Optimizing Active Databases using the Split Technique⋆ (Preliminary Report)

Serge Abiteboul[1] and Allen Van Gelder[2]

[1] INRIA, Serge.Abiteboul@inria.fr
[2] University of California at Santa Cruz and INRIA, avg@cs.ucsc.edu

Abstract. A method to perform nonmonotonic relational rule computations is presented, called the *split technique*, The goal is to avoid redundant computations with rules that can insert and delete sets of tuples specified by the rule body. The method is independent of the control strategy that governs rule firing. Updatable relations are partitioned, as the computation progresses, into blocks of tuples such that tuples within a block are indiscernible from each other based on the computation so far. Results of previous rule firings are remembered as "relational equations" so that a new rule firing does not recompute parts of the result that can be determined from the existing equations. Seminaive evaluation falls out as a special case when all rules specify inserts. The method is amenable to parallelization.

1 Introduction

Database computations with explicit delete and insert operations are gaining more and more attention (e.g., [AV90, AV91a, NK88, MW88]). Such computations arise not only in databases with so-called *active* features (e.g., [Esw76, BC79, SHP88, dMS88, MD89, Han89, HJ91, Wid91]), but also in production rule systems (e.g., OPS5 [BFKM85], RDL1 [dMS88]). However, little is known about the optimization of such computations. The main known technique is seminaive, or differential, computation (see [BR86]). However, it applies only to relations that are used monotonically. The technique presented here can be viewed as a generalization of this technique to relations that are allowed to both increase and decrease.

Our method, which we call the *split technique*, applies to computations that are specified as follows. There is a set of rules of the form

$$\pm t(\mathbf{t}) \leftarrow r_1(\mathbf{r}_1), \ldots, r_k(\mathbf{r}_k), \neg s_1(\mathbf{s}_1), \ldots, \neg s_m(\mathbf{s}_m).$$

where $\mathbf{t}$, $\mathbf{r}_i$ and $\mathbf{s}_j$ denote (not all distinct) vectors of variables such that all variables appear in some positive premise. In the rule head, "+" denotes that tuples $\mathbf{t}$ are to be inserted into relation t, while "−" denotes deletion from relation t. When a rule is "fired", all tuples computed by the body are applied "simultaneously" to the target t. However, the control of rule firings is itself an arbitrary computation, which is not known to, and is independent of, our relational computation method.

⋆ Partially funded by PRC-BD3 (French Ministry of Research) and National Science Foundation grant CCR-8958590.

As a motivating example of when the technique might be applied beneficially, suppose certain derived relations are precomputed from a database to speed up queries; a simple example is materializing transitive closure, but we contemplate more complicated materializations. These derived relations might be updated only periodically, due to the expense. For example, a day's transactions on the underlying database might be collected and used *en masse* to update the materializations over night. Updating the materializations could well be done by a rule program of the type described above.

Other possible applications are the computation of models associated with various nonmonotonic logic program semantics, such as well-founded models via the alternating fixpoint technique [VG92] and stable models via the backtracking fixpoint technique [SZ90], or the tie-breaking fixpoint technique [PY92].

The main idea of the split technique is that while performing the actual computation, the system does some relatively inexpensive bookkeeping that will permit it to avoid redoing the same computations twice. This bookkeeping consists in clustering logically indiscernible tuples into blocks. Intuitively, a block contains a set of tuples that have behaved similarly up to now, in other words, that have not been differentiated by the computation so far. The blocks constitute a partition of the relational domain, that is, they are pairwise disjoint. Blocks of the partition will be refined only when necessary for the computation.

The technique described here was inspired by theoretical works on the expressive power of partial fixpoint logics [AV91a, ACV92]. Partial fixpoint logic can alternatively be viewed as relational algebra together with assignment and iteration, and so is clearly related to database updates. The known worst case complexity for such languages is polynomial space. However, it is argued in [AV91a, ACV92] that this bound is unrealistic. Indeed, it is demonstrated in [ACV92] that a logspace bound can be achieved for random data distribution *on the average*. The work presented here can be viewed as a practical implementation of some ideas developed in [AV91a, ACV92].

In the next section, we present the algorithm. In the following one, we address the issue of its performances. (This is a preliminary report. A prototype in Prolog has been written to validate the technique. Limited precise measurements will be given.) In a last section, we further discuss various aspects of the technique as well as possible improvements.

2 The Technique

In this section, we describe the technique. First, we give an informal overview, then we introduce the framework and some necessary notation, and follow with a more detailed description of the algorithm and data structures.

2.1 Overview

The core of the technique consists in partitioning the set of tuples over an updatable relation into blocks. Each block is given a *block identification number* (*bin*). The goal is to try as much as possible to compute *with the bins* instead of *with actual data*. Of course, the number of blocks needs to be small compared to the number of tuples. When this is not the case, the technique will perform poorly, that is, there will not be any substantial gain and on the other hand, the overhead consisting of the management of *bins* will not be negligible. This issue is discussed in the paper.

