INCREMENTAL EVALUATION OF RULES AND ITS RELATIONSHIP TO PARALLELISM

Ouri Wolfson
Hasanat Dewan
Salvatore Stolfo
Yechiam Yemini

Columbia University
Department of Computer Science
Technical Report CUCS-058-90

Incremental Evaluation of Rules and its Relationship to Parallelism

PRELIMINARY VERSION

Ouri Wolfson
Hasanat Dewan
Salvatore Stolfo
Yechiam Yemini

Distributed Computing and Communications Laboratory
450 Computer Science Building
Columbia University
New York, NY 10027

December 4, 1990

Abstract

Rule interpreters usually start with an initial database and perform the inference procedure in cycles, ending with a final database. In a real time environment it is possible to receive updates to the initial database after the inference procedure has started or even after it has ended. We present an algorithm for incremental maintenance of the deductive database in the presence of such updates. Interestingly, the same algorithm is useful for parallel and distributed rule processing in the following sense. When the processors evaluating a program operate asynchronously, then they may have different *views* of the database. The incremental maintenance procedure we present can be used to synchronize these views.

1 Introduction

Traditional rule based systems are composed of a knowledge base which consists of a set of facts, the database, and a set of rules that operate on them. Ordinarily, the database changes only as a result of inference activity being carried out by a *rule interpreter*, that evaluates rules. Such systems assume a *static* environment, in the sense that changes to the database take place only as a result of the rule-program evaluation. These systems are not capable of efficiently incorporating modifications to the database of facts resulting from sources other than the actions of the rules themselves.

In real time applications, such as communication network management, the database may change independently of the inference process, since usually one cannot afford the luxury of

collecting all the relevant facts before starting the inference procedure. Relevant information may arrive after inferencing has already begun. Consider, for example, an expert analyzer to automatically detect a fault in a communication network that is in operation, and is continuously producing information relevant to the detection process (the fault may be very subtle, such as the software bug that recently disabled the long distance telephone network for several hours). In this environment, messages may arrive from time to time that nullify previous assumptions. In our example, suppose the expert analyzer has assumed that some link *LINK A_B* is up when the diagnosis of some problem was started, and at some later point, say at iteration n of the standard *match-select-act* cycle, a message arrives indicating that *LINK A_B* is down. The naive approach for incorporating this late arriving information would be to restart the expert analysis from the beginning. However, considering that network management data is being produced continuously, this strategy may result in an infinite regression and the inference process may never terminate. It is essential in these situations to have an inference mechanism that performs only the actions that are necessary to *incrementally* bring the database to a consistent state.

Our approach for incrementally updating a database can be summarized as follows. In the context of the previous example, we would *undo* only the consequences of the initial asumption that *LINK A_B* was up, and *redo* the consequences of the newly received fact that *LINK A_B* is down. In other words, we revise only the consequences associated with the newly arrived information.

Closely tied to the question of incremental updates to a database is the semantics of the underlying rule language. For Datalog [1] programs, where negation is not allowed in either the head or body of a rule, incremental processing is simple. Consider for example the semi-naive evaluation [2] of the linear Datalog program that computes the transitive closure of a graph. Incorporation of a newly arrived arc of the graph is easy. The arc is simply added to the differential, and the evaluation proceeds as usual, until a fixpoint is reached. This can be done regardless of whether the new arc arrived before or after the transitive closure evaluation has completed. Since Datalog is strictly monotonic, incremental update involves only *redoing*, or recomputing the consequences of the newly arrived arc.

On the other hand, retraction of an arc from the graph, even in the simple transitive closure example, is a more subtle non-monotonic process. Consider languages that allow negation in

the body and head of rules (negation in the head means deletion). Suppose that the fact f is retracted from a database after some inferences that depend on f have occurred. The high level view of subsequent processing that must take place is the following: the facts that were derived from f (and recursively, their derivatives) must be backed out or *undone*, and then the facts that can be derived from $\neg f$ (and their derivatives) must be *redone*. For Datalog programs (that do not have negative atoms in the rules), no actions are derived from negative facts. Therefore, for a retracted fact we only need an undo phase, and for an asserted fact we only need a redo phase. However, for more powerful rule languages, we need both an undo and a redo phase to *incrementally* update the database.

