Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach

John V. Harrison and Suzanne W. Dietrich

Department of Computer Science and Engineering
College of Engineering and Applied Sciences
Arizona State University
Tempe, AZ 85287–5406, U.S.A.
{harrison|dietrich}@enuxha.eas.asu.edu

Abstract

A view is a derived relation that is defined in terms of stored relations and other derived relations. If the derived relation representing a view is retained between references, as opposed to being reconstructed each time it is required, then it is termed *materialized*. Materialized views are used to increase the speed of query processing and to support *procedures* in relational database systems. Updates made to stored relations that participate in the view's definition can invalidate the materialized view. Optimally, the database system should be capable of updating a materialized view to reflect updates to the stored relations thereby avoiding the cost of reconstructing the materialized view.

In a deductive database, the task of maintaining materialized views is challenging because views can be defined using negation and recursion. This paper presents an algorithm for maintaining materialized views in a deductive database consisting of safe, recursive Datalog with stratified negation. Materialized views that cannot be maintained using previously proposed methods, namely those views that are recursively defined or are defined using all of the relational operators, can be maintained using this approach.

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 represented by the view is reconstructed each time it is referenced the view is considered *virtual*. If the derived relation is retained between references then the view 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. Materialized views can also be used to both restructure the internals of a relational database and to extend a relational database to support procedures[TOM88].

Updates made to the stored relations that participate in a view's definition can invalidate the materialized view. One way to resolve the invalidation is to recompute the materialized view after the database is updated, i.e., discard the materialized view and recreated it using the current database state. This approach can be very costly in the presence of complex view definitions and frequent database updates.

To avoid these costs, the materialized view itself can be updated to reflect the changes made to stored relations referenced in 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, i.e., recomputing, 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 materialized 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 recomputed using their *differential re-evaluation* algorithm, which is described in [BLA86]. The views supported by their approach are restricted to select-project-join (SPJ) expressions.

The approach described in this paper extends the work described above by providing support for views that are defined using all of the relational operators, as well as recursion. Specifically, an algorithm is proposed, which is known as *Propagation/Filtration* (PF), that directly updates materialized views defined using safe, recursive Datalog rules with stratified negation. The PF algorithm computes the updates to materialized views, i.e., constructed derived relations, that result from updates made to the extensional database, i.e., the stored relations.

This paper is a companion paper to [HAR91], which applies the PF algorithm to the problem of efficient condition monitoring in an *active* deductive database. 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 of the problem is not addressed in this work.

The remainder of the paper is organized as follows. In the second section, initial concepts and terminology is introduced. The third section describes the PF algorithm and provides an example of its application to the materialized view maintenance problem. Related work is discussed in section four.

2 Basic Concepts

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. Once computed, this difference can be used to directly update the materialized view.

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 U be a set of updates to the EDB. The database state before U is performed is referred to as *old*. The database state after U is performed is referred to as *new*. Let the function *materialize(IDB_Pred,DB_State)* compute an IDB relation defined by the predicate *IDB_Pred* using the EDB indicated by *DB_State*. 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 can be formalized using the definitions below, which assume that the definitions of the IDB relations are not updated.

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. Let IDB(EDB) represent an IDB derived using a specific EDB.

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

□

To perform the view maintenance function, the Δset ΔP is required. The tuples in ΔP indicate how the materialized view should be updated to reflect the updates made to the stored relations. Using the definition above as an algorithm is clearly unsatisfactory since complete recomputation of the view is required. Instead, an *incremental* approach is employed, which is implemented using an update propagation algorithm and is the subject of the next section.

3 Update Propagation

This section describes the PF 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 extension that supports stratified negation.

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

3.1 The Propagation Phase

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 evaluate the candidate rules, the rule bodies are posed as queries to the database. If a rule body contains multiple literals, each representing a relation that has been updated, then a separate query is issued corresponding to each literal. The query associated with a literal l is constrained using the bindings contained within the set of updates for the relation corresponding to l. A separate query is issued for the additions to and the removals from the 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 propagation phases to obtain all potential updates for p. In our implementation of the PF algorithm, a multiple query optimizer identifies common subexpressions in the queries, all of which involve the same set of predicates, thereby significantly reducing the actual computation performed.

