Incremental Update Propagation – A Research Proposal

Dimitra Vista

Abstract

For a database query, its *differential query* computes the change to the query's answer induced by changes in the underlying database. We propose to study differential queries in relational and deductive databases.

1. Introduction

We are given a database in the relational or logical model and a data manipulation language. For fast response to queries we allow query answers to be materialized. Such a scenario is desirable in a situation where the query is frequently asked and its answer is expensive to compute. It is also useful when the query's answer is used (as an intermediate result) in subsequent queries. When a change occurs in the underlying database, there are two alternatives for bringing the materialized query answer up to date. The first is to discard the previously stored answer and re-evaluate the query on the updated database. The second is not to discard the previous answer but to compute the *change* to it induced by the database update. The query evaluated to compute the change is a *differential* query. This second approach is called *incremental query evaluation.*

Techniques that optimize a task by eliminating duplicate work among a sequence of computations, by computing the increment from the result of one computation to the next, are often referred to as incremental computations. Incremental computations are used in database systems in several areas, such as view maintenance [BCL89, BLT86, CW91, DT92, GKM92, GMS93, NY83, Rou91, SI84], reasoning about changes [Küc91, UO92], integrity constraint verification [CW90, Ple93], alerting [BC79], and active database systems [BA93, GJS92]. More remote work on incremental techniques includes work on distributed computing [AISN90, Ita91, WDSY91], programming languages [SH91, TR81], maintenance of graph properties [BKV90, CC82], and maintenance of other data structures [CH91, Jag90, KP81, LHM+86]. A study of methods for incremental database query computation appears in [Vis94].

The rationale behind the development and use of incremental algorithms to solve the problem of updating a query answer once the underlying database has changed is that these algorithms perform better than re-evaluation, at least in the majority of cases. Little analysis or experimentation, however, supports this belief. To the best of our knowledge, incremental methods for query computation have not been implemented (with the exception of semi-naive evaluation of logic programs [Ban85] which also utilizes differential expressions to reduce the time of the computation). Incremental methods usually need to store much information in addition to query answers (i.e., many intermediate results). However, it is not clear that maintaining these partial results is better than re-evaluation. For example, maintaining transitive closures may not even be desirable for small graphs. Also, none of the proposed methods is accompanied by adequate complexity analysis or empirical evaluation. We believe that we can make this area of research more scientific by providing sound empirical evidence as to how incremental methods compare with each other and with re-evaluation.

Our motivation for studying differential queries originates from our intention to support them in the Hy$^+$ visualization system. Hy$^+$ is a generic visualization tool that allows visual presentations of databases, queries and query answers [Con94, CEH$^+$94]. When Hy$^+$ is used with applications where the underlying database changes frequently, the data views that the user browses on the screen may quickly become obsolete. Instead of discarding these views and computing new ones, we could compute the changes and refresh them. Examples of interactive applications of Hy$^+$ include distributed computing [CHM93], network management [CH93] and, more recently, world-wide-web network navigation [HMV94].

We propose to pursue work in the following main two areas. We would like to empirically characterize and study existing methods for incremental query evaluation. In this study we would like to use not only real data sets available to us but also well-known database benchmarks, such as the Wisconsin Benchmark [Gra91] and possibly others. Preliminary experiments have indicated a number of ways in which existing approaches can be optimized to generate more efficient differential queries. Therefore, we also propose to develop another algorithm for generating differential queries. We have devised such an algorithm that works for append-only databases and queries without negation and aggregation.

The rest of the proposal is organized as follows. Section 2 presents background information. Sections 3 to 5 present our approach to incremental query evaluation and the new algorithm. Sections 6 to 9 present some results of our preliminary experiment that compares existing methods with our own and with re-evaluation.

2. Background

In the Datalog model, there are two ways relations can be defined. A predicate whose relation is stored in the database is called an *extensional* database (EDB) relation, while one defined by logical rules is called an *intensional* database (IDB) relation. We assume that each predicate symbol denotes either an EDB relation or an IDB relation, but not both. Datalog programs are built from atomic formulas, which are predicate symbols with a list of arguments, such as $p(X_1, \ldots, X_n)$, or built-in predicates, such as $X_1 = X_2$. A *literal* is either an atomic formula or a negated atomic formula. A *rule* is an expression of the form $q \leftarrow p_1, p_2, \ldots, p_n$ where q is an atomic formula whose predicate is an IDB predicate and each p_i is a literal. In a rule $q \leftarrow p_1, p_2, \ldots, p_n$, q is called the head of the rule and each p_i is called a subgoal. A rule is *safe* if every variable appearing in the rule is limited by some positive subgoal, a constant, or another limited variable. A collection of rules is called a *Datalog program*.

