R_U, where l appears in a negated context, as removals from R_U assuming l had appeared in a non-negated context.

Conversely, removals from R_U, labeled as Δl_{rem}, can only have the effect of generating potential additions for P. This is because tuples for P that could be formed with bindings that occur in tuples of Δl_{rem} are now provable with the rule. Algorithm EPF processes removals from R_U where l appears in a negated context as additions to R_U assuming l had appeared in a non-negated context. The modification required to upgrade algorithm EPF to support stratified negation is shown in Figure 4.

3.5 Modification of the View Definition

In this section, we address view maintenance when the definition of the derived relation is modified. When the derived relation is defined using deductive rules, these modifications are in the form of *rule updates*. Rule updates are defined here as either the addition or removal of a rule from the database. This changes the definition of a derived relation. The derived relation can be viewed as being comprised of a n-way union where each rule defines one operand of the union. An update to the definition of a candidate IDB relation can result in the generation of actual IDB updates to the IDB relation that the rule defines.

Consider the case where a rule update is performed on a non-recursive IDB relation I. When a rule r defining I is added to the database, an unconstrained query consisting of the conjunction of literals representing the rule body is issued over DB_{New}. The result is an approximation that is filtered using DB_{Old}. Conversely, the body of a deleted rule is issued over DB_{Old} to obtain the approximation and DB_{New} is used for filtration.

The query issued over either state includes all literals from the rule body and there are no bindings to constrain computation. Actual IDB updates computed for P resulting from a rule update are propagated using the *EPF* algorithm in the same manner as those resulting from base relation updates. The rule update algorithm can be applied to recursive IDB relations. There is a constraint, however, as to the order in which rules are removed. When removing rules defining a recursive predicate, the base rule must be removed last to avoid an ill-defined recursive specification. When adding rules defining a recursive predicate, the order of addition is inconsequential. The complete algorithm for rule updates and proofs can be found in [HAR92c].

4 Optimizations

In this section, optimizations are described that improve the efficiency of the update propagation approach described above. The optimizations are motivated from deductive database query evaluation strategies but require adaptation for use in incremental view maintenance.

Optimal SIP Selection. Both the propagation and filtration phases of the *EPF* algorithm invoke queries to the database. Each query issued during the propagation phase consists of a subset of the literals that appear in a rule body. The *sideways information passing* strategy (SIP)[BEE91] chosen for query evaluation has a direct effect on the efficiency of the *EPF* algorithm. Informally, a sip implements the decision as to how bindings will be utilized during each step of query evaluation. In our prototype, which is implemented in Prolog, the different sip's are implemented by reordering the subgoals appearing in the Prolog queries. To be consistent with the semantics of negation, negated subgoals are ordered such that their evaluation will be delayed until their arguments are fully bound. Evaluable predicates are also delayed until appropriately bound. The optimization is especially beneficial in the case of non-linear recursion to avoid unconstrained queries being issued.

Dependency Graph Analysis. In an active database, a user may define an *event-condition-action (ECA)* rule that is triggered by either a removal from the virtual view, an addition or both. In the event that the user defines the rule to only detect removals, there is no need to propagate tuples that result in additions (and vice-versa) as they represent irrelevant computation.

To avoid this, the program's dependency graph is analyzed to identify paths that the *EPF* algorithm would use to propagate irrelevant tuples. The objective is to inform the *EPF* algorithm to remove the paths from consideration.

Let *Utype* specify the type of update, e.g., removals or additions, that triggers the active rule. Let $\mathcal{DG}$ be the dependency graph for the database. Let $ArcSeq$ be a sequence of arcs that comprise a path $p \Rightarrow g \in \mathcal{DG}$ (expressed as $p \Rightarrow_{ArcSeq} g$). Let *edb-rel* $\in$ EDB. The following rules define the optimization.