Consider a rule r defining a predicate p where both additions and removals exist for a relation corresponding to a literal l in the body of r. To process the additions, the rule body is evaluated using DB_{New} and is constrained using bindings from Δl_{add}. To process the removals, the rule body is evaluated using DB_{Old} and is constrained using bindings from Δl_{rem}. 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. After a propagation phase has been performed for r, a (possibly empty) set of tuples representing potential IDB updates to p will have been computed.

3.2 The Filtration Phase

During the filtration phase of PF, the approximation computed during the propagation phase is refined. Potential IDB updates that cannot be proven are removed from the approximation. The potential additions (removals) are posed as queries to the database using DB_{Old} (DB_{New}). Tuples returned as a result of the query are already provable and therefore do not represent a change in the database state. They are deleted from the approximation.

Definition. A potential IDB removal (addition) computed during the propagation phase in DB_{Old} (DB_{New}) is *disqualified* if it is provable in DB_{New} (DB_{Old}). The potential IDB updates belonging to an approximation that are not disqualified are termed *actual IDB updates*.

Actual IDB updates not previously discovered during an earlier recursive invocation of the algorithm are tabled in a Δset and are then used for a subsequent propagation phase. The Δsets are global so they can be accessed by all recursive invocations.

3.3 Algorithm PF

Let EDB_REL be the set of base relations in the database and let ΔEDB_REL represent the updates to the base relations as presented to the system. To process the updates, the system calls procedure *update*, given in Figure 1, which in turn calls procedure *PF*, which appears in Figure 2.

```
Procedure update
begin
    For each ΔE ∈ Δℰ_u do begin
        p = pred_sym(ΔE);
        Rem_tuples = removals(ΔE);
        Add_tuples = additions(ΔE);
        if Rems ≠ ∅ then PF(p, Rem_tuples, rem)
        if Adds ≠ ∅ then PF(p, Add_tuples, add)
    end;
end {update}
```

Figure 1: Update Procedure

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 *update*, given in Figure 1, which in turn calls procedure *PF*, which appears in Figure 2.

The function *query_appr(Query,Ex_Lit,Updates,State)* is called by procedure *propagate_filter* to compute the approximation. It issues the query *Query*, consisting of a conjunction of literals representing a rule body, in the database state represented by *State*. The literal *Ex_Lit*, which corresponds to an updated relation, is excluded from the query since bindings extracted from *Updates* are used to constrain the evaluation of *Query*. The function *query_disq(Pred,Approximation,State)* is called by procedure *propagate_filter* to filter the approximation. It poses a set of tuples *Approximation* 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.

For the remaining description of the algorithm, rules defining IDB relations will be represented using the form:

$$head_{rule}(\overline{H}) \leftarrow body_{rule}(\overline{B})$$

where $\overline{H}$ represents the set of variables appearing the the head literal of the rule and $\overline{B}$ represents the set of variables appearing in the various literals contained in the rule body. The propagation phase is implemented by line 5. The filtration phase is implemented by lines 6-7. Actual IDB updates that were not previously identified during execution of the algorithm are used to update the Δsets at lines 9-10. Newly discovered actual IDB updates force additional propagation at line 11.

Note that the PF algorithm computes Δsets, not DB_{New}. State DB_{New} becomes available immediately when the updates to the EDB relations are submitted. Newly inserted tuples 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 PF algorithm issues queries to each state. Proofs of soundness and completeness for the PF algorithm appear in [HAR91].

It may appear to the reader that it is feasible to employ an integrity constraint (IC) checking algorithm to identify updates to the IDB relations. The idea would be to pose the view defintion as an IC. An "IC violation", i.e., the detection of a refutation, caused by an update to a base relation would identify an addition to, or alternatively, a removal from, the view. Unfortunately, this approach is suboptimal because of a fundamental characteristics of most IC checking algorithms.