The *dependency graph* of a Datalog program is a graph where the nodes are the ordinary predicates. There is an arc from a predicate p to a predicate q if there is a rule with a subgoal whose predicate is p and with a head whose predicate is q. A program is *recursive* if its dependency graph has one or more cycles. A program is *non-recursive* if its dependency graph is acyclic. A program is *linear* if each rule has at most one recursive subgoal. A program is *stratified* if whenever there is a rule with a head predicate q and a negated subgoal with predicate p, there is no path in the dependency graph from q to p. One positive literal (which may contain constants) corresponding to an IDB predicate defined in a Datalog program is called the query predicate.

It has been proven that for each linear stratified Datalog program there exists an equivalent linear stratified Datalog program whose recursive predicates define only transitive closures [Con89]. Given this equivalence result, in what follows, we only consider recursive rules defining transitive closures with rules (and only with such rules) of the form:

$$tc\_p(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \leftarrow p(X_1, \ldots, X_n, Y_1, \ldots, Y_n).$$
$$tc\_p(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \leftarrow p(X_1, \ldots, X_n, Z_1, \ldots, Z_n), tc\_p(Z_1, \ldots, Z_n, Y_1, \ldots, Y_n).$$

3. Our Approach to Incremental Query Computation

Given a linear Datalog program P and a query predicate q, our algorithm derives the differential program D_P^q. The differential program D_P^q uses the insertions made to EDB predicates and their old values to compute the insertions δ_q that are needed to the extension of q. The new value of q is found by combining (using set union) the old value of q with the value of δ_q. The algorithm works for insertions only (at this stage). A modification of the algorithm to generate differential programs that handle deletions is under investigation.

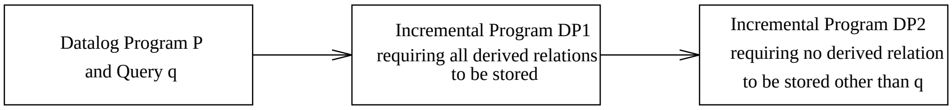

Figure 1: Our approach to incremental query evaluation of Datalog queries

Our method is depicted in Figure 1. Starting from a Datalog program P, a differential program $DP1$ is generated by syntactic manipulation of the rules in P. If this program is to be used for evaluating the change in a query's answer, the values for all derived predicates in the program must be stored in the database. Given the old values of all derived predicates in P and given the changes made to the EDB relations and their old values, the program $DP1$ computes the changes that must be made to every derived predicate and the change in the query predicate q. This approach is similar to other methods in the literature [BLT86, CW91, DT92, GKM92, GMS93].

Storing the values of every derived relation in the database is not space efficient. If we do not wish to store them, we can further manipulate the rules in $DP1$ to produce another program $DP2$ (D_P^q). The differential program $DP2$, given only the changes made to the EDB relations and their old values, computes the changes that must be made to the query predicate q.

The following section presents the first phase of the algorithm, and the section following it presents the second phase. Both are inspired by the translation from GraphLog to Datalog presented in [VW93, VW94].

4. The Basic Algorithm

For simplicity, we use the notation $p(\overline{X})$ to refer to $p(X_1, X_2, \ldots, X_k)$. Also, for EDB predicate p, the predicate δ_p describes the changes made to the relation of p. The algorithm introduces a new predicate

δ_s for each IDB predicate s defined in the program. The algorithm treats non-recursive and recursive rules separately.

Non-Recursive Rules

For each non-recursive rule $p(\overline{X}) \leftarrow s_1(\overline{X_1}), s_2(\overline{X_2}), \ldots, s_k(\overline{X_k})$, we introduce k new predicate symbols $\delta\_r_1, \delta\_r_2, \ldots, \delta\_r_k$. Each of these predicate symbols has a set of variables $\overline{R_i}$ $(i = 1, 2, \ldots, k)$ which is defined as follows: $\overline{R_i} = \overline{X_i} \cap \{\overline{X} \cup \overline{X_{i+1}} \cup \ldots \cup \overline{X_k}\}$. The differential program $DP1$ contains the rules:

- $\delta\_p(\overline{X}) \leftarrow \delta\_r_k(\overline{R_k})$.
- $\delta\_r_1(\overline{R_1}) \leftarrow \delta\_s_1(\overline{X_1})$.
- and, for each $i = 1, \ldots, k-1$ the rules:
  - $\delta\_r_i(\overline{R_i}) \leftarrow \delta\_r_{i-1}(\overline{R_{i-1}}), \delta\_s_i(\overline{X_i})$.
  - $\delta\_r_i(\overline{R_i}) \leftarrow \delta\_r_{i-1}(\overline{R_{i-1}}), s_i(\overline{X_i})$.
  - $\delta\_r_i(\overline{R_i}) \leftarrow \delta\_s_i(\overline{X_i}), s_1(\overline{X_1}), \ldots, s_{i-1}(\overline{X_{i-1}})$.

