Encyclopedia of Database Systems, Editors-in-chief: Özsu, M. Tamer; Liu, Ling, Springer, 2009.

MAINTENANCE OF RECURSIVE VIEWS

Suzanne W. Dietrich

Arizona State University
http://www.public.asu.edu/~dietrich

SYNONYMS

incremental maintenance of recursive views; recursive view maintenance

DEFINITION

A view is a derived or virtual table that is typically defined by a query, providing an abstraction or an alternate perspective of the data that allows for more intuitive query specifications using these views. Each reference to the view name results in the retrieval of the view definition and the recomputation of the view to answer the query in which the view was referenced. When views are materialized, the tuples of the computed view are stored in the database with appropriate index structures so that subsequent access to the view can efficiently retrieve tuples to avoid the cost of recomputing the entire view on subsequent references to the view. However, the materialized view must be updated if any relation that it depends on has changed. Rather than recomputing the entire view on a change, an incremental view maintenance algorithm uses the change to incrementally compute updates to the materialized view in response to that change. A recursive view is a virtual table definition that depends on itself. A canonical example of a recursive view is the transitive closure of a relationship stored in the database that can be modeled as directed edges in a graph. The transitive closure essentially determines the reachability relationship between the nodes in the graph. Typical examples of transitive closure include common hierarchies such as employee-supervisor, bill-of-materials (parts-subparts), ancestor, and course prerequisites. The incremental view maintenance algorithms for the maintenance of recursive views have additional challenges posed by the recursive nature of the view definition.

HISTORICAL BACKGROUND

A view definition relates a view name to a query defined in the query language of the database. Initially, incremental view maintenance algorithms were explored in the context of non-recursive view definitions involving select, project, and join query expressions, known as SPJ expressions in the literature. The power of recursive views was first introduced in the *Datalog* query language, which is a declarative logic programming language established as the database query language for deductive databases in the 1980s. Deductive databases assume the theoretical foundations of relational data but use Datalog as the query language. Since its relational foundations assume first normal form, Datalog looks like a subset of the Prolog programming language without function symbols. However, Datalog does not assume Prolog's top-down left-to-right programming language evaluation strategy. The evaluation of Datalog needed to be founded on the fundamentals of database query optimization. In a database system, a user need only specify a correct declarative query, and it is the responsibility of the database system to efficiently execute that specification. The evaluation of Datalog was further complicated by the fact that Datalog allows for relational views that include union and recursion in the presence of negation. Therefore, the view definitions in Datalog were more expressive than the traditional select-project-join views available in relational databases at that time. Therefore, the incremental view maintenance algorithms for recursive views in the early 1990s are typically formulated in the context of the evaluation of Datalog. The power to define a recursive union in SQL was added in the SQL:1999 standard.

Historically, it is important to note that the incremental maintenance of recursive views is related to the areas of integrity constraint checking and condition monitoring in active databases. These three areas were being explored in the research literature at about the same time. In integrity constraint checking, the database is assumed to be in a consistent state and when a change occurs in the database, it needs to incrementally determine whether the database is still in a consistent state. In active databases, the database is responsible for actively checking whether a condition that it is responsible for monitoring is now satisfied by incrementally evaluating condition specifications affected by changes to the database. Although closely related, there are differences in the underlying assumptions for these problems.

FOUNDATIONS

Recursive View Definition

A canonical example of a recursive view definition is the reachability of nodes in a directed graph. In Datalog, the reach view consists of two rules. The first non-recursive rule serves as the base or seed case, and indicates that if the stored or base table edge defines a directed edge from the source node to the destination node, then the destination can be reached from the source. The second rule is recursive. If the source node can reach some intermediate node and there is an edge from that intermediate node to a destination node, then the source can reach the destination.

```
    reach(Source, Destination) :- edge(Source, Destination).
    reach(Source, Destination) :- reach(Source, Intermediate), edge(Intermediate, Destination).
```

Intuitively, one can think of the recursive rule as an unfolding of the joins required to compute the reachability of paths of length two, then paths of length three, and so on until the data of the underlying graph is exhausted.