Typically, any update submitted to the system that results in an IC violation is aborted. Therefore, the database is restricted to states where no condition corresponding to an IC is satisfiable. This restriction is used as an underlying assumption in many IC checking algorithms [BRY88, LLO87, SAD88]. However,

```
Procedure PF(Pred,Updates,Type)
begin
    Affected_Rules = {candidate rules | ∃ literal L ∈ body_rule representing Pred}
    For each rule R ∈ Affected_Rules do
        For each occurrence L_i of L ∈ R do
            propagate_filter(R,L_i,Updates, Type)
end;


Procedure propagate_filter(Rule,Ex_Lit,Updates,Type)
begin
1)      if Type = add then begin
2)          State_Appr = new; State_Disq = old end
3)      else begin /* Type = rem /*
4)          State_Appr = old; State_Disq = new end;

5)      Approximation = π_(H̄) (query_appr(body_Rule, Ex_Lit, Updates, State_Appr));
6)      Disqualified_tuples = query_disq(head_Rule, Approximation, State_Disq);
7)      Updates = Approximation − Disqualified_tuples;
8)      if Updates ≠ ∅ then begin
9)          New_Updates = Updates − ΔHeadPred_Type;
10)         ΔHeadPred_Type = ΔHeadPred_Type ∪ New_Updates;
11)         if New_Updates ≠ ∅ then
                PF(pred_sym(head_Rule), New_Updates, Type);
        end;
    end;
```

Figure 2: The *PF* Algorithm

r1: link(X,Y) ← direct_link(X,Y).

r2: link(X,Y) ← link(X,Z), direct_link(Z,Y).

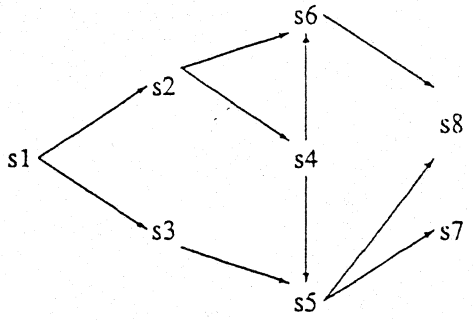


Figure 3: Transitive Closure Example

with a view maintenance mechanism, there is no requirement that the associated "constraint" remain unsatisfiable since updates are not aborted. Therefore, future updates could result in the recomputation of previously discovered refutations. Since a refutation is what triggers a view update in this proposed approach, duplicate updates to the materialized view would be reported.

3.4 Example

Consider the following example, which is based on an electric power distribution network. Let the materialized view *LINK*, defined by the predicate *link*, represent the reachability between source and destination stations on the network. Let the EDB be represented by a single relation *DIRECT_LINK*. Let predicate *link* be comprised of two rules that represent a simple left-recursive transitive closure. The program and underlying graph are shown in Figure 3 where the power stations are represented by the abbreviations s1,s2,...,s7. The PF algorithm will compute the changes to relation *LINK* when relation *DIRECT_LINK* is updated.

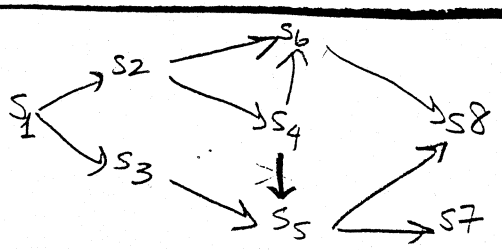
Assume relation *DIRECT_LINK* is updated by removing *direct_link(s4,s5)*, which is represented by the Δset Δ*DIRECT_LINK*. To determine the changes to IDB relations that are defined in terms of relation *DIRECT_LINK*, which in this case is simply the relation *LINK*, the routine *update* makes the following call to procedure PF:

PF(*direct_link*, {(s4,s5)}, *rem*)

A trace of the execution of PF is given in Figure 4. The subscript to each call to PF shown in the figure represents the recursive invocations and the rule being considered. The result of execution is the union of all *link* tuples formed with bindings indicated as *new*: {(s4,s5), (s4,s7), (s2,s5), (s2,s7)}. These tuples correspond to network links lost as a result of the update and would therefore be deleted from the materialized view.

3.5 Negation

An extension to PF provides support for stratified negation. Assume a rule defining an IDB relation P contains a negated literal $\neg l$ that represents 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 PF 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.



link(X,Y) ← direct_link(X,Y).
link(X,Y) ← link(X,Z), direct_link(Z,Y).