Each of the new predicate symbols corresponds to changes in an intermediate relation. Predicate δ_r_1 corresponds to changes in s_1, predicate δ_r_2 corresponds to changes in $s_1 \bowtie s_2$, and so on. Predicate δ_r_k, thus, corresponds to changes in the relation $s_1 \bowtie s_2 \bowtie \ldots \bowtie s_k$, that is, the relation of the body of the rule. The arguments of each δ_r_i are these variables of the relation $s_1 \bowtie \ldots \bowtie s_i$ that are needed for computation of the subsequent relations $\delta\_r_{i+1}, \ldots, \delta\_r_k$ (they include the variables of the head and these variables common with subsequent subgoals).

Note that all rules in the differential program have as their first subgoal a delta predicate. Assuming that the changes in a relation are relatively small compared to the size of the relation, the incremental program, in a left to right evaluation, joins a relatively small relation with some other relation(s). Also, if only one relation out of the k appearing in the rule changes, many rules do not fire. The incremental program does not compute any unnecessary tuples, as it only starts the computation from the actual changes made.

As an example, let $p(Y) \leftarrow a(X,Y), b(Y,Z,W), c(Y)$. be the program. We introduce three new predicate symbols, δ_r_1, δ_r_2 and δ_r_3. Predicate δ_r_1 corresponds to the changes in $a(X,Y)$, predicate δ_r_2 corresponds to the changes in $a(X,Y) \bowtie b(Y,Z,W)$, and predicate δ_r_3 corresponds to the changes in $a(X,Y) \bowtie b(Y,Z,W) \bowtie c(Y)$. The variables of δ_r_1 are $\overline{R_1} = \{X,Y\} \cap \{Y,Z,W\} = \{X,Y\}$. The variables of δ_r_2 are $\overline{R_2} = \{Y,Z,W\} \cap \{Y\} = \{Y\}$. The variables of δ_r_2 are $\overline{R_2} = \{Y\} \cap \{Y\} = \{Y\}$. The generated incremental program is:

$\delta\_p(Y) \leftarrow \delta\_r_3(Y)$.
$\delta\_r_1(X,Y) \leftarrow \delta\_a(X,Y)$.
$\delta\_r_2(Y) \leftarrow \delta\_r_1(X,Y), \delta\_b(Y,Z,W)$.
$\delta\_r_2(Y) \leftarrow \delta\_r_1(X,Y), b(Y,Z,W)$.
$\delta\_r_2(Y) \leftarrow \delta\_b(Y,Z,W), a(X,Y)$.
$\delta\_r_3(Y) \leftarrow \delta\_r_2(Y), \delta\_c(Y)$.
$\delta\_r_3(Y) \leftarrow \delta\_r_2(Y), c(Y)$.
$\delta\_r_3(Y) \leftarrow \delta\_c(Y), a(X,Y), b(Y,Z,W)$.

Recursive Rules

For each recursive definition, that is, for each pair of rules $p(\overline{X},\overline{Y}) \leftarrow e(\overline{X},\overline{Y})$. and $p(\overline{X},\overline{Y}) \leftarrow e(\overline{X},\overline{Z}), p(\overline{Z},\overline{Y})$., the incremental program $DP1$ contains the following rules:

- $\delta\_p(\overline{X},\overline{Y}) \leftarrow \delta\_e(\overline{X},\overline{Y})$.
- $\delta\_p(\overline{X},\overline{Y}) \leftarrow \delta\_p(\overline{X},\overline{Z}), e(\overline{Z},\overline{Y})$.
- $\delta\_p(\overline{X},\overline{Y}) \leftarrow \delta\_p(\overline{Z},\overline{Y}), e(\overline{X},\overline{Z})$.
- $\delta\_p(\overline{X},\overline{Y}) \leftarrow \delta\_p(\overline{X},\overline{Z}), \delta\_e(\overline{Z},\overline{Y})$.
- $\delta\_p(\overline{X},\overline{Y}) \leftarrow \delta\_p(\overline{Z},\overline{Y}), \delta\_e(\overline{X},\overline{Z})$.

The intuition behind this differential program is the following. Let $edge(X,Y)$ be a relation and let $path(X,Y)$ be its transitive closure. If the tuple (a,b) is inserted in $edge$, this tuple should be added to the transitive closure (first rule). Then, we append an edge either at the beginning of (a,b) or at the end of it to find other tuples that must be inserted into the transitive closure (second and third rule). And, we do this recursively until no new tuples are generated. If several tuples change in the relation $edge$, we can also append to the generated tuples another new $edge$ tuple, either from the front or from the tail (fourth and fifth rule).

Again, for each rule the first subgoal in a delta predicate.

Note that the insertions in a derived predicate's relation are not necessarily tuples not in the relation already. If the computation of the real delta is desired, for each rule $\delta\_p(\overline{X}) \leftarrow body.$, the fact $\neg p(\overline{X})$ can be appended to derive the rule $real\_\delta\_p(\overline{X}) \leftarrow body, \neg p(\overline{X})$., because $p(\overline{X})$ is stored in the database.