These two phases are not always straightforward. Rule languages such as OPS5 and Datalog$^{\neg *}$ [3] usually operate in *match-select-act* cycles, and at each cycle (or *iteration*), some conflict resolution strategy is applied. Consider for example redoing the consequences of f. This involves matching all the rules (or some subset of them in a language such as stratified datalog) and considering the instantiations that have f in the body. Suppose that the head of such an instantiation is add_e. Can we simply add e to the database during the redo phase? The answer is **no** for the following reason. The incremental algorithm must produce the same result as if f were in the initial database. Suppose that f were in the initial database. It is possible that the add_e operation would have been eliminated by the conflict resolution strategy. In the language Datalog$^{\neg *}$, add_e would have been cancelled out by a $delete\_e$ operation in the same cycle; moreover, the $delete\_e$ operation may not be a consequence of f. Similarly, in OPS5 add_e would not have been executed, had there been a more recent instantiation in the cycle. Therefore, the incremental algorithm must have enough information to determine for each redo instantiation whether or not it would have been eliminated by the conflict resolution strategy. Furthermore, this information obviously depends on the conflict resolution strategy, which in turn depends on the rule language.

In this paper we devise the incremental update algorithm for the language Datalog$^{\neg *}$. The reason for choosing this language is that it has several attractive characteristics. First, it is set-oriented, a typical feature of database rule languages. Second, it is sufficiently rich, allowing negation in the body and in the head of rules. Third, it is abstract enough to enable identifying principal concepts that relate to incremental evaluation of rules, independently of the idiosyncrasies of commercial languages. We will show that the incremental update algorithm

can also be applied to stratified datalog [14].

Now consider the data reduction paradigm for distributed and parallel rule processing [4, 6, 15, 13]. It stipulates that each processor evaluates the original rule program, but with less data. The operations of each processor that result from the evaluation (e.g. *add_a_fact* or *delete_a_fact*) are transmitted to other processors that may use these to derive other operations. Now, if the processors are synchronized at each cycle, meaning that they all complete their evaluation and transmission in a certain cycle before any processor proceeds to the next one, then incremental evaluation is not necessary. However, this implies that the evaluation proceeds at the speed of the slowest processor at each iteration, which is not acceptable in a distributed setting. Suppose therefore that the evaluation proceeds asynchronously. Then it is possible that a processor p_1 that is evaluating at cycle 5 receives an *add_e* from a processor p_2 that is evaluating at cycle 3. This means that p_1 has to incorporate e into the database of cycle 3. Then the problem it faces is exactly the problem of incremental rule processing: it has to undo the consequences of $\neg e$ at cycle 3, and redo the consequences of e at cycle 3, but without backing its whole rule evaluation to cycle 3. In other words, individual processors need to have the ability to incorporate changes in arbitrary previous iterations. In contrast, for real time rule processing the update is always to the input database, namely to cycle 0. However, our algorithm for incremental update is recursive and works for an arbitrary cycle update, and therefore, as presented, it can be applied to asynchronous parallel-and-distributed rule processing by data reduction.

Our work is related to previous work in truth maintenance systems [7, 8, 9] in the AI literature. These systems are also designed for incremental evaluation; however truth maintenance systems build a dependency graph with nodes that correspond to base and derived facts, and also to instantiations of inference rules. In data intensive applications, maintaining such a graph can become prohibitively expensive. Arcs join antecedents and consequents to nodes representing inference rules. In our algorithm for incremental update, rule instantiations do not appear in the maintenance data structures. We explicitly recompute, via lightweight computations, only those instances needed to redo or undo a prior inference chain, rather than storing all prior instances ever computed. We enable the incremental processing by attaching to each database fact a chain that represents the status (*in* or *out*) of the fact at each iteration. On the other hand, truth maintenance systems are more flexible, enabling incremental changes to the rules as well as the facts.

Our work is also related to the materialized-view-maintenance research and snapshot refresh algorithms in databases [11, 12, 16]. However, those works are concerned with incremental processing of relational expressions. They do not address the problem of inferencing in cycles, with a conflict resolution step at each cycle. In contrast, these are the issues on which we concentrate in this paper. In fact, the aforementioned research can be incorporated in the incremental update algorithm, as we shall point out in Section 4.

The rest of this paper is organized as follows. In Section 2, we discuss the language Datalog$^{\neg *}$ and in Section 3 we present the $INCR\_UPDATE$ algorithm. In Section 4, we discuss the data structures that enable $INCR\_UPDATE$ to reconstruct the inference database at each cycle. In Section 5, we point out how $INCR\_UPDATE$ can be applied to stratified Datalog. In Section 6, we discuss the synchronized data reduction paradigm for Datalog$^{\neg *}$, and in Section 7 we show how the $INCR\_UPDATE$ procedure can be used to remove the need for synchronization among the processors. In Section 8, we conclude.

2 The language Datalog$^{\neg *}$

In this section we introduce our basic terminology. Intuitively, rules in Datalog$^{\neg *}$ have the general form:

$$\pm h(\ldots) :- \quad \pm b_1(\ldots), \ldots, \pm b_k(\ldots), \ldots\ldots \pm b_n(\ldots)$$

A positive head implies the corresponding fact should be added to the database on rule firing, whereas a negative head implies the corresponding fact should be removed from the database, if it exists.

