Transaction Restarts in Prolog Database Systems

Shridhar Acharya
Gael Buckley
Department of Computer Science
State University of New York
Stony Brook, NY 11794

Abstract

This paper considers the problem of transaction restarts in a database system When a transaction cannot be completed, most concurrency control mechanisms abort the transaction and reexecute the entire transaction We propose an alternate scheme that allows a transaction to restart from an intermediate state Restarting a failed transaction from an intermediate state can result in substantial reduction of unnecessary computation which would be done if the entire transaction were reexecuted We introduce a notion of state and consistent state of a transaction, and present a scheme that restarts transactions from consistent intermediate states A implementation of the scheme using Prolog as the query language is presented

1. Introduction

A database system consists of a set of objects and user processes that access those objects One execution of a user process is termed a *transaction* A database system usually imposes *consistency constraints* on the objects, and requires that any single transaction, executing alone, transform the database from one consistent state to another

Concurrent access to shared objects is used to improve throughput and response time for each transaction When many transactions access the database concurrently, the outcome of their execution should be equivalent to some serial execution of the transactions A system which guarantees this is said to ensure *serializability*

Database systems must remove transactions from the system for reasons such as deadlock, creation of a nonserializable schedule, etc When a transaction is aborted, the system usually restarts the transaction from the beginning, which can result in significant computation identical to the previous execution of the transaction This is not necessary if previous states of the transaction are saved in some fashion Consider a database system using locks, where a transaction has to unlock some number of items to break a deadlock If the system had saved the state of the transaction before these locks were acquired, the transaction could start execution again from that state, and not need to reacquire locks or reaccess the data items, saving disk access However, it is not easy to determine when to save the state and what variables need to be saved The added overhead to save the state may well exceed the savings of not reexecuting the transaction from the beginning

Recently, there have been several proposals to use the logic programming language Prolog as a database interface [10] Many issues to extend Prolog have been considered, including extending the language to include database updates [13] Other researchers [4] show that use of logic programs as a query language provides a convenient formalism for studying database problems Prolog execution uses backtracking, and hence relies on mechanisms that save intermediate states of the computation We show that the same mechanisms can be adapted and used to restart transactions from intermediate consistent states We use two different concurrency controls to illustrate how a transaction which is aborted due to nonserializability can be restarted from some intermediate state, thus avoiding the need to redo significant valid computation

There are several major approaches to concurrency control, including locking schemes[3, 11, 14],

timestamp schemes(TS) [2, 8] and optimistic schemes [1, 6, 9] Optimistic schemes allow unrestricted access to the database during the execution of transactions, and check for serializability only when a transaction completes Hence, such schemes have no locking and deadlock detection overhead and, if conflicts are rare, allow a large number of transactions to run concurrently [5] These characteristics provide two major reasons why we devote our major interest in this paper to adapting Prolog mechanisms to optimistic concurrency control for two reasons First, it seems a good choice when dealing with such systems as expert systems, which are query dominant and have complex searches In this case, the slight overhead incurred is more than compensated by the savings to avoid recomputing states Second, the optimistic concurrency control scheme uses Prolog mechanisms for concurrency which simple locking schemes cannot take advantage of, as is illustrated in a later section

The rest of the paper is organized as follows We present an overview of the Prolog state maintenance mechanisms in section 2 In Section 3 we describe optimistic concurrency control schemes in detail, define a consistent state of a transaction, and give the restart algorithm Section 4 contains how the restart algorithm can be used in Prolog systems, and in Section 5 we present the modifications needed in the Prolog mechanisms to accommodate our scheme Section 6 outlines the overheads involved in our implementation, and in Section 7 we briefly discuss additional research areas