5. The Revised Algorithm

The algorithm presented in the previous section generates a differential program where all the rules have one of the following forms:

$$\begin{aligned}
&\delta_p \leftarrow \delta_s.\\
&\delta_p \leftarrow \delta_s, \delta_s_1.\\
&\delta_p \leftarrow \delta_s, s_1.\\
&\delta_p \leftarrow \delta_s, s_1, s_2, \ldots, s_k.
\end{aligned}$$

If some of the s_i's are not base predicates, this program requires that the value for their relation be available. We can replace each s_i in the rule by s_i's definition, thus, computing the relation of s_i as is needed to propagate changes.

To generate differential program $DP2$ from differential program $DP1$ do:

1. Eliminate all rules of the form $\delta\_p \leftarrow \delta\_s, s_1, s_2, \ldots, s_k$. by introducing k new predicates $\delta\_r_1, \ldots$, δ_r_k and replacing the rule with these new ones:

$$\delta\_r_1 \leftarrow \delta\_s, s_1.$$
$$\delta\_r_i \leftarrow \delta\_r_{i-1}, s_i. \quad \text{for } i = 2, \ldots, k$$
$$\delta\_p \leftarrow \delta\_r_k.$$

2. For each rule $\delta\_p \leftarrow \delta\_s, s_1$, where s_1 is a derived predicate which has a non-recursive definition defined by k rules, replace the rule with these k new ones:

$$\delta\_p \leftarrow \delta\_s, body_i. \quad \text{where } body_i \text{ is the body of rule } i$$

3. For each rule $\delta\_p \leftarrow \delta\_s, s_1$, where s_1 is a derived predicate defined as the transitive closure of e we introduce one new predicate δ_r and replace the rule with these new ones:

$$\delta\_r \leftarrow \delta\_s, e.$$
$$\delta\_r \leftarrow \delta\_r, e.$$
$$\delta\_p \leftarrow \delta\_r.$$

4. Repeat the same process in the resulting program until no rules are modified.

As an example consider a database with the relation `contains` describing the member functions of classes in some object oriented program and the relation `calls` describing the calls between member functions. The following program defines the transitive closure of `calls` and the relation `class_calls` between a class and the functions that the member functions of the class directly or indirectly call.

$$\mathtt{tc\_calls(F1,F2)} \leftarrow \mathtt{calls(F1,F2)}.$$
$$\mathtt{tc\_calls(F1,F2)} \leftarrow \mathtt{calls(F1,F), tc\_calls(F,F2)}.$$
$$\mathtt{class\_calls(C,F2)} \leftarrow \mathtt{contains(C,F1), tc\_calls(F1,F2)}.$$

The first phase of the algorithm introduces the predicates $\delta\_\mathtt{tc\_calls}$ and $\delta\_\mathtt{class\_calls}$ for the derived predicates of the program and the auxiliary predicates $\delta\_\mathtt{r}_1$ and $\delta\_\mathtt{r}_2$. It generates the following incremental program:

$$\delta\_\mathtt{tc\_calls(F1,F2)} \leftarrow \delta\_\mathtt{calls(F1,F2)}.$$
$$\delta\_\mathtt{tc\_calls(F1,F2)} \leftarrow \delta\_\mathtt{tc\_calls(F1,F), calls(F,F2)}.$$
$$\delta\_\mathtt{tc\_calls(F1,F2)} \leftarrow \delta\_\mathtt{tc\_calls(F,F2), calls(F1,F)}.$$
$$\delta\_\mathtt{tc\_calls(F1,F2)} \leftarrow \delta\_\mathtt{tc\_calls(F1,F)}, \delta\_\mathtt{calls(F,F2)}.$$
$$\delta\_\mathtt{tc\_calls(F1,F2)} \leftarrow \delta\_\mathtt{tc\_calls(F,F2)}, \delta\_\mathtt{calls(F1,F)}.$$

$$\delta\_\mathtt{class\_calls(C,F2)} \leftarrow \delta\_\mathtt{r}_2\mathtt{(C,F2)}.$$
$$\delta\_\mathtt{r}_1\mathtt{(C,F1)} \leftarrow \delta\_\mathtt{contains(C,F1)}.$$
$$\delta\_\mathtt{r}_2\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{contains(C,F1)}, \delta\_\mathtt{tc\_calls(F1,F2)}.$$
$$(*) \quad \delta\_\mathtt{r}_2\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{contains(C,F1), tc\_calls(F1,F2)}.$$
$$\delta\_\mathtt{r}_2\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{tc\_calls(F1,F2), contains(C,F1)}.$$

If changes are made to all database relations, $\delta\_\mathtt{contains}$ and $\delta\_\mathtt{calls}$ are both non-empty. This program computes the changes, $\delta\_\mathtt{class\_calls}$, that need to be made to `class_calls`. But it needs to have access to the value of `tc_calls`. We avoid storing the value for `tc_calls` by introducing a new predicate, $\delta\_\mathtt{r}_3$, and replacing the rule $(*)$ with the following three rules:

$$\delta\_\mathtt{r}_3\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{contains(C,F1), calls(F1,F2)}.$$
$$\delta\_\mathtt{r}_3\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{r}_3\mathtt{(C,F1), calls(F1,F2)}.$$
$$\delta\_\mathtt{r}_2\mathtt{(C,F2)} \leftarrow \delta\_\mathtt{r}_3\mathtt{(C,F2)}.$$