Formally, Datalog$^{\neg *}$ programs are built from *atoms*, which are predicate symbols followed by a list of arguments. The arguments may be either variables or constants. For simplicity, constants are natural numbers. A *literal* is either an atom, also called a *positive literal*, or a negated atom, also called a *negative literal*. If all arguments are constants, we call the literal a *fact*. A *rule* consists of a literal, the *head* of the rule, and possibly a conjunction of positive and negative literals which form the *body* of the rule. We use the usual notation for writing programs. Variables are denoted by capital letters, and predicate names are strings built from lower case letters. An example of a rule is

$$a(X, Y) :- \quad a(X, Z), p(Z, Y), \neg q(Z, 2)$$

where the head of the rule is always to the left of :-, and the body to the right. If Q is a set of facts, then $\neg Q$ denotes the set having the same literals as Q, but with their sign reversed.

A *program* is a set of rules. A program is called *safe* if each rule fulfills the following conditions: (1) Each variable in the head of the rule also occurs in the body of the rule, and (2) Each variable in a negative literal in the body also occurs in a positive literal of the body. We require that programs be safe.

The *instantiation* of a rule r is defined with respect to a *database*, i.e. a finite set of positive facts; it is an assignment of constants to the variables in r such that all the positive facts in the body are in the database, and the negative ones are not.

The operational semantics of the evaluation are as follows. The *input* to a program is a database. The *output* of a program P for an input I, denoted $O(P, I)$, is the database obtained at the end of the following iterative procedure.

procedure $DATALOG^{\neg *}\_SEMANTICS$

1. Start with the database consisting of the input. Initialize the evaluation iteration count $IC \leftarrow 0$.

2. (**Match**) Determine S, the set of operations each of which is the head of an instantiation with respect to the current database. If this set is empty, stop.

3. (**Select**) Let S' be the subset of S consisting of all the operations whose negation is not in S (i.e. if f and $\neg f$ are both in S then neither operation is in S').

4. (**Act**) Execute the operations in S'. Increment IC by 1. Return to 2.

end {*procedure* $DATALOG^{\neg *}\_SEMANTICS$ }

Notice that the **Select** step indicates that if a fact f is added and deleted at the same iteration, regardless of the number of occurrences of these operations, the status of f (*In* or *Out* of the database) remains as it was in the previous iteration.

3 Outline of the $INCR\_UPDATE$ Algorithm

Basically, the inference consists of two phases: DO and $INCR\_UPDATE$. The DO phase consists of the procedure $DATALOG^{\neg *}\_SEMANTICS$ of the previous section. After the **Act**

step, there is a check to determine whether an *interrupt* message has been received. If so, the $INCR\_UPDATE$ procedure is invoked. The incremental update stage consists of an $UNDO$ phase and a $REDO$ phase. The *interrupt message* consists of a set Q of positive and negative literals, denoting additions and deletions respectively, to be applied to the input database.

Consider the DO phase. With each fact f that was in the database at some iteration, we associate a *fact chain* that enables us to determine the status (*in* or *out* of the database) of f at each iteration. The structure of fact chains will be discussed in Section 4.

Suppose that the positive fact f is in Q, and f was not in the input database. Then we first execute the *match* step of the inference procedure, considering only instantiations with respect to the input database in which $\neg f$ appears in the body of the instantiated rule. This is the $UNDO$ phase. Let U be the set of operations in the heads of these instantiations. Each operation in U will have to be *undone*. Second, we execute the *match* step of the inference procedure, considering only instantiations with respect to the input database, in which f appears in the body of the instantiated rule. This is the REDO phase. Let R be the set of operations in the heads of these instantiations. Each operation in R will have to be *redone*.

Now suppose that $+\epsilon$ (or simply, ϵ) is in R, indicating that the fact ϵ would have been added to the database in the first cycle, had f been in the input database. Does it necessarily mean that the incremental procedure should repeat the redo phase with ϵ? The answer is **no**. It is possible that in the first cycle ϵ was added to the database not only as a result of the instantiation that has f in the body, but also as a result of another instantiation. Then ϵ is not *new* in the second iteration, and it should be eliminated from the incremental evaluation of the subsequent cycles. In other words, based on the U and R sets we should determine which facts are *new*. Only they should be carried forward, to the next cycle in the incremental evaluation.