A *bin* for a block for relation t is denoted by the symbol t with an integer as a subscript. These integers should be thought of as sequences of bits. First, the set of tuples over the schema of t is split into two blocks: t_{10} denotes the content of t originally and t_{11} its complement. When we refine a block of t, say t_{101}, we split it into two new blocks t_{1010} and t_{1011}, this will allow to easily compare blocks corresponding to the same relation: they intersect only if the index of one is a prefix of that of the other, in which case the second is included in the first. (Such comparison can be realized fast using test for equality and bits shifting.)

At each stage, a table gives a summary of the computations remembered thus far. This summary gives information on the combinations of *bins* in terms of other *bins*. Because we do not want to split unnecessarily blocks of tuples, some information will involve somewhat approximate information when exact one would have produced an unnecessary split. This will become clear when we detail the technique.

2.2 Framework

Le $\mathbf{R}$ be a database schema, i.e., a set of relation names together with some fixed arities for these relations. Let $\mathcal{R}$ be a set of rules over $\mathbf{R}$ of the form

$$\begin{array}{rcl} +t(\mathbf{t}) & \leftarrow & r(\mathbf{r}), [\neg]s(\mathbf{s}). \\ & \text{or} & \\ -t(\mathbf{t}) & \leftarrow & r(\mathbf{r}), [\neg]s(\mathbf{s}). \end{array}$$

The "+" indicates insertion and "−" deletions from t. The premisses are interpreted in a standard way; $[\neg]$ indicates that the second premise may be negated. The second subgoal may be absent, in which case it is interpreted as *true*, to achieve a projection. We require that every variable in the rule appears in a positive literal of the body. As an abbreviation, we also allow an "assignment":

$$t(\mathbf{t}) := r(\mathbf{r}), [\neg]s(\mathbf{s}).$$

which means $-t(\mathbf{t}) \leftarrow t(\mathbf{t})$ followed by $+t(\mathbf{t}) \leftarrow r(\mathbf{r}), [\neg]s(\mathbf{s})$.

Example 1. Examples of rules are as follows

$$\begin{array}{ll} +tc(X,Y) & \leftarrow tc(X,Z), t(Z,Y). \\ bad(X) & := t(Z,X), \neg wf(Z). \\ -bhv(X) & \leftarrow dom(X), \neg bad(X). \end{array}$$

□

The restriction to bodies with at most two literals is for the sake of simplicity of presentation. For more than two literals, the technique can easily be extended or such program can be easily simulated (with some overhead) using intermediary relations. Similarly, rules with several actions in the head are easily simulated in this framework.

The program will consist of the application of rules in $\mathcal{R}$ in some sequence. This could be applying the rules in $\mathcal{R}$ in turns until some fixpoint is reached or a more sophisticated control. The split technique is entirely independent of the control.

Example 2. (Well-Founded Vertices) To test meaningful examples with the prototype, we have implemented a simple control language in the spirit of the rule algebra of Imielinski and Naqvi [IN88]. The rules and the control are illustrated in the following example.

Consider a directed graph of edges e. A vertex is "well-founded" if there exist no infinite paths descending from it, i.e.,

$$wf(X) \leftarrow dom(X) \wedge \forall Z[e(Z,X) \rightarrow wf(Z)]$$

logically defines the desired relation. The well-founded vertices can be obtained by the following program:

$$\begin{array}{lll} \alpha_0 : +dom(X) & \leftarrow & e(X,Y). \\ \alpha_1 : +dom(X) & \leftarrow & e(Y,X). \\ \alpha_2 : bad2(X,Z) & := & e(Z,X), \neg wf(Z). \\ \alpha_3 : bad(X) & := & bad2(X,Z). \\ \alpha_4 : +wf(X) & \leftarrow & dom(X), \neg bad(X). \end{array}$$

with

$$fire[\alpha_0, \alpha_1], fixpt[\alpha_2, \alpha_3, \alpha_4]$$

as the control. Observe that the ":=" here is just syntactic sugaring. For example, rule α_2 can be viewed as the composition of the two rules:

$$\begin{array}{l} \alpha_2' : -bad2(X,Z) \leftarrow bad2(X,Z). \\ \alpha_2'' : +bad2(X,Z) \leftarrow e(Z,X), \neg wf(Z). \end{array}$$

Rules α_2 and α_3 are separated because relational difference cannot be combined soundly with projection in our technique. Finally, commas in the control part are used for composition and *fixpt* specifies iteration until there is no change to the last target (wf in α_4). □

We will use the following notation, in which $\mathbf{t}$, $\mathbf{x}$, and $\mathbf{y}$ are vectors of variables, not all distinct. Let rule α_i be $\pm t(\mathbf{t}) \leftarrow r(\mathbf{x}), s(\mathbf{y})$, and let $\mathcal{A}$, $\mathcal{B}$ be some sets of r and s tuples respectively. We denote by $\alpha_i[\mathcal{A}, \mathcal{B}]$ the following set of tuples over t:

$$\{v\mathbf{t} \mid \text{ for some valuation } v, v\mathbf{x} \in \mathcal{A} \text{ and } v\mathbf{y} \in \mathcal{B}.\}$$

Similarly, for a rule $\alpha_j = \pm t(\mathbf{t}) \leftarrow r(\mathbf{x}), \neg s(\mathbf{y})$, the set

$$\{v\mathbf{t} \mid \text{ for some valuation } v, v\mathbf{x} \in \mathcal{A} \text{ and } v\mathbf{y} \notin \mathcal{B}\}$$

is denoted $\alpha_j[\mathcal{A}, \mathcal{B}]$.