6. Experiments

We have conducted preliminary experiments to compare the execution efficiency of existing incremental algorithms for database query evaluation amongst themselves and with re-evaluation. For our experiments, we have used two different sample data sets. The first one describes a software engineering example. The second describes a world wide web navigation using the X Mosaic tool. For each query that was tested, each of the methods was applied to generate the corresponding differential query. These differential queries were then evaluated (without further optimization) using the CORAL deductive database system. All experiments were conducted on a Sparc 10 under low system use. The time indicated in the plots is System and User CPU time in seconds and is the average time produced by 5 different executions of the differential queries.

We have compared the following techniques with re-evaluation:

- The method of Blakeley, Larson and Tompa [BLT86]: given the changes to the database relations, their old values, and a query, this method derives a transaction that updates the answer to the query. It only works for queries specified with select-join-project relational algebra expressions.
- The method of Dong and Topor [DT92]: this method derives, for each regular chain Datalog query, a non-recursive program to compute the change to the answer of the query. It first constructs a regular expression from the query and then generates a logical program recursively from the components of the regular expression.
- The method of Gupta, Mumick and Katiyar [GKM92]: this method needs to store the value for every relation defined in a logic program. Given the logic program defining a set of relations, the method derives a logic program that uses the changes made to base relations and the old values of the derived and base relations to compute the changes to the derived relations. It associates a count with every derived tuple equivalent to the number of alternative derivations of the tuple.
- The method of Kuchenhoff [Küc91]: this method derives a logic program to compute the changes in derived relations in three steps: the first step computes the changes to the dependencies of relations induced by database updates, the second step computes the changes to the derivations induced by the updates, and the third step computes the actual changes in the values of the derived relations.
- The Delete and Rederive method of Gupta, Mumick and Subrahmanian [GMS93]: this method first deletes from a query answer an overestimation of the tuples to be deleted, then rederives those with alternative derivations and finally derives additional facts that must be inserted into the query answer.
- Our method which was developed during our experiments and (an adaptation of it) is implemented in Smalltalk for GraphLog.

We have not tested the algorithm of Ceri and Widom [CW90, CW91]. The reason is that their algorithm depends on key information to be present in the query answer (every defined relation must contain the key of every database relation used to define it), something that we did not have in our examples.

7. Example 1: NIH C++ Class Library

This is a software engineering example. The underlying database in this example is comprised of about 12 000 facts about the structure of the NIH C++ class library. The relationships represented in the database include the `subclass` and `friend` relationships between classes, the `contains` relationship between a class and its member functions, and a `calls` relationship between functions.

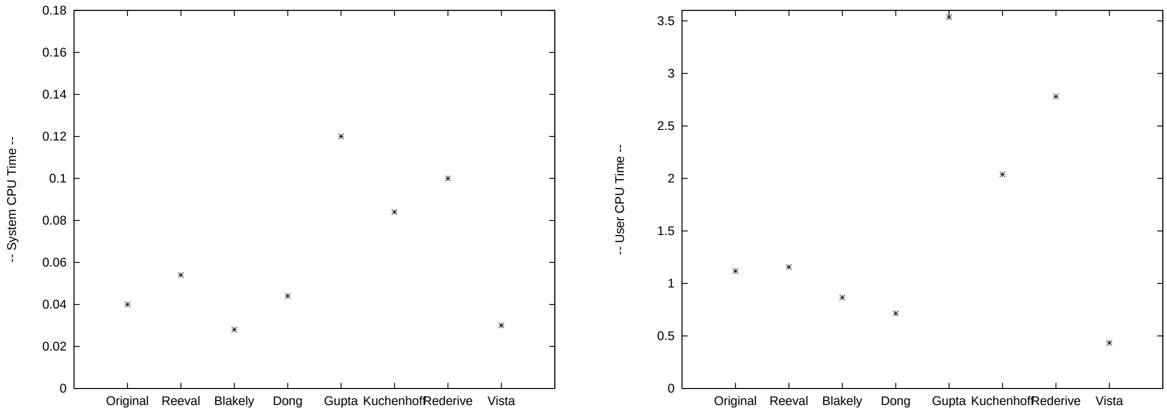

Figure 2: System and User CPU time of differentials for `directly_uses` for the first update