In SQL, this recursive view is defined with the following recursive query expression:

```
with recursive reach(source, destination) as
    (select E.source, E.destination
     from edge E)
    union
    (select S.source, D.destination
     from reach S, edge D
     where S.destination = D.source)
```

SQL limits recursive queries to linear recursions, which means that there is at most one direct invocation of a recursive item. The specification of the reach view above is an example of a linear recursion. There is another linear recursive specification of reach where the direct recursive call appears on the right side of the join versus the left side of the join:

```
    reach(Source, Destination) :- edge(Source, Intermediate), reach(Intermediate, Destination).
```

However, there is a logically equivalent specification of reach that is non-linear:

```
    reach(Source, Destination) :- reach(Source, Intermediate), reach(Intermediate, Destination).
```

The goal of Datalog evaluation is to allow the user to specify the recursive view declaratively in a logically correct way, and it is the system's responsibility to optimize the evaluation of the query.

SQL also restricts recursions to those defined in deductive databases as stratified *Datalog with negation*. Without negation, a recursive Datalog program has a unique solution that corresponds to the theoretical fixpoint semantics or meaning of the logical specification. In the computation of the reach view, each unfolding of the recursion joins the current instance of the recursive view with the edge relation until no new tuples can be added. The view instance has reached a fixed point and will not change. When negation is introduced, the interaction of recursion and negation must be considered. The concept of stratified negation means that there can be no negation through a recursive computation, i.e., a view cannot be defined in terms of its own negation. Recursive views can contain negation but the negation must be in the context of relations that are either stored or completely computed before the application of the negation. This imposed level of evaluation with respect to negation and recursion are called strata. For stratified Datalog with negation, there also exists a theoretical fixpoint that represents the intuitive meaning of the program.

Consider an example of a view defining a peer as two employees that are not related in the employee-supervisor hierarchy:

```
    peer(A, B) :- employee(A, … ), employee(B, …), not (supervisor(A,B)), not(supervisor(B,A)).
    supervisor(Emp, Sup) :- immediateSupervisor(Emp, Sup).
    supervisor(Emp, Sup) :- supervisor(Emp, S), immediateSupervisor(S, Sup).
```

Since peer depends on having supervisor materialized for the negation, peer is in a higher stratum than supervisor. Therefore, the strata provide the levels in which the database system needs to compute views to answer a query.

Evaluation of Recursive Queries

Initial research in the area emphasized the efficient and complete evaluation of recursive queries. The intuitive evaluation of the recursive view that unions the join of the current view instance with the base data at each unfolding is known as a naïve bottom-up algorithm. In a bottom-up approach to evaluating a rule, the known collection of facts is used to satisfy the subgoals on the right-hand side of the rule, generating new facts for the relation on the left-hand side of the rule. To improve the efficiency of the naïve algorithm, a semi-naïve approach can be taken that only uses the new tuples for the recursive view from the last join to use in the join at the next iteration. A disadvantage of this bottom-up approach for evaluating a query is that the entire view is computed even when a query may be asking for a small subset of the data. This eager approach is not an issue in the context of materializing an entire view.

Another recursive query evaluation approach considered a top-down strategy as in Prolog's evaluation. In a top-down approach to evaluation, the evaluation starts with the query and works toward the collection of facts in the database. In the context of the reach recursive view, the reach query is unified with the left-hand side of the non-recursive rule and rewritten as a query involving edge. The edge facts are then matched to provide answers. The second recursive rule is then used to rewrite the reach query with the query consisting of the goals on the right-hand side of the rule. This evaluation process continues, satisfying the goals with facts or rewriting the goals using the rules. The unification of a goal with the left-hand side of a rule naturally filters the evaluation by binding variables in the rule to constants that appear in the query. However, the evaluation of a left-recursive query using Prolog's evaluation strategy enters an infinite loop on cyclic data by attempting to prove the same query over and over again. A logic programmer would not write a logic program that enters an infinite loop but the deductive database community was interested in the evaluation of truly declarative query specifications.