2.3 The Representation

Let I be a database instance over scheme $\mathbf{R}$ and t an updatable relation in $\mathbf{R}$. We maintain a sequence of *bins* for t: $t_{i_1}, \ldots, t_{i_{comp}}$. Each *bin* t_δ denotes a set of tuples of the arity $arity(t)$, denoted $val(t_\delta)$. A boolean function $current(t_\delta)$ tells for each t_δ, whether $val(t_\delta)$ is included in the current relation instance $I(t)$. We further enforce the following constraints:

1. $\{val(t_{i_1}), ..., val(t_{i_{comp}})\}$ is a partition of the set of $arity(t)$ tuples;
2. $\{val(t_\delta) \mid t_\delta \text{ is current}\}$ is a partition of $I(t)$; and
3. $t_{i_{comp}}$ is never current.

We will show how to maintain at each step of the computation such a data structure for each updatable relation.

From an implementation point of view, we store $val(t)$ for each *bin* t except for the last one $t_{i_{comp}}$ which has a particular status (and denotes the complement of the others). We also store the set of current ones. Observe that a nonupdatable relation, say r, can be viewed as consisting of two *bins*, r_2 and r_3, where $val(r_2)$ is just the value of r and $r_3 = r_{comp}$ is the complement; only r_2 is current.

We also memorize results of previous computations in a table, the *transition table*. The table for a rule $\alpha = \pm t(\mathbf{t}) \leftarrow r(\mathbf{x}), (\neg)s(\mathbf{y})$ consists of three columns. A tuple

$$[r_i, s_j, \{t_{i_1}, ..., t_{i_k}\}]$$

in the transition table for that rule indicates that

1. r_i, s_j are *bins* for r, s, respectively, and $t_{i_1}, ..., t_{i_k}$ are pairwise disjoint *bins* for t; and
2. $\alpha[val(r_i), val(s_j)] = val(t_{i_1}) \cup ... \cup val(t_{i_k})$.

More precisely, some of the *bins* in the third column may be annotated with the sign "$\subset$". The interpretation of a tuple

$$[r_i, s_j, \{t_{i_1}, ..., t_{i_k}, t_{j_1}^{\subset}, ..., t_{j_l}^{\subset}\}]$$

is that $\alpha[val(r_i), val(s_j)]$ satisfies the following lower and upper bounds:

$$\begin{aligned} val(t_{i_1}) \cup \cdots \cup val(t_{i_k}) &\subseteq \alpha[val(r_i), val(s_j)] \\ &\subseteq val(t_{i_1}) \cup \cdots \cup val(t_{i_k}) \cup val(t_{j_1}) \cup \cdots \cup val(t_{j_l}) \end{aligned}$$

The use of the "$\subset$" annotation will become clear soon.

As mentioned, a *bin* for a relation t will be the symbol t indexed with a string of bits, e.g., t_{1011} or t_{11110}. When a block t_δ is split, this yields two new blocks $t_{\delta 0}$ and $t_{\delta 1}$. This particular naming schema enforces the constraint that:

1. $val(t_\delta) \subseteq val(t_\beta)$ iff β is a prefix of δ, and
2. $val(t_\delta) \cap val(t_\beta) \neq \phi$ iff $val(t_\delta) \subseteq val(t_\beta)$ or $val(t_\beta) \subseteq val(t_\delta)$.

As a consequence, we just have to maintain the function *val* for the most refined blocks.

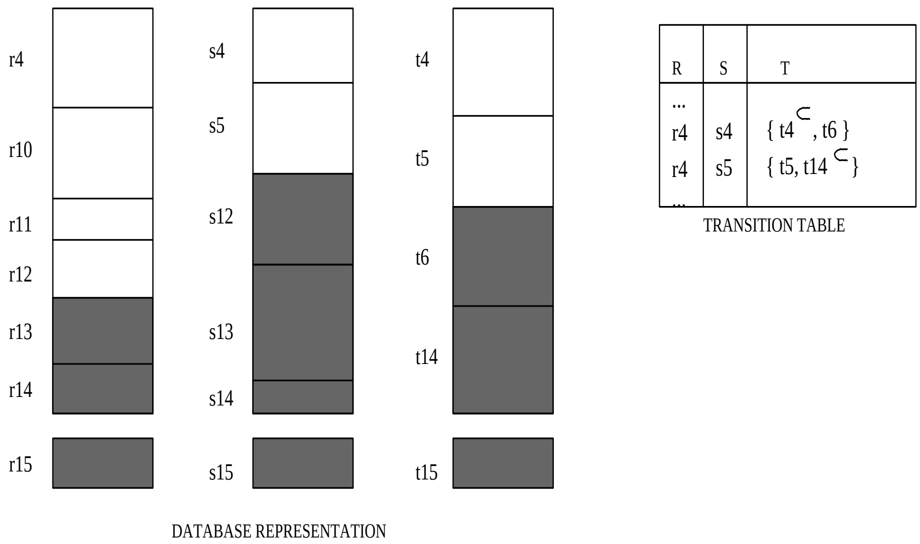

Fig. 1. A configuration

2.4 An Example

Consider a schema consisting of three relations r, s, t and the database of Figure 1 during some computation. The sets of r,s, and t-tuples have been split respectively in 7, 6 and 5 blocks. The shaded areas correspond to tuples not belonging to the relations. (The number of current blocks in r, s and t are respectively 4,2 and 2.)