Rule 1: If there exists *edb-rel* such that *edb-rel* $\Rightarrow_{ArcSeq}$ *pred* $\in \mathcal{DG}$ and $ArcSeq$ contains an even number of negative arcs then updates of type *Utype* made to *edb-rel* must be propagated to *pred.*

Rule 2: If there exists *edb-rel* such that *edb-rel* $\Rightarrow_{ArcSeq}$ *pred* $\in \mathcal{DG}$ and $ArcSeq$ contains an odd number of negative arcs and *Utype* is *removals* then addition updates made to *edb-rel* must be propagated to *pred.*

Rule 3: If there exists *edb-rel* such that *edb-rel* $\Rightarrow_{ArcSeq}$ *pred* $\in \mathcal{DG}$ and $ArcSeq$ contains an odd number of negative arcs and *Utype* is *additions* then removal updates made to *edb-rel* must be propagated to *pred.*

After the completion of path analysis, the update types, i.e., additions or removals, to each stored relation that cannot result in an update to the view will have been identified. Any update classified as one of these types is ignored and will not be propagated. The optimization can reduce irrelevant computation in instances where negation is present.

Partial Evaluation. Lakhotia and Sterling describe partial evaluation, as it applies to logic programming, as follows. Given a program $\mathcal{P}$ and a goal $\mathcal{G}$, the result of partially evaluating $\mathcal{P}$ with respect to the goal $\mathcal{G}$ is the program $\mathcal{P}'$ such that for any substitution θ, evaluating $\mathcal{G}\theta$ results in the same answers with respect to both $\mathcal{P}$ and $\mathcal{P}'$. The objective of partial evaluation is to produce a $\mathcal{P}'$ on which $\mathcal{G}\theta$ can be evaluated more efficiently than on $\mathcal{P}$ [LAK90]. A theoretical foundation for partial evaluation is given in [LLO91].

Partial evaluation can be used to form optimized deductive rules that increase the efficiency of update propagation when constants appear in the definition of the view. The predicate v, representing the view, serves as the goal $\mathcal{G}$. The definition of v serves as the program $\mathcal{P}$. Actual IDB updates to the relations that define v that will not contribute to updates to v (due to constants in v) are not propagated. This results in a reduction in irrelevant computation. Informally, the partially evaluated view definition v' is created by pushing the constants appearing in the view definition down through the rules that directly or indirectly define the view. The constants bind variables in the rules and are used to form new, restricted versions.

5 Related Work

This section describes related work that focuses on the problem of view maintenance. The most closely related work is the *Delete & Rederive (DRed)* algorithm proposed by Gupta et al.[GUP93]. The *DRed* algorithm is based on similar concepts to the *propagation/filtration (PF)* algorithm[HAR92a], which is the predecessor of the *EPF* algorithm described here.

The *DRed* algorithm performs one propagation phase and one filtration phase per stratum. This behavior allows the creation and propagation of disqualified IDB updates between predicates within a stratum. After *DRed* completes this irrelevant propagation, it must then filter, i.e. rederive, each of these erroneously propagated tuples thereby incurring the cost of still more irrelevant computation.

The *PF* algorithm does not propagate disqualified IDB updates like *DRed.* Instead, it detects and removes any irrelevant tuples immediately after they are generated. This pruning eliminates costly erroneous propagation. However, the *PF* algorithm will filter actual IDB updates more than once in certain instances. When this occurs, it becomes unclear whether the *PF* or the *DRed* algorithm will perform the least computation. There are examples showing how the *PF* and *DRed* algorithms can outperform each other by an order of magnitude depending on the view definition and stored data[GUP93a].

The *EPF* algorithm, however, is more efficient (even in its unoptimized form) than both *PF* and *DRed* since *EPF* will never filter either potential or actual IDB updates more than once (as indicated by proposition 3) nor will it propagate disqualified IDB updates. The net result is a significant increase in efficiency.

Tompa and Blakeley[TOM88] and Blakeley et al.[BLA89], describe tests for identifying irrelevant and autonomously computable updates. The tests are only applicable on views defined using SPJ operations. In [BLA86] a differential re-evaluation algorithm was described for recomputing materialized views efficiently. Again, however, only SPJ views are supported. The *EPF* algorithm focuses on the efficient recomputation of the view. The *EPF* algorithm identifies some irrelevant updates using the dependency graph for the Datalog program. More importantly, however, is the *EPF* algorithm can be applied to a larger class of views.