The resulting evaluation approaches combine the best of top-down filtering with bottom-up materialization. The magic sets technique added top-down filtering by cleverly rewriting the original rules so that a bottom-up evaluation would take advantage of constants appearing in the query [1]. Memoing was added to a top-down evaluation strategy to achieve the duplicate elimination feature that is inherent in a bottom-up evaluation of sets of tuples [3]. This duplicate elimination feature avoids the infinite loops on cyclic data. Top-down memoing is complete for subsets of Datalog on certain types of queries [4]. For stratified Datalog with negation, top-down memoing still requires iteration to guarantee complete evaluation. Further research explored additional optimizations as well as implementations of deductive database systems [12] and led to research in active databases and materialized view maintenance.

Incremental Evaluation of Recursive Views

A view maintenance algorithm uses the change to incrementally determine updates to the view. Consider a change in the underlying graph for the transitive closure example. If a new edge is inserted, this edge may result in a change to the materialized reach view by adding a connection between two nodes that did not exist before. However, another possibility is that the new edge added another path between two nodes that were already in the materialized view. A similar situation applies on the removal of an edge. The deletion could result in a change in the reachability between nodes or it could result in the removal of a path but the nodes are still connected via another route. In addition, in the general case, a view may depend on many relations including other (recursive) views in the presence of negation. Therefore, the approaches for the incremental maintenance of recursive views typically involve a propagation or derivation phase that determines an approximation or overestimate of the changes, and a filtering or rederivation phase that checks whether the potential change represents a change to the view. There are differences in the underlying details of how these phases are performed.

The two incremental view maintenance algorithms that will be presented by example are the DRed algorithm [6] and the PF Algorithm [8]. Both the DRed and PF algorithms handle recursive stratified Datalog programs with negation. There are other algorithms developed for special cases of Datalog programs and queries, such as the counting technique for nonrecursive programs, but this exposition will explore these more general approaches for incremental view maintenance. Historically, the PF algorithm was developed in the context of top-down memoing whereas DRed assumes a bottom-up semi-naïve evaluation. To assist with the comparison of the approaches, the notation introduced for the DRed algorithm [6] will be used to present both algorithms in the context of the transitive closure motivational example.

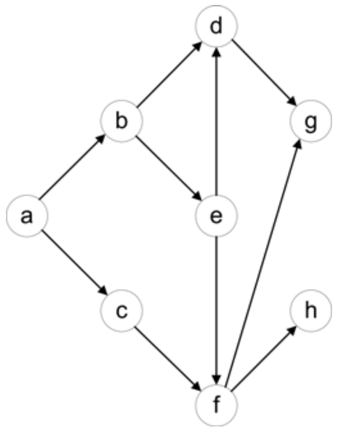

Figure 1. Sample Graph

Figure 1 provides a graphical representation of an edge relation. Assume that the view for reach is materialized, and the edge (e,f) is deleted from the graph. The potential deletions or overestimates for reach, denoted by δ^-(reach), are computed by creating Δ^- rules for each rule computing reach. Each reach rule has k Δ^- rules where k corresponds to the number of subgoals in the body of the rule. The ith Δ^- rule uses the current estimate of deleted tuples (δ^-) for the ith subgoal. For the nonrecursive rule, there is only one subgoal. Therefore, there is only one Δ^- rule indicating that potential edge deletions generate potential deletions to the reach view.

Δ^-(r1): δ^-(reach(S, D)) :- δ^-(edge(S, D)).

Since the recursive rule has two subgoals, there are two Δ^- rules:

Δ^-(r21): δ^-(reach(S, D)) :- δ^-(reach(S, I)), edge(I, D).

Δ^-(r22): δ^-(reach(S, D)) :- reach(S, I), δ^-(edge(I, D)).

Potential deletions to the reach view as well as the edge relation can generate potential deletions to the view.

These potential deletions need to be filtered by determining whether there exist alternative derivations or paths between the nodes computed in the potential deletion. There is a Δ^r rule defined for each reach rule that determines the rederivation of the potential deletions, which is denoted by δ^+(reach):

Δ^r(r1): δ^+(reach(S, D)) :- δ^-(reach(S, D)), edge^v(S, D).

Δ^r(r2): δ^+(reach(S, D)) :- δ^-(reach(S, D)), reach^v(S, I), edge^v(I, D).