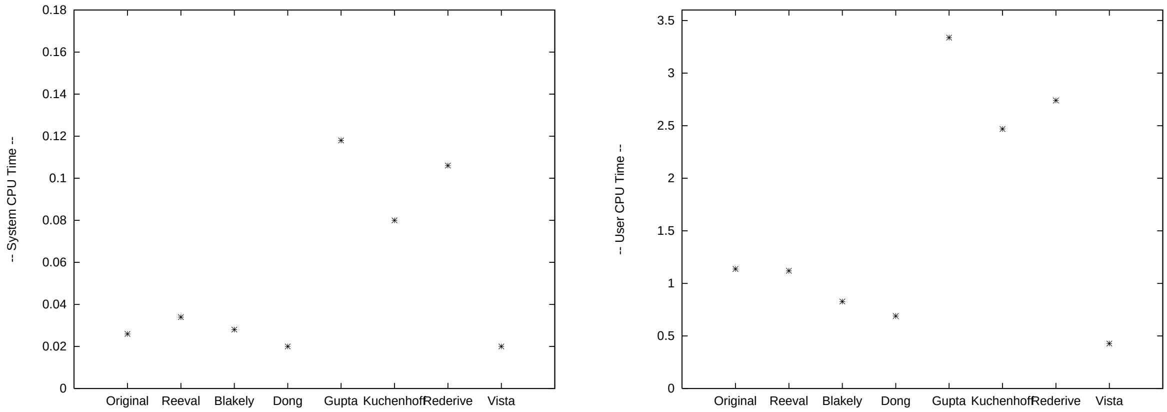

Figure 3: System and User CPU time of differentials for `directly_uses` for the second update

We have tried two queries on this database. The first query computes the relation `directly_uses` between two classes, defined to hold between two classes, when the first class contains a function that calls a function in the second class. This relation involves a three-way join and in Datalog is expressed by the rule:

`directly_uses(C1,C2)` $\leftarrow$ `contains(C1,F1), calls(F1,F2), contains(C2,F2)`.

Figures 2 and 3 depict the execution time of the different methods for two different updates, each one comprised of a single tuple insertion into the `calls` relation. In each figure, two diagrams are shown: the first shows the System CPU time taken by the method and the second the User CPU time, when the methods were tested five times using a machine with low CPU usage.

We can see that, for both updates, the methods by Gupta et al. [GKM92], Küchenhoff [Küc91], and the Delete and Rederive algorithm [GMS93] are significantly worse than re-evaluation for this query. For the method of Gupta et al. [GKM92], the derivation counts that are needed are computed through aggregation which, in general, is expensive. As the authors claim in their paper, if the database management system's evaluation component is changed so that it computes these counts during query evaluation, their computation can be done at no extra cost above the cost of the program evaluation. Our method, labelled "Vista" in the plots, the method by Dong, and that of Blakeley et al. surpass all tested methods, but only with a slight improvement.

The second query that we tried defines the relation `uses` between two classes when the first class contains a function that either directly or indirectly calls a function in the second class. A three-way join and a transitive closure are needed for this query, which in Datalog is computed by the following rule (where `tc_calls` denotes the transitive closure of `calls`):

$$\texttt{uses(C1,C2)} \leftarrow \texttt{contains(C1,F1), tc\_calls(F1,F2), contains(C2,F2)}.$$

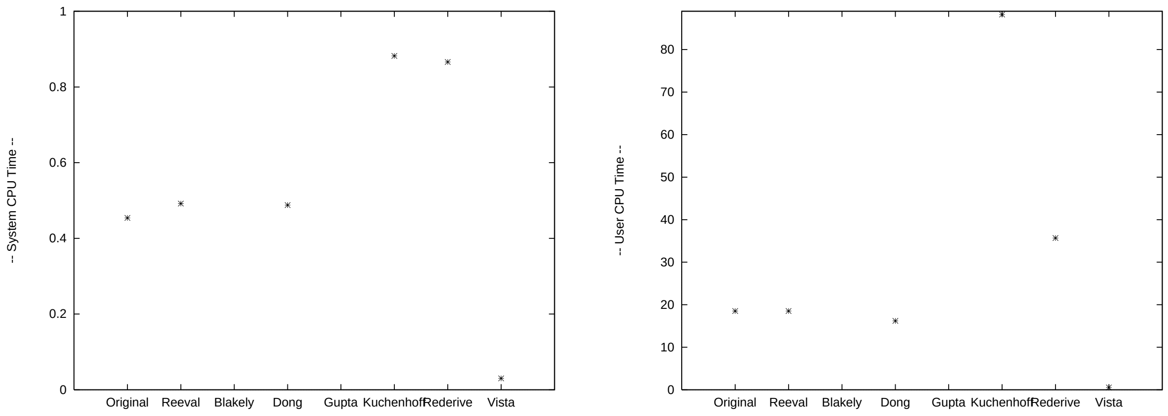

Figure 4: System and User CPU time of differentials for `uses` when only `uses` is stored