The *EKS-V1* system [KUC91, VIE91] is a "knowledge base management system" developed at ECRC. The system contains an update propagation mechanism that was designed for integrity constraint checking and can be used for materialized view maintenance. In the *EKS-V1* approach, additional rules, which are termed *propagation rules*, are added to the database to direct propagation. These propagation rules result in a significant increase in the total number of rules that must be maintained by the database. Specifically, for each rule with n literals, $2n$ propagation rules are created. Although this approach does not propagate the superfluous tuples (like *DRed*), the propagation rules may perform unnecessary rederivations to identify superfluous tuples. With the *EPF* algorithm, no additional propagation rules are required and through the use of memoing, the *EPF* algorithm does not perform any unnecessary rederivation.

Dong and Topor [DON92] describe an approach for creating incremental queries that is related to the work presented here. Incremental query evaluation creates a non-recursive query to obtain the answer to an initial query assuming that the view is materialized and also that only additions are made to the database. The *EPF* algorithm described here differs in that both additions and removals are supported and that no requirement that the view is materialized is made. However, the queries issued by the *EPF* algorithm may be recursive and therefore may be more expensive.

Other work addressing view maintenance impose restrictions such as supporting non-recursive views only [WOL91], or depends on information such as keys or functional dependencies that may not be available [CER92, URP92]. Note that another aspect of the materialized view maintenance problem involves identifying the updates that must be made to stored relations resulting from updates made directly to the materialized view. This aspect has not been addressed in this paper.

6 Summary and Future Work

This paper described an algorithm, referred to as *Extended Propagation/Filtration (EPF)*, that computes updates to views that are defined using safe, recursive Datalog with stratified negation. The *EPF* algorithm can be employed to implement database systems that intend to support the proposed SQL3 standard. An extension was described that allowed *EPF* to incrementally maintain materialized views when the view definition is modified if the view is defined using the union operator. Optimizations were described that increase the efficiency of the algorithm. Because the algorithm does not require the view to be materialized, the algorithm can also be applied to the problem of complex event detection in an active database. For future work, we intend to develop more optimizations and implement *EPF* in a parallel environment. We believe that an opportunity exists to exploit parallelism since multiple invocations of *EPF* only query the database.

Acknowledgements

We are grateful to Ashish Gupta for the examples that he provided. We are also grateful to the referees whose suggestions resulted in improvements to this paper.

References

[BAN86] Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J., "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Washington, DC, 1986, pp. 1–15.

[BEE91] Beeri, C. and Ramakrishnan, R., "On the Power of Magic", *Journal of Logic Programming*, October ,1991:10, pp. 255–299.

[BLA86] Blakeley, J., Larson, P. and Tompa, F., "Efficiently Updating Materialized Views", Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, 1986, pp. 61–71.

[BLA89] Blakeley, J. A., Coburn, N., and Larson, P., "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", ACM Transactions on Database Systems, September 1989, pp. 369–400.

[CER92] Ceri, S. and Widom, J., "Deriving Production Rules for Incremental Maintenance", Proc. 17th VLDB, 1991.

[CHA86] Chakravarthy, U. S. and Minker, J., " Multiple Query Processing in Deductive Databases using Query Graphs", *Proc. of the 12th Intl. Conf. on Very Large Data Bases*, Kyoto, August 1986.

[CHA89] Chakravarthy, S., "Rule Management and Evaluation: An Active DBMS Perspective", SIGMOD RECORD, Special Issue on Rule Management and Processing in Expert Database Systems,Vol. 18, No. 3, September 1989, pp. 20–28.

[CHA91] Chakravarthy, S. and Garg, S., "Extended Relational Algebra (ERA): for Optimizing Situations in Active Databases", Technical Report UF-CIS TR-91-24, CIS Department, University of Florida, Gainesville, November 1991.

[DAY88] Dayal, U., "Active Database Management Systems", Proc. of the Third Int. Conf. on Data and Knowledge Bases, Jerusalem, June 1988, pp. 150–170.

[DIE87] Dietrich, S. W., "Extension Tables: Memo Relations in Logic Programming", *IEEE Symposium on Logic Programming*, San Francisco, CA, 1987, pp. 264–272.

[DIE92] Dietrich, S., Urban, S., Harrison, J. and Karadimce, A., "A DOOD RANCH at ASU: Integrating Active, Deductive and Object-Oriented Databases", *Special Issue on Active Databases, Data Engineering Bulletin*, Vol. 15, No. 1-4, December 1992, pp. 40–43.