The superscript v on the subgoals in the rule indicates the use of the current instance of the relation corresponding to the subgoal. If the potential deletion is still reachable in the new database instance, then there exists another route between the source and destination, and it should not be removed from the materialized view. The actual removals to reach, indicated by Δ^-(reach), is the set of potential deletions minus the set of alternative derivations:

Δ^-(reach) = δ^-(reach) - δ^+(reach)

Table 1 illustrates the evaluation of the DRed algorithm for incrementally maintaining the reach view on the deletion of edge(e,f) from Figure 1. The DRed algorithm uses a bottom-up evaluation of the given rules, starting with the deletion δ^-(edge(e, f)). In the first step, the Δ^- rules compute the overestimate of the deletions to reach. The result of the Δ^- rules are shown in the right column, which indicates the potential deletions to reach as δ^-(reach). The second step uses the Δ^r rules to filter the potential deletions. The right column illustrates the source destination pairs that are still reachable after the deletion of edge(e,f) as δ^+(reach). The tuples that must be removed from the materialized view are indicated by Δ^-(reach): {(e, f) (e,h) (b,f) (b,h)}.

Table 1. DRed Algorithm on deletion of edge(e,f) on materialized reach view

DRed Algorithm		
Step 1	*Compute Overestimate of Potential Deletions*	**δ^-(reach)**
	Δ^-(r1): δ^-(reach(S, D)) :- δ^-(edge(S, D)).	(e,f)
	Δ^-(r21): δ^-(reach(S, D)) :- δ^-(reach(S, I)), edge(I, D).	(e,g) (e,h)
	Δ^-(r22): δ^-(reach(S, D)) :- reach(S, I), δ^-(edge(I, D)).	(a,f) (b,f)
	Repeat until no change: No new tuples for Δ^-(r1) and Δ^-(r22)	
	Δ^-(r21): δ^-(reach(S, D)) :- δ^-(reach(S, I)), edge(I, D).	(a,g) (a,h) (b,g) (b,h)
	Last iteration does not generate any new tuples	
Step 2	*Find alternative derivations to remove potential deletions*	**δ^+(reach)**
	Δ^r(r1): δ^+(reach(S, D)) :- δ^-(reach(S, D)), edge(S, D).	
	Δ^r(r2): δ^+(reach(S, D)) :- δ^-(reach(S, D)), reach^v(S, I), edge^v(I, D).	(e,g) (a,f) (a,g) (a,h) (b,g)
Step 3	*Compute actual changes to reach*	**Δ^-(reach)**
	Δ^-(reach) = δ^-(reach) - δ^+(reach)	(e, f) (e,h) (b,f) (b,h)

The PF (Propagate Filter) algorithm on the same example is shown in Table 2. PF starts by propagating the edge deletion using the nonrecursive rule, which generates a potential deletion of reach(e,f). This approximation is immediately filtered to determine whether there exists another path between e and f. Since there is no alternate route, the tuple (e,f) is identified as an actual change, and is then propagated. The propagation of Δ^-(reach): {(e,f)} identifies (e,g) and (e,h) as potential deletions. However, the filtering phase identifies that there is still a path from e to g, so (e, h) is identified as a removal to reach. The propagation of (e,h) does not identify any potential deletions. The propagation of the initial edge deletion δ^-(edge):{(e, f)} must be propagated through the recursive rule for reach using Δ^-(r22). The potential deletions are immediately filtered, and only actual changes are propagated. The PF algorithm also identifies the tuples {(e, f) (e,h) (b,f) (b,h)} to be removed from the materialized view.

Table 2. PF Algorithm on deletion of edge(e,f) on materialized reach view

PF Algorithm				
Propagate			*Filter*	
	Rule	**δ^-(reach)**	**δ^+(reach)**	**Δ^-(reach)**
δ^-(edge):{(e, f)}	Δ^-(r1)	(e,f)	{}	(e,f)
Δ^-(reach): {(e,f)}	Δ^-(r21)	(e,g) (e,h)	(e,g)	(e,h)
Δ^-(reach): {(e,h)}	Δ^-(r21)	{}		{}
δ^-(edge): {(e, f)}	Δ^-(r22)	(a,f) (b,f)	(a,f)	(b,f)
Δ^-(reach): {(b,f)}	Δ^-(r21)	(b,g) (b,h)	(b,g)	(b,h)
Δ^-(reach): {(b,h)}	Δ^-(r21)	{}		{}

As shown on the above deletion example, the DRed and PF algorithms both compute overestimates or approximations of tuples to be deleted from the recursive materialized view. The PF algorithm eagerly filters the potential deletions before propagating them. The DRed algorithm propagates the potential deletions within a stratum but filters the overestimates before propagating them to the next stratum. There are scenarios in which the DRed algorithm outperforms the PF algorithm and others in which the PF algorithm outperforms the DRed algorithm.