Generally speaking, given $UNDO$ and $REDO$ sets, the incremental procedure, called $INCR\_UPDATE$, determines which facts must be viewed as having changed their status as a result of the incremental procedure at the specified iteration. The set of facts that are determined to have changed their status is called the NEW set. This set is the focus of interest of procedure $INCR\_UPDATE$. The procedure recursively redoes all the inferences that descend from the NEW set, and it also recursively undoes the inferences that stem from $\neg NEW$. In other words, $UNDO$ and $REDO$ sets are generated for successive iterations, given the initial ones. Formally, procedure $INCR\_UPDATE$ is as follows.

recursive procedure $INCR\_UPDATE(U, R, i)$

Comment: This procedure is invoked by the inference system, as an exception handler on receiving incremental modifications to past assumptions made on the database. The procedure modifies the database, given two sets of operations, U and R, that have to be incorporated at some past evaluation iteration i. Each set contains positive and negative facts. The set U represents a collection of facts whose addition or deletion (depending on the sign) has to be undone at iteration i. It is called the **undo** *set. The set R represents a collection of facts whose addition or deletion has to be incorporated at iteration i. It is called the* **redo** *set.*

1. η — Current Evaluation Iteration Number in the *DO* Phase.

2. If $(i = \eta)$ RETURN to the *DO* Phase.

3. For each $f_u \in U$ call $UNDO\_MAINTAIN\_CHAIN(f_u, i)$

Comment: undoing an operation amounts to the manipulation of the corresponding fact chain

4. For each $f_r \in R$ call $REDO\_MAINTAIN\_CHAIN(f_r, i)$

Comment: redoing an operation amounts to the manipulation of the corresponding fact chain

5. NEW — $COMPUTE\_NEW(i)$

Comment: the NEW set is computed by examining the fact chains

6. If $(NEW = \emptyset)$ RETURN to the *DO* Phase.

7. Find all rule instantiations with respect to the database at iteration i, such that a fact $f_r \in NEW$ is in the body. Denote by F_r the set of facts in the heads of these instantiations. This will serve as the **redo** set for the recursive call.

8. Find all rule instantiations with respect to the database at iteration i, such that a fact $f_u \in \neg NEW$ is in the body. Denote by F_u the set of facts in the heads of these instantiations. This will serve as the **undo** set for the recursive call.

9. Call $INCR\_UPDATE(F_u, F_r, i+1)$.

end {*procedure INCR_UPDATE*}

The initial invocation is $INCR\_UPDATE(\emptyset, Q, 0)$, where Q is the set of facts in the interrupt message. We point out that the work on incremental evaluation [11] can be useful for speeding up the execution of steps (7) and (8) of procedure $INCR\_UPDATE$.

The computations in steps (7) and (8) of the above recursive procedure is clearly less expensive, in general, than computing all instantiations of rules at iteration i.

4 Fact Chains

As mentioned earlier, for each fact f there is a chain $\mathcal{C}(f)$ that enables the system to determine the status of f at each iteration. For each iteration in which f was added or deleted (either in the *DO* or the *REDO* phases) we keep a count of the number of times it was added, and the number of times it was deleted. In other words, for each iteration we keep a *state record* in $\mathcal{C}(f)$, which has the following structure:

State Record	
Field	*Type*
Added	Integer
Deleted	Integer
Iter	Integer
Status	Binary

The *Iter* field indicates which iteration a particular state record corresponds to, and the *Status* field holds the status of the fact (*In* or *Out* of the database) at that iteration. The *Added* and *Deleted* fields are necessary to count the number of occurrences of additions and deletions

(even though the operational semantics do not require so), since one does not know how many of these operations will be undone in the future.

4.1 Initialization and Maintenance

Each fact present in the input database is called an *initial fact*. We set up fact chains for the initial facts at startup time. All such facts have fact chains initialized with a single state record s, with $s.Iter$ set to 0, and $s.Status$ set to In. Additional chains are established for those facts which are *added* or *deleted* during inference, and for which no chain was established initially. Chains established during inference consist of a state record at iteration 0 with a status of Out, and a state record for iteration t, where t is the iteration when f was added or deleted during the inference.

Maintenance operations on chains occur in all three phases of incremental evaluation. *DO* and *REDO* phases increment the count in the *Added* or *Deleted* fields of the state record, depending on whether the fact in question is added or deleted as a result of rule firing. In contrast, *UNDO* decrements the appropriate field. The procedure that manipulates the chains during *DO* and *REDO* phases is *REDO_MAINTAIN_CHAIN*. Procedure *UNDO_MAINTAIN_CHAIN* manipulates chains during the *UNDO* phase. *REDO_MAINTAIN_CHAIN* receives a set of facts and an iteration number, and increments the counts of the appropriate fields of the state record indicated by the iteration number, whereas *UNDO_MAINTAIN_CHAIN* decrements the counts. As an example, consider the effect of the procedure call