2. Prolog

In this section we informally explain the Prolog execution strategy The implementation of our transaction restart scheme uses existing mechanisms of Prolog systems, and requires some knowledge of the behavior of Prolog implementations We begin by briefly describing the pertinent aspects of such prolog implementations, which are described in detail in [12] Database prolog is similar to ordinary prolog except that the facts are stored in a relational database Users can have their own set of facts as well as access the facts in the database In the discussion that follows we assume, for the sake of simplicity, that all the facts are stored in the database

To avoid discussing Prolog execution in detail, we present an example which uses the mechanisms we will alter for the concurrency control algorithm, and that can also be easily read as a logical query with minimal reference to exact Prolog execution Consider an environment where a supervisor wishes to fire anyone who does not produce adequately and has been on the job for less than 2 years We execute this query on the database given in Example 1

Example 1

```
1 notprod(X) - poorworker(X),longlunch(X)
2 notprod(X) - absent(X,Z), Z > 39
3 poorworker(Mark)
4 absent(Irma,40)
5 employed(Mark,1 5)
6 employed(Irma,1 5)
```

Rules 1 and 2 in Example 1 can be read as the conjunction of subgoals on the right side of the " -" notation implies the left side The left side is called the *head* of the rule and the right side the *body* Hence, this database states that X does not produce adequately if X is a poor worker and takes a long lunch (by rule 1), or if X is absent more than 39 (days, presumably), by rule 2 Rules 3 through 6 are tuples of relations, or facts, about the employees Mark and Irma

Whenever more than one rule has a head which matches the same goal, the execution faces different alternatives Prolog uses a depth first search combined with backtracking to find all the alternatives The execution of example 1 will explain the Prolog execution strategy

Let the goal be to find the employees who don't produce enough and have been with the company for less than 2 years This can be stated as - notprod(X),employed(X,Y),Y < 2, where each predicate to be solved is referred to as a *subgoal* A subgoal is solved by placing an activation record for it in the stack and applying a rule whose head matches the subgoal Since the goal statement is a conjunction of subgoals, the system first tries to solve the left most subgoal and, only if that succeeds, does it try to solve the rest of the subgoals In our example, notprod(X) can be solved by either rule 1 or rule 2, which are termed *available procedures*, denoted by **av** Prolog systems apply these in order of appearance, and so an activation record for notprod(X) is placed on the stack, rule 1 is selected to solve notprod(X), notprod(X) is replaced by the body of rule 1, and the new goal statement becomes - poorworker(X), longlunch(X),employed(X,Y),Y < 2

The system continues the above process of applying rules to the left most subgoal and producing a new goal statement, until the goal statement becomes empty The only way that the goal statement can become empty is when the applied procedures are facts In this example the new goal statement has poorworker(X) solved by rule 3, and X gets bound to Mark Since rule 3 is a fact the new goal statement becomes -longlunch(Mark) employed(Mark,Y),Y<2

Note that the goal statement has been reduced because rule 3 is a fact However, longlunch(Mark) is not in the database, and so the goal statement cannot be solved any further The system backtracks and attempts to solve the earlier goal statement - notprod(X),employed(X,Y),Y< 2 using a different rule This time rule 2 is chosen to solve the subgoal notprod(X), and rule 4 succeeds by finding absent(Irma,40) In general, if no more rules exist to solve a particular subgoal, then its activation record is removed and other alternatives, if any, are tried for the previous activation record If no more alternatives exist, this activation record is also removed If no activation records are left in the stack then the query fails

Pictorially, we will represent the activation stack as a sequence of records, with the most recent record on the right Each record has the subgoal being solved at the bottom of the record, the procedure presently being used to solve it denoted by **ap** (for applied procedure), and other procedures which can be used to solve it denoted by **av** (for available procedure) When a rule is rejected it is denoted by **rp**

We present four stacks for Example 1 for the following cases

1) just before the system backtracks
2) when longlunch(Mark) has been rejected
3) when rule 2 is invoked
4) the final stack for the query