Suppose first that the activated rule is the following insertion rule

$$+t(X, Z) \leftarrow r(X, Z), s(Z, Y).$$

with the transition table given in the figure.

We have to add

$$\cup_{i \in \{4,10,11,12\}} \cup_{j \in \{4,5\}} \alpha[val(r_i), val(s_j)]$$

to t. By inspection of the transition table, we find out that the join of the blocks r_4 and s_4 yields some tuples from t_4 and the whole of t_6. There is no need to compute that join and it suffices to turn t_6 from shaded to white, i.e., to make it current.

On the other hand, consider the join of r_4 and s_5. It contains some tuples from t_{14} and possibly not all. This join has to be computed (and not the join of the entire relations). It may lead to splitting t_{14} into new blocks t_{28}, t_{29}, one will be current and the other not. When this happens, the transition table is updated to record the new information. In this particular case, the second line is replaced by:

$$[\ r_4,\ \ s_5,\ \ \{\ t_5,\ t_{28}\ \}\ ].$$

A computation of the join is also needed if one finds no information on the join of two blocks.

Now suppose instead that we are activating the deletion rule

$$-t(X, Z) \leftarrow r(X, Z), s(Z, Y).$$

and that its transition table is the one shown in the figure. The join of r_4 and s_5 leads to deleting t_5 which can be done by simply changing its color, plus deleting tuples in t_{14} that were already non existing, so there is nothing else to do.

Finally consider r_4 and s_4. This leads to deleting "some" tuples from t_4 but not all. Thus we cannot avoid computing this join. This may lead to the splitting of t_4 into one block which will remain white and one (the deleted tuples) which will be shaded.

2.5 The Algorithm

The algorithm works as follows.

Insertion and Positive Premisses We consider first the application of a rule $\alpha = +t(\mathbf{t}) \leftarrow r(\mathbf{x}), s(\mathbf{y})$. The other cases are very similar and are considered next. We have to consider all the combinations of current r_i, s_j. For a pair r_i, s_j, several cases may arise:

1. The transition table for rule α gives an upper bound on $\alpha[val(r_i), val(s_j)]$, say

   $$\{t_{i_1}, ..., t_{i_k}, t^{\subseteq}_{j_1}, ..., t^{\subseteq}_{j_l}\},$$

   and the "upper bound" blocks ($t^{\subseteq}_j$) are all included in blocks that are current in t:
   In this case, it is not necessary to compute $\alpha[val(r_i), val(s_j)]$. The function *current* is updated (if necessary) and the transition table is not modified. By updating *current*, we mean that we make current all of the most refined *bins* that are subsets of $\{t_{i_1}, ..., t_{i_k}\}$.
2. Otherwise, we use an upper bound on the *bins* for t that may be affected. This upper bound χ is $\{t_{i_1}, ..., t_{i_k}, t_{j_1}, ..., t_{j_l}\}$ if there is a tuple

   $$[r_{i'}, s_{j'}, \{t_{i_1}, ..., t_{i_k}, t^{\subseteq}_{j_1}, ..., t^{\subseteq}_{j_l}\}]$$

   in the transition table with $r_{i'}, s_{j'}$ supersets of r_i, s_j. If no such information is found, we use for χ the set of all most refined partitions of t obtained so far.
   The set $\mathcal{A}ij = \alpha[val(r_i), val(s_j)]$ is next computed; this represents the result of a join, possibly followed by a projection if some variables in the rule body are absent from the head. We must then compute a representation of $\mathcal{A}ij$ as a set of *bins* (possibly with annotations). Let ν_{ij} be that set. For each $val(t_\delta)$ for t_δ in χ, the following cases occur:
   (a) $val(t_\delta) \cap \mathcal{A}ij = \phi$.
   The *bin* t_δ is omitted from ν_{ij}.
   (b) $val(t_\delta) \subseteq \mathcal{A}ij$.
   t_δ is in ν_{ij}.

(c) $val(t_\delta) \cap \mathcal{A}ij \neq \phi$, $val(t_\delta) - \mathcal{A}ij \neq \phi$ and t_δ is not current.
We compute two new *bins* $t_{\delta 0}$ and $t_{\delta 1}$ with

$$(\dagger) \quad t_{\delta 0} = val(t_\delta) \cap \mathcal{A}ij \quad \text{and} \quad t_{\delta 1} = val(t_\delta) - \mathcal{A}ij.$$

Only $t_{\delta 0}$ belongs to ν_{ij}. (If t_δ is the current t_{comp}, then $t_{\delta 1}$ is not computed and becomes the new t_{comp}.) This operation is called *refinement*, and was counted in the experimental work.

(d) $val(t_\delta) \cap \mathcal{A}ij \neq \phi$, $val(t_\delta) - \mathcal{A}ij \neq \phi$ and t_δ is current.
Then $t_\delta^{\subseteq}$ belongs to ν_{ij}.