[DON92] Dong, G. and Topor, R., "Incremental Evaluation of Datalog Queries", *Fourth International Conference on Database Theory*, 1992.

[GUP93] Gupta, A., Mumick, I. S. and Subrahmanian, V. S., "Maintaining Views Incrementally", *Proceedings of the 1993 ACM SIGMOD*, Washington, DC, May 1993, pp. 157–166.

[GUP93a] Gupta, A., "Examples comparing PF with DRed", unpublished manuscript.

[HAR92a] Harrison, J. V. and Dietrich, S. W., "Maintaining Materialized Views in Deductive Databases: An Update Propagation Approach", Proceedings of the Deductive Database Workshop held in conjunction with the Joint International Conference and Symposium on Logic Programming, Washington, D.C., November, 1992, pp. 56–65.

[HAR92b] Harrison, J. V. and Dietrich, S. W., "Towards an Incremental Condition Evaluation Strategy for Active Deductive Databases", In Proceedings of Databases '92, Third Australian Database Conference, Melbourne, Australia, February 1992. pp. 81–95.

[HAR92c] Harrison, J. V., "Condition Monitoring in an Active Deductive Database", Ph.D. Dissertation, Arizona State University, August, 1992.

[HAR93] Harrison, J. V., "Monitoring Complex Events defined using Aggregates in an Active Deductive Database", University of Queensland Tech. Rep. 268, May, 1993 (revised).

[KUC91] Kuchenhoff, V.,"On the efficient computation of the difference between consecutive database states", In Proc. of the Second Intl. Conf. on Deductive and Object-Oriented Databases (DOOD), Munich, Germany, December 1991.

[LEF92] Lefebvre, A., "Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases", *Proc. of FGCS'92.*

[LAK90] Lakhotia, A. and Sterling, L., "ProMiX: a Prolog Partial Evaluation System", In *The Practice of Prolog*, Sterling, L. (eds), MIT Press, Cambridge, 1990, pp. 137–179.

[LLO91] Lloyd, J. W. and Shepherdson, J. C., "Partial Evaluation in Logic Programming", *Journal of Logic Programming*, 11:217-242, 1991.

[MUM91] Mumick, I. S., Pirahesh, H. and Ramakrishnan, R., "The Magic of Duplicates and Aggregates", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, 1990, pp. 264–277.

[PAR88] Park, J. and Segev, A., "Using Common Subexpressions to Optimize Multiple Queries", Proc. of Seventh IEEE Conf. on Data Engineering, 1988, pp. 311–319.

[ROS89] Rosenthal, A., Chakravarthy, S., Blaustein, B., and Blakeley, J., "Situation Monitoring for Active Databases", Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, 1989.

[TOM88] Tompa, F. and Blakeley, J., "Maintaining Materialized Views without Accessing Base Data", *Information Systems*, Vol. 13, No. 4, 1988, pp. 393–408.

[ULL88] Ullman, J., Principles of Database and Knowledge-base Systems, Vol. 1, Computer Science Press, Rockville, MD, 1988.

[ULL89] Ullman, J., *Principles of Database and Knowledge-base Systems*, Vol. 2, Computer Science Press, Rockville, MD, 1989.

[URP92] Urpi', T. and Olive, A., "Events and Event rules in Active Databases", *Special Issue on Active Databases, Bulletin of the Technical Committee on Data Engineering*, December, 1992 Vol. 15, No. 1-4.

[VIE91] Vieille, L., Bayer, P. and Kuchenhoff, V., "Integrity Checking and Materialized Views Handling by Update Propagation in the EKS-V1 System", ECRC Technical Report TR-KB-35, ECRC, Munich, Germany, June 1991.

[WOL91] Wolfson, O., Dewan, H. M., Stolfo, S. and Yemini, Y., "Incremental Evaluation of Rules and its Relationship to Parallelism", *Proceedings ACM-SIGMOD 1991 Intl. Conf. on Management of Data*, pp. 78–87.

ADC '94

Proceedings of the
5th Australasian Database Conference

Christchurch, New Zealand 17 – 18 January 1994

Editor

R. Sacks-Davis
Research Director
CITRI: Collaborative Information Technology Research Institute
of RMIT and The University of Melbourne
Australia

Global Publications Services