$PF_{1,r1}$ Lit: *direct_link*, Removals: {(s4,s5)},
Head: *link*, Approximation: {(s4,s5)}, New: {(s4,s5)}

$PF_{2,r2}$ Lit: *link*, Removals: {(s4,s5)},
Head: *link*, Approximation: {(s4,s7),(s4,s8)}, New: {(s4,s7)}

$PF_{3,r2}$ Lit: *link*, Removals: {(s4,s7)},
Head: *link*, Approximation: {∅}, New: {∅}

$\Delta link_{rem}$ (s4,s5) (s4,s7) (s2,s5) (s2,s7)

$PF_{1,r2}$ Lit: *direct_link*, Removals: {(s4,s5)},
Head: *link*, Approximation: {(s1,s5),(s2,s5)}, New: {(s2,s5)}

$PF_{4,r2}$ Lit: *link*, Removals: {(s2,s5)},
Head: *link*, Approximation: {(s2,s7),(s2,s8)}, New: {(s2,s7)}

$PF_{5,r2}$ Lit: *link*, Removals: {(s2,s7)},
Head: *link*, Approximation: {∅}, New: {∅}

Figure 4: Trace of PF

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 PF 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 PF to support stratified negation is shown in Figure 5.

4 Related Work

This section describes related work that focuses on the problem of maintaining materialized views. 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 an SPJ expression. In [BLA86], a differential re-evaluation algorithm was described for incrementally updating materialized views. Again, however, only views defined using an SPJ expression are supported. The PF algorithm can be applied to a more general class of views. In addition, the PF algorithm identifies a subset of the irrelevant updates using the dependency graph for the Datalog program.

The *EKS-V1* system [KUC91, VIE91] is a "knowledge base management system" under development at ECRC. The system contains an update propagation mechanism that was designed for integrity constraint checking and can also be used for materialized view maintenance. The *EKS-V1* update propagation mechanism, which is described in both [VIE91] and [KUC91], was developed independently, and in parallel, with the work presented in this paper. The mechanism is based on concepts similar to those introduced in the description of the *PF* algorithm however there are some fundamental differences.

In the *EKS-V1* approach, additional rules, which are termed *propagation rules*, are added to the database to direct propagation. The *EKS-V1* propagation rules result in a significant increase in the total number of rules that must be maintained by the database. Specifically, for each candidate IDB rule with

```
Procedure PF¬(Pred,Updates,Type)
begin
    Affected_Rules = {candidate rules | ∃ literal L ∈ body_rule representing Pred}
    For each rule R ∈ Affected_Rules do
        For each occurrence L_i of L ∈ R do begin
            if positive(L_i) then
                propagate_filter(R, L_i, Updates, Type)
            else {negative(L_i)}
                if Type = add then propagate_filter(R, L_i, Updates, rem)
                else {Type = rem} propagate_filter(R, L_i, Updates, add)
        end;
end;
```

Figure 5: Modification to allow stratified negation

n literals, $2n$ propagation rules are created. With the PF algorithm, no additional rules need be added to the database.

5 Summary and Future Work

This paper proposes an algorithm to support the maintenance of materialized views in a deductive database that is defined using safe, recursive Datalog with stratified negation. Unlike previous approaches, the PF algorithm can maintain relational views defined using union, difference (stratified negation), and recursion. For future work, we plan to extend our optimized system prototype to support aggregates. Since the PF algorithm only queries the database, there also appears to be an opportunity to exploit parallelism to increase performance.

References

[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.

[BRY88] Bry, F., Decker, H. and Manthey, R., "A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases", in Proceedings of the Intl. Conf. on Extending Database Technology, Venice, 1988, pp. 488–505.

[HAR91] Harrison, J. V. and Dietrich, S. W., "Condition Monitoring in an Active Deductive Database", Technical Report TR-91-022, Department of Computer Science, Arizona State University (submitted for journal publication).

[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.

[LLO87] Lloyd, J. W., Sonenberg, E. A., and Topor, R. W., "Integrity Constraint Checking in Stratified Databases", J. Logic Programming, 4:331–343, 1987.

[SAD88] Sadri, F. and Kowalski, R., "Theorem-Proving Approach to Database Integrity", Appears in: Foundations of Deductive Databases and Logic Programming (ed. Jack Minker), Morgan Kaufmann Pub., Los Altos, CA,1988, pp. 313–362.

[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.

[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.

Proceedings of the

Workshop on Deductive Databases

held in conjunction with the

Joint International Conference and Symposium on Logic Programming

Washington, D.C., USA

Saturday, 14th November, 1992

Kotagiri Ramamohanarao, James Harland, Ghozhu Dong (editors)