$$UNDO\_MAINTAIN\_CHAIN(\{f_1, \neg f_2\}, i).$$

Suppose $C(f_1)$ has a state record u with $u.Iter{=}i$, $u.Added = 1$ and $u.Deleted = 1$, and $C(f_2)$ has a state record v with $v.Iter{=}i$, $v.Added = 0$ and $v.Deleted = 1$. After the procedure executes, we will have $u.Added = 0, u.Deleted = 1$, $v.Added = 0, v.Deleted = 0$. Subsequent to this, if we make the call

$$REDO\_MAINTAIN\_CHAIN(\{\neg f_1, f_2\}, i)$$

we will end up with $u.Added = 0, u.Deleted = 2$, $v.Added = 1, v.Deleted = 0$.

4.2 Computing the NEW Set

The status of a fact f for any iteration is determined by examining the *Added* and *Deleted* fields in the state record for that iteration. This applies to the *DO* phase and to the incremental update phase as well. In procedure $INCR\_UPDATE$, the final value of the status of a fact f and the set of *new* facts at an iteration is computed with the $COMPUTE\_NEW$ function.

$COMPUTE\_NEW$ takes an evaluation iteration number i as argument, and scans the chains for the facts that were changed during the $UNDO$ or $REDO$ phases. If both the *Added* and *Deleted* fields are zero in the state record for i, or if both are non-zero, then we revert to the status at iteration $i-1$. Otherwise, if the *Deleted* field is 0, we interpret that as indicating that the fact was only added, and the status is set to In, whereas if the *Added* field is 0, the status is set to Out. A fact is flagged *new* if its newly computed status is the inverse of its former status. We present the outline of procedures $INITIALIZE\_CHAINS$, $UNDO\_MAINTAIN\_CHAIN$, $REDO\_MAINTAIN\_CHAIN$, and $COMPUTE\_NEW$ in appendix A.

4.3 Correctness

In order to demonstrate that procedure $INCR\_UPDATE$ works correctly, we need to show that if an incremental update message to incorporate a positive or negative fact f in the input database is received while the processor is at iteration $t > 0$, then the final database produced would be the same as in the case where f is in the input database. The proof is by induction on t; we omit the details from this preliminary version.

5 Applicability of $INCR\_UPDATE$ to Stratified Datalog

For stratified Datalog [2], the rule program interpreter evaluates the rules one stratum at a time. Iteration numbers are then replaced by the pair $(Stratum\_Number, Iter\_Number)$, which may be regarded as our new iteration counter. The implication of this adaptation is the following. We have to adopt a modified numbering scheme for chain maintenance operations, and for matching rules in steps (7) and (8) of procedure $INCR\_UPDATE$. First, instead of ordering the fact chains by iteration number, we now use the order imposed by lexicographic ordering of the $(Stratum\_Number, Iter\_Number)$ pairs. Therefore, in the incremental update phase,

instead of referring to iteration $i-1$, we refer to the previous iteration in the lexicographic order. Second, in steps (7) and (8) of procedure $INCR\_UPDATE$, we restrict the matches to rules in the particular stratum being considered. With these simple modifications, $INCR\_UPDATE$ is applicable to stratified Datalog.

6 Distributed Synchronous Evaluation of Datalog$^{\neg *}$ Programs

The *data reduction* paradigm has been introduced in [6] for Datalog (without negation). Intuitively, instantiations of the rules in a given program are partitioned among a set of processors. The original program is evaluated on each processor, but with less data. The partition of the instantiations is achieved by restricted versions of the original program, such that each restricted version is evaluated at one processor. In general, it is necessary for the processors to communicate intermediate results to each other by message passing. In this section, we extend the paradigm to Datalog$^{\neg *}$. In particular, we discuss a synchronous variation of this extension, whereas in the next section, we discuss an asynchronous one.

Formally, let $P = \{r_1, ..., r_m\}$ be a program with m rules and $\{p_0, ..., p_{k-1}\}$ be a set of $k > 1$ processors. For each rule r_i, we designate k restricting predicates, $h_{ij}(X_1, ..., X_{q_i})$, for $0 \leq j \leq k-1$. The arguments $X_1, ..., X_{q_i}$ are the same for all the k predicates, and by definition, all the arguments are variables of r_i. We require that for each instantiation of the variables $X_1, ..., X_{q_i}$, the predicate h_{ij} is *true* for exactly one j. Denote by r_{ij} the restricted version of the rule r_i having the restricting predicate $h_{ij}(X_1, ..., X_{q_i})$ appended to its body. Denote by P_j the restricted version of P consisting of the set of rules $\{r_{ij} | 1 \leq i \leq m\}$. The set $\{P_0, ..., P_{k-1}\}$ is called a *data-reduction parallelization strategy* for P.