It is important to note that when updating the partition for t, we try to avoid introducing too many unnecessary blocks. In that spirit, we do not split a block t_δ already in t to distinguish tuples that were already in t and are inserted again from tuples that were in t and are not inserted again (case (2d)). The price to pay is the introduction of the annotation and the fact that the transition table contains "pseudo" equations instead of perfect ones.

Deletion and Positive Premisses We now consider the case of deletion rules $-t(\mathbf{t}) \leftarrow r(\mathbf{x}), s(\mathbf{y})$.

The algorithm is identical with noncurrent/current playing the roles of current/noncurrent above. For instance, (1) becomes:

the transition table for rule α gives for $\alpha[val(r_i), val(s_j)]$,

$$\{t_{i_1}, ..., t_{i_k}, t_{j_1}^{\subseteq}, ..., t_{j_l}^{\subseteq}\},$$

and the "incomplete" blocks are all included in blocks that are not current in t. In that case, it is not necessary to compute $\alpha[val(r_i), val(s_j)]$. The function *current* is updated (if necessary) and the transition table is not modified.

Negative Premisses With negative premisses, the join operation corresponding to $\alpha[val(r_i), val(s_j)]$ is now a difference. The treatment is again rather similar and a tuple in the transition table for a rule $\pm t(\mathbf{t}) \leftarrow r(\mathbf{x}), \neg s(\mathbf{y})$ indicates the result of a difference.

It is not immediately obvious how to factor the computation so $\alpha[val(r_i), val(s_j)]$ is reusable. We use the identity,

$$\cup_i r_i - \cup_j s_j = \cup_i \cap_j (r_i - s_j)$$

Efficiency is gained by exploiting the pairwise disjointness of the s_j.

Thus, the main difference from the case of positive premisses is that for a given i, we take the intersection over all j of $\alpha[val(r_i), val(s_j)]$ instead of the union. However, projection cannot be combined soundly with this method of computing set difference, so all variables appearing in the rule must appear in its head. The main structure of the algorithm remains the same.

3 Performances

This is a preliminary report and the question of the performance improvements of the technique and its range of applicability is somewhat open. We briefly discuss here its theoretical foundations and some preliminary results obtained with a prototype.

3.1 Theory

The present work is based on results from [AV91b] and [ACV92]. In [AV91b], a normal form for partial fixpoint computation is demonstrated that consists of first partitioning the set of tuples into equivalence blocks which are given identification numbers (similar to the *bin* here), computing tables that are in the spirit of our transition tables, and then abstractly computing on the id numbers. This was further analyzed in [ACV92] where it is shown that *on average,* this heavily reduces the cost of partial fixpoint logic computations compared to a standard algorithm. Indeed, the cost goes from polynomial space to logarithmic space.

However, these theoretical results rely heavily on the fact that the data distribution is random and somewhat uniform (which is arguable on real database examples). Also the results are obtained at the limit when the database size grows to infinity. These two conditions guarantee that the number of *bin*s is very small compared to the size of the database even if the splitting policy is very radical.

For these reasons, the theoretical bounds are not directly usable. However, they are strong indications of the potentiality of the technique.

The main reason of the impracticality of the algorithm in [ACV92] is that it requires computing the active domain and for each relation of arity k, its k-th product. The next step is as prohibitive since it requires doing systematic splitting to obtain *bin*s that are much more refined than necessary. Although these two steps can be performed in polynomial time (so, presumably less than polynomial *space*), they remain practically unfeasible.

The technique developed here presents the advantage that only blocks which naturally arise in the actual computation are considered. This is the basic philosophy underlying the issue of when to split a block of tuples.

On the other hand, it is easy to show that the technique proposed here at the limit performs as well as the original technique. Thus the theoretical bounds proved in [ACV92] apply here as well.

3.2 Practice

A prototype has been developed in Prolog and several examples run on it. The obvious problem with the technique is that it performs well only when the number of equivalence blocks remains small. The prototype turned out to be quite useful in choosing the split rules that are described in the paper to restrict the number of splits that are generated. Indeed, in a first version of the prototype, annotations (the $\subset$ superscripts to denote "some subset of") were not used in transition tables and blocks were split to obtain exact equations. This resulted in severe efficiency problems. (Compared measures are given below.)

For other programs, our technique yields unnecessary computations. Further experimentation may lead to refining these rules. Some of the directions are discussed in the next section. However, it is important to remember that splitting can cost a lot of overhead and, when in doubt, we believe that it is preferable to refrain from splitting.

Also, the implementation is rather naive and inefficient; for instance, *bin* information is kept as lists. Clearly, a real-world implementation should provide sophisticated access structures for the *bins*, the *val* function and the transition tables. As a consequence, we only ran the programs on "small" graphs (of less than 1000 edges).

The main goals that were pursued were to verify empirically that the number of blocks remains tractable and to check that redundant work is often avoided.