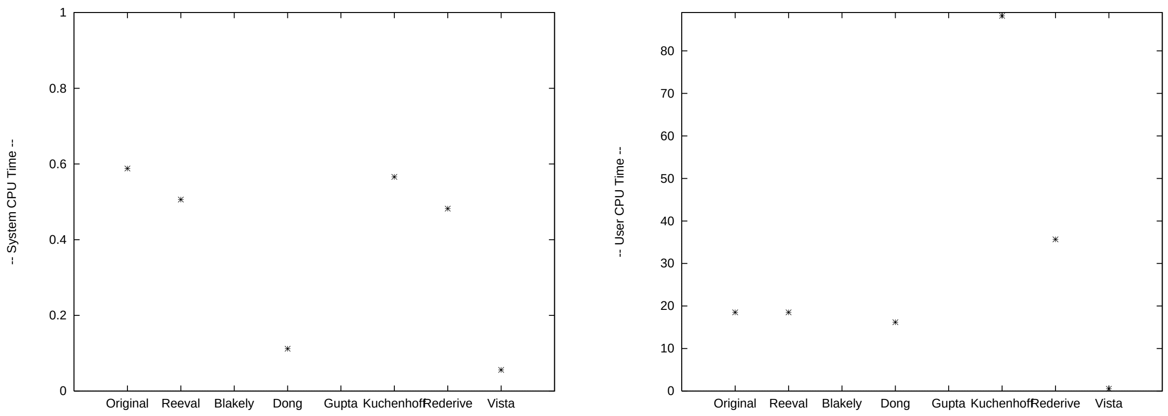

Figure 5: System and User CPU time of differentials for `uses` when both `uses` and `tc_calls` are stored

In executing the differentials for this query, we had two alternatives: to store only the relation `uses` or to store it together with the `tc_calls` relation. Figures 4 and 5 reflect these two possibilities. Again, the first graph of each figure represents System CPU time and the second User CPU time. Both figures refer to the same update, a single tuple addition to the `calls` relation. As one can see, there is no time depicted for the counting method of Gupta et al. and for the method of Blakeley et al. This is because these methods do not support recursive queries.

When the transitive closure is not stored and the incremental method has to compute it, the method of Küchenhoff and the Delete and Rederive algorithm are worse than re-evaluation. When the transitive closure is stored, these methods have the same time efficiency as re-evaluation. Note that this is in contrast with the previous example (Figures 2 and 3) where again a three-way join is maintained but where these methods are by far worse than re-evaluation. We believe the reason for this anomaly is that the original program in the second experiment is by far more expensive to compute and it is the computation of transitive closure in the original program that slows down the original program rather than the differential programs being faster in this case.

The method by Dong [DT92] and the method by Vista do not depend on `uses` being stored. The method by Dong depends on `tc_calls` being stored while the method by Vista does not (this is the reason why this method performs consistently in both Figure 4 and Figure 5). The method by Dong produces a non-recursive program for the maintenance of this recursive program if `tc_calls` is stored and this is the reason that it behaves so well in this case. Our method computes the change in the transitive closure only as needed, starting from the point where `calls` changed and finding the changes in paths involving this change (the immediate `tc_calls` changed) and so on. This method performs significantly better than re-evaluation.

8. Continuing the experiments

The generated differential programs of the previous sections were tested with no further optimization. Whether combining the methods with other optimization techniques, such as magic sets [BMSU86, BR87], factoring [NRSU89b, NRSU89a], and so on, improves their performance remains to be seen. In other experiments, not shown here, we have seen that magic sets improve the performance of Küchenhoff but weaken all other methods.

All experiments measure the User and System CPU time taken by each tested differential query. Disk I/O could also be a relevant factor in characterizing the different queries, especially in large databases. Both example databases that were tested could fit in the main memory of a machine, so disk I/O was not relevant. One of them, the software engineering database, is very difficult to scale. But the X Mosaic database could become larger easily. In general, when dealing with real data it is rather difficult to scale. Another problem with real data is that the values they contain are not flexible enough to permit systematic benchmarking. For example, with real data, it is very difficult to specify a selection query with a 10 percent or 50 percent selectivity factor or one that retrieves a given number of tuples. To have some control over the data distribution, the join selectivities of relations and other such factors, we plan to use the Wisconsin Benchmark. Another data set that could be used in our experiments may be from the Human Genome project [HB94].

Number of page faults, total number of CPU cycles and other such measures could also be relevant. Confidence level number will be assigned to each experiment.

9. Conclusions

We have presented experiments that we have conducted in order to find out which of the proposed algorithms for incremental query computation perform better than re-evaluation. Our experiments were conducted on two sample databases with arbitrarily chosen queries.

We have also devised and implemented a different method for incremental query computation. From our preliminary experiments, we see that this method performs better than other methods. The method works for insertions only and for programs without negation and aggregation. We would like to generalize it to also work in these cases.

References

[AISN90] G. Ausiello, G. Italiano, A.M. Spaccamela, and U. Nanni. Incremental Algorithms for Minimal Length. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–21, 1990.

[BA93] D. Barbará and R. Alonso. Answering Continuous Queries in General Environments. Technical report, Matsushita Information Technology Laboratory, 1993.

[Ban85] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*. Springer-Verlag, 1985.

[BC79] O.P. Buneman and E.K. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Data Base Systems*, 4(3):368–382, 1979.

[BCL89] J.A. Blakeley, N. Coburn, and P-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Data Base Systems*, 14(3):369–400, 1989.