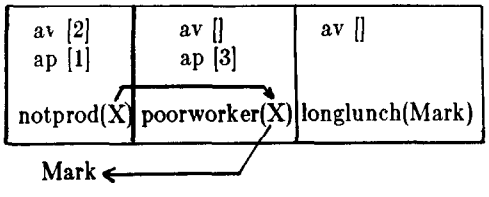


figure : 1a

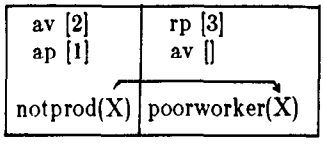


figure : 1b

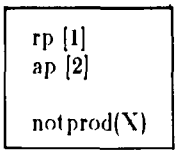


figure . 1c

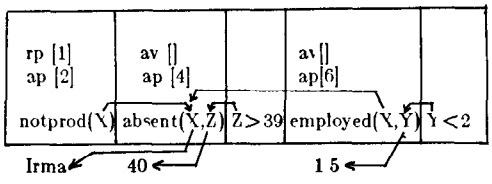


figure • 1d

In the figures we use, arrows associated with variables indicate the bindings of the variables by the applied procedures When an applied procedure is rejected, bindings done by it are removed Thus when rule 1 is rejected for solving poorworker(X), X's binding to Mark is removed in figure 1b Prolog uses a depth first search strategy, which amounts to executing the goal to a successful conclusion using the first rules which succeed If additional answers are required, then the user tells the system to fail with the present answer and search for the next one in turn, until as many answers are returned as are required This mechanism to return a varying number of answers plays an important part in the use of optimistic concurrency control schemes with prolog, as will be shown later using this example as an illustration

2.1. Adapting Prolog Mechanisms for Two Phase Locking

As a simple case to show how these state saving mechanisms can be used in concurrency control, we briefly discuss how this can be used to restart transactions which must release some number of locks, as in the example of deadlock given in the introduction In two phase locking, a transaction must lock an item before it accesses that item, and cannot unlock an item until all accesses are complete Hence, the lock table needs to include a pointer to the activation record containing the rule for the first access to an item, when the lock was issued Breaking deadlock requires all locks obtained after the lock being released must also be released This implies that the stack to the right and including the activation record in which the lock was issued must be removed, and execution restarted from the top entry of the stack This would require the name of the transaction issuing the lock to be kept in the lock table, which is already a feature in some concurrency control systems [7]

We now discuss optimistic concurrency control schemes, and how the mechanisms of Prolog would be adapted to conform to such a concurrency control As stated earlier, we believe this concurrency control to be well suited to database systems with more complex retrievals, such as expert systems

3 Optimistic Schemes & Transaction State

In this section we describe optimistic concurrency control in detail We define states and consistent states of a transaction, and describe how a restart algorithm can determine the most recent consistent state

3.1. Optimistic Concurrency Control

A transaction in an optimistic concurrency control scheme has a consistent execution if no other transaction updated the variables it read between the time it started and finished execution The "time" read is the value of the *global transaction number*, termed **TN**, which is the largest transaction number of all completed transactions

A transaction consists of three phases a read phase, a validation phase and a write phase When a transaction T enters the system it initializes its start transaction number, **stn(T)** to the value of TN and enters its read phase In this phase the transaction does all its computation as well as database accesses Each transaction has a *read set*, a set of database objects that the transaction read, and a *write set*, a set of database objects that the transaction intends to modify together with their new values Both sets are maintained by the concurrency control mechanism in a buffer called the *transaction buffer* A read request by transaction T for object B is executed in the following manner If the object B already exists in the transaction buffer, its value is returned from the buffer, otherwise it is read from the database and the value inserted into the buffer When the transaction writes a value for an object, both its name and value are inserted into the buffer, possibly overwriting a previous value for the object Note that none of the modifications are done onto the database when the transaction is executing, so when a transaction is aborted all that need be done is discard its transaction buffer