The set of processors $\{p_0, ..., p_{k-1}\}$ cooperate in evaluating P in parallel as follows. The database is partitioned among the processors such that each tuple has a unique *Data Handler* processor (DH) at which it resides. The DH processor for each tuple is computed by a hash function associated with the relation. For example, the tuple $a(X, Y)$ such that $h(X) = i$ resides at processor i.

In the course of the evaluation, a tuple may be added and deleted by multiple processors. In databases, concurrency control is concerned with guaranteeing that the result of all transactions is equivalent to some serial execution of those transactions. In our case, we can view the p_i's as

executing separate transactions, where a transaction consists of the actions executed by p_i at a given iteration. However, it is not enough to provide standard concurrency control on accesses to the distributed database, as some serial executions may not be acceptable according to the semantics. It becomes necessary to provide some means to act as an arbiter over database adds and deletes. This is realized by the *Data Handler* for each tuple. One can think of a data handler as a monitor program that runs on each processor that stores tuples and sequentially services requests for tuples by other processors. The data handlers are reponsible for realizing the correct semantics with respect to database updates, as will be demonstrated below. For this variation, fact chains are not necessary.

Restricted versions of the rules are distributed to the processors, as described earlier. These are called the *Rule Handlers* (RH) for program P. When a rule handler requires a fact for matching, it requests it from the appropriate data handler. Determining the appropriate data handler involves a trivial evaluation of the associated hash function.

Each processor p_i performs the instantiations of P_i as in the normal evaluation procedure. Any newly inferred facts are transmitted to the appropriate data handlers for storage. All processors execute in step with respect to the evaluation iteration number, i.e. all processors have the same iteration number, which is used for synchronization. The following procedure is executed by Rule Handler i at iteration j, in the synchronous variation of the data reduction paradigm.

procedure $SYNC\_DATA\_REDUCTION$

At iteration j do:

1. Request facts from appropriate Data Handlers for the rule instantiations of p_i with respect to the database at iteration j.
2. Add or delete tuples, as indicated by these instantiations. This is done by transmitting the operations (i.e additions or deletions of facts) to the appropriate Data Handlers.
3. When all the rule and data handlers have completed iteration j (as determined by some distributed termination algorithm, e.g. as in [17, 18]), p_i moves to the next iteration.

Data Handlers do the following sequence of steps:

1. Receive all the messages from the rule handlers, indicating the operations at iteration j.

2. Update local database as follows:

- If for a fact, f, an add and a delete operation is received (possibly from different rule handlers) the status of f at iteration $j+1$ remains the same as the status of f at iteration j. This is obviously also the case if no operation on f is received.
- Otherwise, if only adds (deletes) are received, f is added to (deleted from) the database.

end *procedure* $\{SYNC\_DATA\_REDUCTION\}$

Increased efficiency is implied by the fact that p_i evaluates a restricted version, meaning it does fewer instantiations compared to the case where the original program P is evaluated.

7 Distributed Asynchronous Evaluation of Datalog$^{\neg *}$ Programs

A major problem with the procedure described above is that no processor can proceed faster than the slowest one at each iteration. Obviously, we desire asynchronous operation of each p_i, but this gives rise to other problems which motivate the use of the $INCR\_UPDATE$ procedure.

Suppose some processor p_i deletes a fact f at iteration t_1, which was used earlier by processor p_j. Suppose also that p_j is now at iteration t_2, and $t_2 > t_1$. Unless handled properly, this situation may cause the operational semantics of Datalog$^{\neg *}$ to be violated.

Recall that $INCR\_UPDATE$ allows the incremental incorporation at iteration t_2, database changes occurring at iteration $t_1 < t_2$. The message indicating deletion of f by p_i in the example above can be viewed as an interrupt message. Thus, by using the procedure $INCR\_UPDATE$, p_j can update its database to a consistent state.

We outline below the asynchronous variation of the data reduction paradigm (henceforth called $ASYNC\_DATA\_REDUCTION$) for parallel and distributed evaluation of Datalog$^{\neg *}$

programs. As before, we have rule handlers that own restricted versions of the rules of the original program. Furthermore, when the database is partitioned among the processors, each *data handler* maintains a chain for each fact. The processors operate asynchronously and each rule handler keeps its own iteration count.

More precisely, in $ASYNC\_DATA\_REDUCTION$ each rule handler, RH_i, executes steps (1) and (2) as in $SYNC\_DATA\_REDUCTION$, at each iteration j. At step (3), instead of synchronizing, RH_i checks whether or not it received an interrupt message from a data handler; such a message contains a set of NEW (positive and negative) facts at an iteration number. If this is not the case, RH_i simply proceeds to the next iteration. If the message pertains to an iteration number higher than j, it is ignored and RH_i proceeds to the next iteration.