[BKV90] A.L. Buchsbaum, P.C. Kanellakis, and J.S. Vitter. A Data Structure for Arc Insertion and Regular Path Finding. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 22–31, 1990.

[BLT86] J.A. Blakeley, P-A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 61–71, 1986.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 269–283, 1987.

[CC82] G.A. Cheston and D.G. Corneil. Graph Property Update Algorithms and their Applications to Distance Matrices. *INFOR*, 20(3):178–201, 1982.

[CEH+94] M.P. Consens, F.Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, M.G. Noik, A.G. Ryman, and D. Vista. Architecture and Applications of the Hy+ Visualization System. *IBM Systems Journal*, 33(3), 1994.

[CH91] J-P. Cheiney and Y-N. Huang. Set-Oriented Propagation of Updates Into Transitively Closed Relations. In *Proceeding of DOOD*, pages 503–523, 1991.

[CH93] M.P. Consens and M.Z. Hasan. Supporting Network Management through Declaratively Specified Data Visualizations. In *Proceedings of the IFIP/IEEE 3rd International Symposium on Integrated Network Management, III*, pages 725–738. Elsevier North Holland, 1993.

[CHM93] M.P. Consens, M.Z. Hasan, and A.O. Mendelzon. Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, 1993. Extended abstract.

[Con89] M. P. Consens. Graphlog: "Real Life" Recursive Queries Using Graphs. Master's thesis, Department of Computer Science, University of Toronto, 1989.

[Con94] M.P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*. PhD thesis, University of Toronto, Dept. of Computer Science, 1994.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proceeding of the 16th International Conference on Very Large Data Bases*, pages 566–577, 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceeding of the 17th International Conference on Very Large Data Bases*, pages 577–589, 1991.

[DT92] G. Dong and R. Topor. Incremental Evaluation of Datalog Queries. In J. Bishup and R. Hull, editor, *Proceeding of 4th International Conference on Database Theory*, pages 282–296, 1992.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object Oriented Database. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 81–90, 1992.

[GKM92] A. Gupta, D. Katiyar, and I.S. Mumick. Counting Solutions to the View Maintenance Problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 157–166, 1993.

[Gra91] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, 1991.

[HB94] E. Harley and A.J. Bonner. A Flexible Approach To Genome Map Assembly. In *Proceedings of the International Symposium on Intelligent Systems for Molecular Biology*, pages 161–169, 1994.

[HMV94] M.Z. Hasan, A.O. Mendelzon, and D. Vista. Using Hy^+ to help navigage through X Mosaic. Work in Progress, 1994.

[Ita91] G.F. Italiano. Distributed Algorithms for Updating Shortest Paths (EXTENDED ABSTRACT). In *Workshop on Distributed Algorithms and Graphs*, pages 200–211, 1991.

[Jag90] H.V. Jagadish. A Compression Technique to Materialize Transitive Closure. *ACM Transactions on Data Base Systems*, 15(4):558–598, 1990.

[KP81] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *7th International Conference on Very Large Databases*, pages 306–318, 1981.

[Küc91] V. Küchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *Proceeding of DOOD*, pages 478–502, 1991.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 53–60, 1986.

[NRSU89a] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Argument Reduction by Factoring. In *Proceeding of the 15th International Conference on Very Large Data Bases*, pages 173–182, 1989.

[NRSU89b] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Efficient Evaluation of Right-, Left-, and Combined-linear Rules. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 235–242, 1989.

[NY83] J-M. Nicolas and K. Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Information Processing*, pages 711–717, 1983.

[Ple93] D. Plexousakis. Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 146–157, August 1993.

[Rou91] N. Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Data Base Systems*, 16(3):535–563, 1991.

[SH91] R.S. Sundaresh and P. Hudak. Incremental Computation via Partial Evaluation. In *Synposium on Principles of Programming Languages*, pages 1–13, 1991.

[SI84] O. Shmueli and A. Itai. Maintenance of Views. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 240–255, 1984.

[TR81] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[UO92] T. Urpí and A. Olivé. A Method for Change Computation in Deductive Databases. In *Proceeding of the 18th International Conference on Very Large Data Bases*, pages 225–237, 1992.

[Vis94] D. Vista. View Maintenance in Relational and Deductive Databases by Incremental Query Evaluation. In *IBM CASCON Conference, CD-ROM,*, 1994.

[VW93] D. Vista and P.T. Wood. Efficient Visual Queries for Deductive Databases. In *Proceedings of the Workshop on Programming with Logic Databases*, pages 44–59, 1993. Available as Technical Report 1183, Computer Sciences Department, University of Wisconsin-Madison.

[VW94] D. Vista and P.T. Wood. Efficient Evaluation of Visual Queries Using Deductive Databases. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic Publ., 1994.

[WDSY91] O. Wolfson, H.M. Dewan, S.J. Stolfo, and Y. Yemini. Incremental Evaluation of Rules and its Relationship to Parallelism. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 78–87, 1991.