We have tried for instance the following programs:

1. transitive closure,
2. well-founded vertices (see Example 2), and
3. insert-delete (see Example 3 below),

The first one is monotone and would yield the same saving with a seminaive bottom-up technique [BR86], as discussed in Section 3.4. The second one is not. (Although one can code well-founded vertices in a monotone way for finite graphs, that was not the purpose of the exercise.) A very nonmonotone example was also tried:

Example 3. (Insert-Delete) The rules are the following.

$$\begin{aligned}\alpha_1 &: +res(X,Y) \leftarrow t(X,Y).\\ \alpha_2 &: +res(X,Y) \leftarrow t(X,Z), res(Z,Y).\\ \alpha_3 &: -res(X,Y) \leftarrow t(X,Z), res(Y,Z).\end{aligned}$$

The control that we used is $fire[\alpha_1], fixpt[\alpha_2, \alpha_3]$. However, a fixpoint may not occur for some input relations t. □

We have tested those programs on several kinds of directed graphs:

1. full binary trees,
2. random graphs with various edge probabilities,
3. full binary trees with random distortion, i.e., binary trees with new edges removed or added to the graph with small probabilities.

Some measures obtained with the prototype are given in Figure 2. We give here the number of refinements in an execution since this seems to be the crucial parameter in the overall performance of the technique.

It should be noted that in most cases, the number of refinements performed by the programs were very small compared to the number of tuples generated by the computation; this clearly highlights the potentiality of the technique. A notable exception is the straightline graph, where many blocks at the end of computations consist of a single tuple. As expected, the split technique performs very badly with such data.

There were in most examples large variations in the sizes of the blocks. Therefore, more refined measures of the improvement should be considered.

	full binary tree 511 vertices	random graph 500 vertices edge prob. 0.001	random graph 500 vertices edge prob. 0.002	straight line graph 500 vertices
transitive closure	9	6	23	501
transitive closure (no $\subset$ annotation)	9	10	734	501
well founded	22	5	72	1004
insert-delete	2	2	45 + loop	2

Fig. 2. Number of refinements in examples

The case of full binary trees studied in [Lin90] and also [AV91b] clearly demonstrates the usefulness of the technique for very regular data structures, i.e., structures presenting a lot of automorphism. The number of blocks is an order of magnitude smaller than the number of tuples (depth of the tree vs. number of vertices). This was verified in our implementation.

The random cases according to [ACV92] should also behave "properly". This was also verified although for a particular threshold (edge probability), the number of splits could be quite large (see edge probability 0.002 in Figure 2) .

The case of full binary tree with distortion is not given in the figure. It behaves almost like full binary trees if the distortion is small, and tends to the behavior of random graphs when it is large.

For the insert-delete program, the interaction of the inserting and deleting rules is crucial. After making 45 refinement on one graph, the control continued to loop although no further relational computations were performed. Existing bins were cycled between current and noncurrent by the actions of rules α_2 and α_3. At least , this was an *efficient* infinite loop!

The table in Figure 2 also allows us to measure the importance of the annotation.

It should be observed that the experiments confirmed the theoretical claim that the number of refinements is "in general" small. Of course, experiments on "real" databases should be performed to further substantiate the claim. We briefly discuss below fixes when particular data violate this optimistic assumption.

3.3 Implementation Issues

The technique is intended to limit computations with actual data and replace them as much as possible with computations with *bin*s. Of course, it is crucial to optimize the *bin* computations.

We can distinguish various sources of optimization for the *bin* computations:

1. The first one is that *bin* information should be small compared to the database size. Thus *bin* information should be maintained in main memory as much as possible.
2. Bin comparisons as discussed above can be done very efficiently using bits shifting. Bin manipulation in general is made easier by the *bin* naming scheme described in the paper. In particular, it is not necessary to store the *val* for each *bin* but only of the most refined one if the *val* access structure is organized as a binary tree.
3. The *bin* tables are also heavily used and appropriate access structures such as indices should be provided for them.
4. One can verify that the computations carried with the *bin*s are mostly standard relational operations. Thus, the *bin* information and notably the transition tables can be stored in the database. (The set of *bins* is then one of the ADTs handled by the relational system.) The *bin* part of the computation can then benefit from the optimization provided by the relational system.
5. The overhead coming from splitting the blocks associated to *bin*s (in † of Algorithm 2.5), is limited. Observe that the normal computation would have to insert the tuples in $\alpha[val(r_i), val(s_j)]$ into relation T. Inserting them in turn in each block t_δ and constructing at the same time $t_{\delta 0}$ and $t_{\delta 1}$ is only marginally more expensive. This can even be improved using parallel computation.

3.4 Comparison with the Differential Technique

Differential techniques have been used in databases in a variety of contexts: e.g., recursive query processing (seminaive evaluation), view maintenance, integrity or trigger maintenance. The intuition is as follows.

Suppose that we apply a rule:

$$s(X,Y) \leftarrow s(X,Z), t(Z,Y)$$

When we derive some new s-tuples, it is not necessary to recompute the join. Instead, it suffices to consider the rule

$$s(X,Y) \leftarrow \Delta(X,Z), t(Z,Y)$$

where Δ is the new s-tuples that have been derived.

This is clearly closely related to our technique.

The differential technique applies only to monotonely increasing relations. No such restriction is made to the split technique developed here. It is however interesting to compare the performances of the two techniques on programs handled by both, i.e., on programs allowing only addition of tuples.