Otherwise, RH_i executes steps (7) and (8) of $INCR\_UPDATE$ for each interrupt message, in increasing iteration number order (meaning that it will not process a NEW set for iteration 500 if a NEW set for iteration 400 is in the queue). RH_i sends the resulting $UNDO$ and $REDO$ sets to the data handlers (note that each set will be partitioned, with each partition being sent to a different data handler). After processing all interrupt messages RH_i proceeds to iteration $j + 1$.

Each DH_i continuously receives $REDO$ and $UNDO$ sets, each associated with an iteration number. DH_i processes these in increasing iteration number order. For each *undo (redo)* set associated with an iteration, say k, DH_i performs $UNDO\_MAINTAIN\_CHAIN$ ($REDO\_MAINTAIN\_CHAIN$), and $COMPUTE\_NEW$. The resulting NEW set and the iteration number k are broadcast to the RH's. If both an undo and a redo set exist for k, then $UNDO\_MAINTAIN\_CHAIN$ and $REDO\_MAINTAIN\_CAHIN$ will both be performed before $COMPUTE\_NEW$.

It should be clear that many optimizations of the transmission sets are possible, as discussed in [6]. However, we omit these discussions here.

8 Conclusion

The main contribution of this paper is the $INCR\_UPDATE$ algorithm and the data structures to support it. We have shown that the algorithm can be used for two seemingly unrelated purposes: first, to maintain the inference database in an incremental fashion in a dynamic

environment where the input may be updated after inferencing is under way, and second, for the asynchronous version of the data reduction paradigm.

We plan to implement the *INCR_UPDATE* algorithm in NETMATE, a communication network management system under development at Columbia University [10]. As explained in the introduction, incremental update capabilities are necessary in this real time environment.

This work is also part of a research effort seeking to develop a new environment for parallel and distributed rule evaluation. In a companion paper, we describe a new rule language (PARULEL = PArallel RULE Language) with semantics similar to Datalog$^{\neg *}$, except that at each iteration additional control is provided by what we call *Meta Redaction Rules*. Meta redaction rules regard the set of instantiations at each cycle as working memory, and remove from the set of instantiations those members that are considered to be conflicting instantiations according to these rules. After redaction, all remaining instantiations in the conflict set can be fired in parallel.

It is expected that incremental rule processing, combined with parallel and distributed processing capabilities will realize real-time expert systems with large databases.

A Pseudocode for Maintenance of Fact Chains

procedure $INITIALIZE\_CHAINS$

$DB \leftarrow GlobalDatabase$

For each $\mathcal{F} \in DB$ do

Create a new chain $\mathcal{C}(\mathcal{F})$ indexed by $\mathcal{F}$.

Link a *state* record s to $\mathcal{C}(\mathcal{F})$. Set the fields of s as follows:

$s.Added \leftarrow 1$

$s.Deleted \leftarrow 0$

$s.Iter \leftarrow 0$

$s.Status \leftarrow In$

end $\{procedure INITIALIZE\_CHAINS\}$

procedure $REDO\_MAINTAIN\_CHAIN(f, i)$

$\mathcal{F} \leftarrow | f |$

$DB \leftarrow GlobalDatabase$

If $\mathcal{C}(\mathcal{F})$ exists

If $(\exists s \in \mathcal{C}(\mathcal{F}))$ such that $s.Iter = i$, then

if f is positive, set $s.Added \leftarrow s.Added + 1$

else if f is negative, set $s.Deleted \leftarrow s.Deleted + 1$

Else

Insert a new state record p at the correct position in the chain.

$p.Iter \leftarrow i$

if f is positive, set $p.Added \leftarrow 1, p.Deleted \leftarrow 0$

else $p.Deleted \leftarrow 1, p.Added \leftarrow 0$

Else if there is no $\mathcal{C}(\mathcal{F})$ then

Create $\mathcal{C}(\mathcal{F})$ and link a state record s to it.

Set $s.Added, s.Deleted, s.Iter$ fields to 0, and $s.Status$ to Out.

Add a state record p to the end of the chain.

$p.Iter \leftarrow i$

If f is positive then $p.Added \leftarrow 1, p.Deleted \leftarrow 0, DB \leftarrow DB \cup \{f\}$

else if f is negative then $p.Added \leftarrow 0, p.Deleted \leftarrow 1$

end $\{procedure REDO\_MAINTAIN\_CHAIN\}$

procedure $UNDO\_MAINTAIN\_CHAIN(f, i)$

$\mathcal{F} \leftarrow |f|$

Find the state record s in $\mathcal{C}(\mathcal{F})$ with $s.Iter = i$

Comment: cancel adds or deletes from some previous iteration.