The transaction enters its validation phase after it finishes all its computation It is assigned a finish transaction number **ftn(T)**, which is the value of TN when T enters its validation phase The system then checks that the read set of T does not intersect the write sets of any transaction with transaction numbers between stn(T)+1 and ftn(T) If it does not, then the objects that T read were not changed between the time T started and finished, so T succeeds validation TN is incremented by 1 to indicate completion of one more transaction, T is assigned the transaction number equal to the value of TN, and the write phase incorporates all T's changes into the database This is to be done in mutually exclusive mode with other transactions There are several other validation algorithms [6] which decrease the time required in the mutually exclusive mode,but require a more complex validation algorithm

3 2. State of a Transaction

We will use the information about read sets described earlier to determine at which point a transaction can restart computation To facilitate this, we introduce the concept of the *state* of a transaction and define consistency of a transaction state We then describe a restart algorithm which determines the most recent consistent state from which to restart a transaction

Definition 3.1· The *state* of a transaction T is the tuple (V_i,D_i), where the set V_i is the set of the value of local variables of the transaction and D_i is the read set of the transaction after execution of step i The state of the transaction T after step i is termed S_i

Definition 3 2: The state S_i is a *consistent state* if the set D_i does not intersect the write sets of transactions with transaction number between stn(T)+1 and ftn(T) inclusive If it does, then S_i is an inconsistent state

We do not put restrictions on what a step is, other than a read or write action be part of only one step S_i can be determined to be consistent for the optimistic concurrency control scheme only when T has finished execution and obtains a value for ftn(T) Validation of a completing transaction is thus equivalent to determining whether the final state of the transaction, the state after it has executed its last step, is a consistent state Transactions operating under the two phase locking scheme, as described earlier, have every state consistent due to the two phase restriction

To illustrate this concept consider the following transaction T, which determines the number of man-years from employees of particular departments, using the database given in Example 1 T adds the number of years Mark and Irma were employeed to the total The previous total for all employees within the department was 40 Tbegin and Tend denote the begin and end of the transaction Prolog systems model update operations as a delete and an upadate operation and we have done the same in the following example

Example 2

T **Tbegin**
1 - employed(Mark,X0),
2 employed(Irma,Y0),
3 total(Z0),
4 delete(total(Z0)),
5 Z0 is Y0+X0+Z0,
6 write(total(Z0))
Tend.

If T is the first and only transaction executing in the system then the states of the transaction are

$S_0=(\phi,\phi)$

$S_1=(\{X0=1\ 5\},\{employed(Mark,1\ 5)\})$

$S_2=(\{X0=1\ 5,Y0=1\ 5\},$
{employed(Mark,1 5),
employed(Irma,1 5)})

$S_3=(\{X0=1\ 5,Y0=1\ 5,Z0=40\},$
{employed(Mark,1 5),
employed(Irma,1 5),total(40)})

$S_4=(\{X0=1\ 5,Y0=1\ 5,Z0=40\},$
{employed(Mark,1 5),
employed(Irma,1 5),total(40)})

$S_5=(\{X0=1\ 5,Y0=1\ 5,Z0=43\},$
{employed(Mark,1 5),
employed(Irma,1 5),total(40)})

$S_6=(\{X0=1\ 5,Y0=1\ 5\ Z0=43\},$
{employed(Mark,1 5),
employed(Irma,1 5),total(40)})

A selection of tuples may be abbreviated in the read set by the name of the relation restricted by any fields which are constants for the selection

If T is run concurrently with other transactions, we need to check if S_6 is a consistent state If T is the first transaction stn(T)=0 If some other transaction T' deletes the total(40) tuple and inserts total(50) between the time T starts and finishes, then T' is assigned a transaction number equal to 1 T then completes and attempts validation with ftn(1) = 1 This will fail since the sets D_3, D_4 D_5 and D_6 contain total(40), which was also in the writeset of T' The corresponding states S_3, S_4 and S_5 are inconsistent with the present database