First, for the case of linear recursions, our technique computes exactly the same joins as seminaive evaluation. Each join's operands are (i) the most recent stages' new tuples and (ii) the original base relation(s).

In the general case (of additive programs), there are variations in how differencing is done, but this is expected to be less costly as the relations involved should be already sorted (to remove duplicates, e.g.) beforehand. Our technique needs to "subtract" each existing segment, which holds the tuples of an earlier stage, from the tuples just produced by the join. Seminaive evaluation combines all previous stages into one relation; this relation is "subtracted" from the new join tuples.

After many stages have been computed, our technique subtracts a number of small relations, while seminaive subtracts one large relation, from the new join tuples. With good implementations, the difference in performance should be minor; but we believe it will favor seminaive evaluation when the number of bins becomes large. The prototype could not provide such a fine analysis and it is left for future research.

Finally, it should be noted that when one knows that a predicate is monotonely used, the algorithm can be slightly improved by essentially mimicking seminaive using a transition table. The only overhead then would be the look-ups in the transition table which are still necessary.

To summarize the comparison with seminaive evaluation, we will say that the split technique can be viewed in some sense as a generalization of the seminaive technique to a larger context (nonmonotone updates). The generalization is obtained at the cost of some limited overhead in the monotone case.

4 Discussion

In this section, we briefly address implementation issues, discuss advantages, disadvantages and possible improvements.

4.1 Advantages

This optimization technique presents several advantages:

1. *capturing seminaive evaluation*
   We believe that the seminaive evaluation although rather straightforward is the major source of optimization proposed so far. Other techniques like magic-set [BMSU86] or Query/SubQuery [Vie87] are primarily intended for query processing and it is not clear that this is the most fundamental place where optimization is needed. However, seminaive works only with monotone computations whereas our technique is applicable in a larger context. Furthermore, we believe that a brute-force use of the seminaive technique would result in a lot of overhead in relations to maintain. Also, it may be unclear in a particular application whether a relation is used monotonically or not. The seminaive optimization can still be obtained using our technique in such contexts.
2. *triggers, integrity constraints, etc.*
   seminaive evaluation is just one occurrence of the use of a larger technique, *differentiation*, that can also be applied for integrity or trigger maintenance. The split technique can also be useful in these contexts.
3. *preprocessing*
   If one knows the rules that are mostly used by future programs/queries, one can use the technique to preprocess based on these rules. Preprocessing would consist in splitting the relations based on the frequently used rules ahead of time.
4. *loop detection*
   The languages considered for active databases like partial fixpoint logic present

the drawback that termination may not been guaranteed (which is not surprising with inserts, deletes and iterations). Loop detection may be quite expensive since it involves the comparison of database states. With the split technique, the database is represented with *bins*. When the number of *bins* is small, *bin*-based loop detection may be much more tractable than data-based loop detection.

5. *adaptability*
   When a user or an application works with a database, it is in a specific context. It is likely that a result of certain rules that are fired will be needed again later on. With the split technique, the tuples are clustered in such a way that the database will almost magically adapt to the specific needs.
6. *clean architecture*
   A prime advantage of the technique is that it is separating the set oriented computation, namely the conjunctive query rules, that can be heavily optimized (notably with parallelism) from the remaining of the computation.

Another known optimization technique, magic sets, also addresses the issue of avoiding redundant work by focusing on relevant data (using constant propagation). This is a somewhat orthogonal issue. However, the two techniques are not incompatible. The "magic" predicates can be used with the split technique. Indeed, magic predicates are probably prime candidates for being split.

The split technique may also have a crucial impact in the domain of clustering. Clustering strategies are fundamental to achieve better performances in databases. However, little is known in this areas. The clustering techniques that are being used are still entirely adhoc. The experimentations done with the split techniques tend to show that, as expected, tuples with similar logical properties are clustered in the blocks of partitions. These logical clusters are thus good candidates for physical clustering. Also, the transition tables give the natural links between blocks of tuples. This may be useful information for guiding physical clustering. This direction remains largely unexplored.

4.2 Drawbacks

The technique also presents serious drawbacks. The main one is that for some data distributions, the number of blocks may grow rather fast to reach the number of tuples in the worst case. This results in a decrease of performances. The situation can be detected to refuse to split in such cases. One may for instance use lower bounds under which a block is not split any further.

4.3 Possible Improvements

Bin Sharing In the present paper, we use a partition for each updatable relation. However, when copying from one relation (possibly with selection) to another occurs frequently, this results in the maintenance of identical blocks of tuples and unnecessary work. One can imagine that the two relations share the same world of blocks of tuples. This may result if one is not careful in unnecessary splits so it has to be done only when explicit copying is done and one knows that such operations often occur.

Argument Splitting In the technique that we described only blocks in the goal relation of a rule are split. It is often useful to also split the premisses relations as illustrated in the following example.

Consider the two rules:

$$\begin{array}{l} +s(X,Y) \leftarrow r(X,Z), t(Z,Y). \\ -t(X,Y) \leftarrow \quad\quad\quad \ldots \end{array}$$

Suppose that r is a relation that is never updated and initially consists of a block of tuples denoted by the *bin* r_2. Now suppose that t consists initially of a block t_2. The first rule is activated and the join of $val(r_2)$ and $val(t_2)$ is computed. Suppose that the second rule is activated and removes some tuples from $val(t_2)$. Next time the first rule is activated we will have to recompute the join of $val(r_2)$ and a new block in t.