If f is positive then $s.Added \leftarrow s.Added - 1$

else $s.Deleted \leftarrow s.Deleted - 1$

end $\{procedure UNDO\_MAINTAIN\_CHAIN\}$

function $COMPUTE\_NEW(i)$

$NewSet \leftarrow \emptyset$

For each fact chain that has been updated

Find the state record s at iteration i if it exists

if $((s.Added = 0) \wedge (s.Deleted = 0))$ then $s.Status \leftarrow$ Logical Status at $(i - 1)$

else if $((s.Added \neq 0) \wedge (s.Deleted \neq 0))$ then $s.Status \leftarrow$ Logical Status at $(i - 1)$

else if $(s.Deleted = 0)$ then $s.Status \leftarrow In$

else if $(s.Added = 0)$ then $s.Status \leftarrow Out$

If the new status represents an inversion, then

$f \leftarrow$ Fact under consideration, with sign.

$NewSet \leftarrow NewSet \cup \{f\}$

RETURN(NewSet)

end $\{functionCOMPUTE\_NEW\}$

References

[1] D. Maier and D. S. Warren; *Computing with Logic: Introduction to Logic Programming; Benjamin-Cummings Publishing Co., 1987.*

[2] J.D. Ullman; *Principles of Database and Knowledge-Base Systems; Vol. 2, Computer Science Press, 1989.*

[3] S. Abiteboul and E. Simon; *Fundamental properties of deterministic and nondeterministic extensions of Datalog*; Journal of Theoretical Computer Science, 1990.*

[4] S.J. Stolfo, D.P. Miranker and R. Mills; *A simple processing scheme to extract and load balance implicit parallelism in the concurrent match of production rules; Proceedings of the AFIPS symp. on fifth generation computing, AFIPS,1985.*

[5] T. Ishida, S.J. Stolfo; *Towards the Parallel Execution of Rules in Production System Programs; Proceedings of the 13th annual international Symposium on Computer Architecture, pp. 28-37, IEEE/ACM, 1986.*

[6] O. Wolfson and A. Ozeri; *A New Paradigm for Parallel and Distributed Rule-Processing; Proceedings of the ACM-SIGMOD 1990, International Conference on Management of Data, Atlantic City, NJ, May 1990.*

[7] Jon Doyle, *A Truth Maintenance System; Readings in Artificial Intelligence; Morgan Kaufmann, 1981; pp. 496-516.*

[8] A. Barr and E. Feigenbaum, eds.; *The Handbook of Artificial Intelligence, Vol. 2.*

[9] Raymond Reiter and Johan de Kleer; *Foundations of Assumption-Based Truth Maintenance Systems: Preliminary Report; Proceedings of AAAI-87, pp. 183-188*

[10] A. Dupuy, S. Sengupta, O. Wolfson, Y. Yemini; *NETMATE: A Network Management Environment; to appear in the special issue on Network Operations and Management, IEEE Network (The Magazine of Computer Communications), 1991.*

[11] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang; *A predicate matching algorithm for database and rule systems; Proceedings of the ACM-SIGMOD 1990, International Conference on Management of Data, Atlantic City, NJ, May 1990; pp. 271-280*

[12] J.A. Blakeley, P.-A. Larson, F.Wm. Tompa; *Efficiently Updating Materialized Views; Proceedings of the ACM-SIGMOD 1986, International Conference on Management of Data, Washington, D.C., May 1986; pp. 61-71*

[13] S. Ganguly, A. Silberschatz, S. Tsur; *A Framework for the Parallel Processing of Queries; Manuscript, Computer Science Dept., Univ. of Texas at Austin, 1989.*

[14] K. R. Apt, H. Blair, A. Walker; *Towards a Theory of Declarative Knowledge; unpublished memorandum, IBM Yorktown Heights, NY.*

[15] S. Cohen and O. Wolfson: *Why a Single Parallelization Strategy is not Enough in Knowledge Bases; Proc. 8th ACM Symp. on PODS, pp. 200-216, 1989. Revised version to appear in a special issue of the Journal of Computer and System Sciences, 1991.*

[16] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh and P. Wilms; *A Snapshot Differential Refresh Algorithm; Proceedings of the ACM-SIGMOD 1986, International Conference on Management of Data, Washington, D.C., May 1986; pp. 53-60*

[17] K.M. Chandy and J. Misra; *On Proofs of Distributed Algorithms with Application to the problem of Termination Detection; Manuscript, Dept. of Computer Science, University of Texas, Austin.*

[18] N. Francez; *Distributed Termination; ACM Tmnsactions on Programming Languages and Systems, 2(1),pp.42-55,1980.*