If we abort T and restart it, it would get a stn of 1 T would read the new value of total, and its execution would be identical to the above, with Z0=50 initially rather than 40 However, we can eliminate the two accesses to the employed relation by restoring T to S_2 and assigning stn(T)=1 Execution will begin from step 3, and T's computation will proceed exactly as if it had been aborted and restarted Though in this case we save the access of only two tuples, for complicated queries the savings can be substantial

3.3. Restart Algorithm

In general, if the previous states of the transaction are kept, a transaction T that fails validation can be restarted as follows Consider a transaction T with start and finish transaction numbers stn(T) and ftn(T), and perform the following

(1) Determine a consistent state S_j, such that S_{j+1} is inconsistent

(2) Reassign ftn(T) as stn(T), restore T to state S_j, and begin execution from S_{j+1}

This algorithm is correct for the following reason If S_j is the state to which it is restored, then all data items read up to and including S_j pass validation Hence transactions with transaction numbers between stn(T) + 1 and ftn(T) have not overwritten the objects in D_i for all $i \leq j$ Therefore the state S_j is a consistent state that is not dependent on any transactions with transaction numbers between stn(T)+1 and ftn(T) Hence, if the transaction is aborted and restarted, it could reach S_j with a new stn(T) equal to the previous ftn(T)

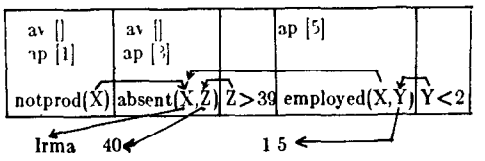
We now illustrate how to adapt the mechanisms of Prolog to handle restart from intermediate states of an optimistic concurrency control, and show procedures to implement the scheme

4. Transaction Restart in Database Prolog

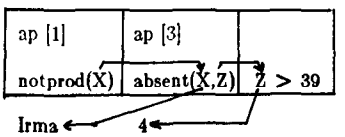
4.1. Implementation of Our Scheme

We begin our discussion of the mechanism needed with a reference to the execution of example 1, where Irma is the one answer returned by the database in response to the query which workers do not produce sufficiently and are employed for less than 2 years This query (-notprod(X),employed(X,Y),Y<2), is executed on the database of example 3 and results in the following stack, which binds X to Irma

Example 3
1 notprod(X) - absent(X,Z), Z>39
2 poorworker(Mark)
3 absent(Irma,40)
4 employed(Mark,1 5)
5 employed(Irma,1 5)



The read set of transaction T is {absent(Irma,40),employed(Irma,1 5)} At validation this read set will have to be checked with the write sets of transactions that completed between the time T started and finished Let us assume that T fails validation because some concurrently executing transaction rectifies a typographical error and modifies the relation absent by deleting the tuple absent(Irma,10) and adding tuple absent(Irma,1) Then T would be restarted from the stack before the tuple absent(Irma,40) was read The intermediate stack of the restarted transaction is shown below, and the query will not return Irma because Z>39 will fail



Thus, one addition to the Prolog state is the names of the tuples or objects read in that state The state in which to restart a transaction is determined by removing all the records including and above the lowest activation record that contains an object causing validation to fail, and restarting from the new top of the stack

However, this is not sufficient for the interpretation of optimistic concurrency control together with Prolog execution In optimistic concurrency control, the serialization order is expected to be in increasing order of transaction number This is not the case using only the read-sets of successful procedures, as is shown in the following example Let us change the goal to require that an employee must be employed more than two years, written -notprod(X), employed(X,Y),Y>2 This query would fail with the original database, since Irma was only employed 1 5 years However, if this transaction validated after the update transaction that changed the tuple to 2 1 years, it would not conform to the requirement that the order of validation is the serial order of transactions as stated by optimistic concurrency controls This is equivalent to stating that a query may not return all the answers it would if the transactions were executed serially in order of transaction number This may not be required for all queries, for if Irma is not discovered by this execution of the present transaction, she will be discovered by a future execution If all answers are required, the optimistic concurrency control mechanism in Prolog must retain additional information to ensure this