An alternative is as follows. While computing the join of $val(r_2)$ and $val(t_2)$, we could split r_2 into r_4, denoting the set of tuples that joins with $val(t_2)$; and r_5, denoting the set of those which do not. Now, the second rule removes tuples from t and splits t_2 into t_4 which is current and t_5 which is not. When the first rule is applied again, we do have to compute $\alpha[val(r_4), val(t_4)]$ but since we find the tuple

$$[r_5, t_2, \phi]$$

in the transition table and since 2 is a prefix of 4 in binary (so $val(t_4) \subseteq val(t_2)$), we can conclude that $\alpha[val(r_5), val(t_4)]$ is empty which saves (a possibly large) computation.

Observe that at the same time the split of t seems to be unnecessary and if all splits are performed for all arguments, the number of *bin*s may soon become too large. Thus remains the most important issue:

($\star$) WHEN SHOULD ONE SPLIT?

Using Knowledge of the Program Control To answer ($\star$), the alternatives are as follows:

1. split systematically at the risk of blowing up the number of *bin*s and making the overhead of *bin* handling prohibitive;
2. do minimum splittings as done in the algorithm above at the cost of having to redo some computations.

One basic assumption in the paper is that we had no a-priori knowledge of the order of application of the rules neither of their frequency. If such knowledge is accessible, one can improve the technique by more carefully selecting the splits:

1. when the monotonicity of some relations is guaranteed, we believe that a careful case analysis would allow to choose the best strategy. The technique can then be viewed as a careful and space efficient implementation of the differential technique.
2. when this is not the case, the analysis is much more intricate. This probably remains a most important open question.

In the same spirit, observe also that one may want to analyze the result of a sequence of operations (that occurs frequently, for instance in an iterative construct) by analyzing its global effect, instead of inspecting the effects of rules in isolation. This may allow besides determining what are the necessary splits and the unnecessary ones, optimizing the computation of that sequence by for instance, avoiding to compute sets of tuples that are deleted later on.

Parallelism The blocks corresponding to bins are disjoint. This makes them very appropriate for parallel computation. In particular, the technique replaces a possibly large join or difference by many (often) small joins and differences that are independent. In the case of join rules, the results of these operations impact independently on the various blocks of the target relation, and this also can benefit from parallelism. In the case of difference, this is not so since a tuple is considered for updating the target relation if it is in the *intersection* of the partial results.

Acknowledgements

Some of the ideas developed in this paper are natural consequences of a paper by the first author and Victor Vianu. Sophie Cluet and Eric Simon are thanked for discussions on this topic.

References

[ACV92] S. Abiteboul, K. Compton, and V. Vianu. Queries are easier than you thought (probably). In *Proc. ACM Symp. on Principles of Database Systems*, 1992.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[AV91a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[AV91b] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.

[BC79] O.P. Buneman and G.K. Clemons. Efficiently monitoring relational databases. *ACM Trans. on Database Systems*, 4(3):368–382, September 1979.

[BFKM85] L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison Wesley, 1985.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. ACM Symp. on Principles of Database Systems*, pages 1–15, 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 16–52, 1986.

[dMS88] C. de Maindreville and E. Simon. Modelling non-deterministic queries and updates in deductive databases. In *Proc. of Internat. Conf. on Very Large Databases*, Los Angeles, 1988.

[Esw76] K.P. Eswaran. Aspects of a trigger subsystem in an integrated data base system. In *Proceedings of the 2nd International Conference in Software Engineering, San Francisco, California*, pages 243–250, 1976.

[Han89] E.H. Hanson. An initial report on the design of ariel: a dbms with an integrated production rule system. In *Sigmod Record, 18(3)*, pages 12–19, 1989.

[HJ91] R. Hull and D. Jacob. Language constructs for programming deductive databases. In *vldb*, 1991.

[IN88] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proc. ACM Symp. on Principles of Database Systems*, pages 103–116, 1988.

[Lin90] S. Lindell. *An analysis of fixed-point queries on binary trees*. PhD thesis, UCLA, 1990.

[MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.

[MW88] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, 1988.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. ACM Symp. on Principles of Database Systems*, page ??, 1988.

[PY92] C. H. Papadimitriou and M. Yannakakis. Tie-Breaking Semantics and Structural Totality. In Eleventh *ACM Symposium on Principles of Database Systems*, pages 16–22, 1992.

[SZ90] D. Saccà and C. Zaniolo. Partial models, stable models and non-determinism in logic programs with negation. Technical report, MCC, Austin, TX, January 1990. (Extended abstract appeared in Ninth ACM Symposium on Principles of Database Systems, 1990.).

[SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES rules system. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[VG92] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 1992. (to appear). Abstract in PODS, 1989.

[Vie87] L. Vieille. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69(1):1–53, 1987.

[Wid91] J. Widom. Deduction in the Starburst production rule system. Technical report, IBM Almaden Research, 1991.

This article was processed using the LaTeX macro package with LLNCS style