For the case of insertions, the PF algorithm operates in a manner similar to deletions, by approximating the tuples to be added and filtering the potential additions by determining whether the tuple was provable in the old database state. However, the DRed algorithm uses the bottom-up semi-naïve algorithm for Datalog evaluation to provide an inherent mechanism for determining insertions to the materialized view. In semi-naïve evaluation, the original rules are executed once to provide the seed or base answers. Then incremental versions of the rules are executed until a fixpoint is reached. The incremental rules are formed by creating k rules associated with a rule where k corresponds to the number of subgoals in the right-hand side of the rule. The ith incremental rule uses only the new tuples from the last iteration for the ith subgoal. However, when the ith subgoal is a stored relation, then the corresponding incremental rules are removed since they won't contribute to the incremental evaluation. For the motivational example, the incremental rule for reach is

Δreach(S,I), edge(I, D)

where Δreach represents the new reach tuples computed on the previous iteration. Since a set of tuples is being computed, duplicate proofs are automatically filtered and are not considered new tuples. This is the inherent memoing in bottom-up evaluation that handles cycles in the underlying data.

KEY APPLICATIONS

Query Optimization; Condition Monitoring; Integrity Constraint Checking; Data Warehousing; Data Mining; Network Management; Mobile Systems

CROSS REFERENCES

Materialized Views, Incremental View Maintenance, Datalog, Datalog with negation

RECOMMENDED READING

1. Bancilhon F., Maier D., Sagiv Y. and Ullman J. (1986): Magic Sets and Other Strange Ways to Implement Logic Programs. Proceedings of the Symposium on Principles of Database Systems, 1986: 1-15.
2. Ceri S. and Widom J. (1991): Deriving Production Rules for Incremental View Maintenance. Proceedings of the International Conference of Very Large Database Bases. Morgan Kaufmann 1991: 577-589.
3. Dietrich S. W. (1987): Extension Tables: Memo Relations in Logic Programming. Proceedings of the Fourth Symposium on Logic Programming, 1987: 264-272.
4. Dietrich S. W. and Fan C. (1997): On the Completeness of Naive Memoing in Prolog. New Generation Computing, Volume 15. 1997: 141-162.
5. Dong G. and Su J. (2000): Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. SIGMOD Record, Vol. 29, No. 1, 2000: 44-51.
6. Gupta A., Mumick I.S., and Subrahmanian V. S. (1993): Maintaining Views Incrementally. Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM Press 1993: 157-166.
7. Gupta A. and Mumick I. S., editors. (1999): Materialized Views: Techniques, Implementations, and Applications. The MIT Press 1999.

8. Harrison J. V. and Dietrich S. W. (1992): Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. Proceedings of the Workshop on Deductive Databases, Joint International Conference and Symposium on Logic Programming 1992: 56-65.
9. Küchenhoff V. (1991): On the Efficient Computation of the Difference between Consecutive Database States. Proceedings of the International Conference on Deductive and Object-Oriented Database Systems. Springer Verlag 1991: 478-502.
10. Martinenghi D. and Christiansen H. (2005): Efficient Integrity Constraint Checking for Databases with Recursive Views. Advances in Databases and Information Systems 2005: 109-124.
11. Ramakrishnan R., editor. (1995): Applications of Logic Databases. Kluwer 1995.
12. Ramakrishnan R. and Ullman D. (1995): A survey of deductive database systems. Journal of Logic Programming, Vol. 23, No. 2, 1995: 125-149.
13. Ullman J. (1989): Principles of Database and Knowledge Base Systems, Vols 1 and 2. Computer Science Press 1989.
14. Urpí T. and Olivé A. (1992): A Method for Change Computation in Deductive Databases. Proceedings of the International Conference of Very Large Database Bases 1992: 225-237.