The above effect is compounded by the traditional use of Prolog, where many programmers use the rules in a sequential order to implement if then else, even though the order of execution is not defined as part of the prolog language Hence in example 1, rule 2 would be used only if rule 1 were to fail In this case, the activation records in the stack corresponding to rule 1 are removed, together with their read sets Consider again example 1 When procedure 3 is selected to solve the subgoal, poorworker(Mark) is read from the database, and longlunch(Mark) is not found and the rule fails Because of the absence of longlunch(Mark) in the database, procedure 1 is rejected for solving the subgoal The system then chooses procedure 2 to solve the subgoal and succeeds The read set of the transaction is { absent(Irma,40) } If the transaction fails validation because of longlunch(Mark), the record which contained longlunch(Mark) was backtracked over and removed when rule 1 was rejected to solve notprod(X) Hence this activation record no longer exists, and there is no method to determine which record is consistent and can be used to restart execution The additional mechanism that we introduce, below, overcomes this shortcoming

With every activation record we associate a read-set of database objects If the activation record is eventually removed due to the failure of its procedures, its read-set contains the set of database objects that were read between the time its procedure was selected to attempt to solve a subgoal and and the time it was rejected For example, after rule [1] was selected to solve the subgoal notprod(X) the tuples poorworker(Mark) and longlunch(Mark) were "read" from the database Thus these two tuples should be in its read-set If the activation record represents a successful procedure, the read set is the database relations read for the applied procedure of the activation record, and all read-sets of rejected procedures until the next activation record for a successful procedure on the stack In this manner the tuples read by rejected procedures are associated with the nearest lower activation record containing a successful procedure An activation record is called *consistent* if its read-set does not fail validation

Read-sets for activation records are constructed according to the following rules

(1) When an activation record is created, its read-set is empty

(2) If its applied (current) procedure is a relation stored in the database with all fields constants or bound to constants by prior execution, then the read set is this tuple and the previous read-set of the record If the procedure is a relation with some unbound variables and the remaining fields match a tuple in the database, the readset is this tuple and the previous read-set of the record If the procedure is a relation with variables whose fields cannot be satisfied by any tuple in the database, then the call fails In this case, the read set is the relation with all variable fields filled in with '_' (denoting unbound variables), together with the previous read-set of the record This is done to prevent phantom problems [9] as will be explained later

(3) If an activation record at height q in the stack is removed due to the failure of its associated procedure, then the read-set of this record is added to the read-set of the record at height q-1

(4) The read-set for the transaction is the read-sets of all activation records in the stack

We further need to distinguish which tuples in the read sets of an activation record were contributed by a particular procedure The procedure for condition (2) is the named relation The procedure for condition (3) is the procedure associated in the activation record at height q, which is also a rejected procedure i the activation record at height q-1

We again draw the stack of example 1, adding the read sets (denoted by rd) with each applied procedure at each activation record

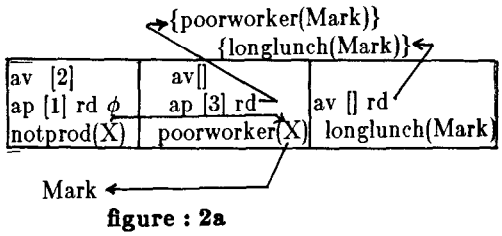


figure : 2a

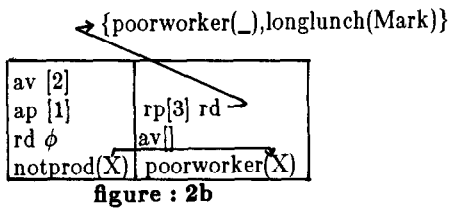


figure : 2b

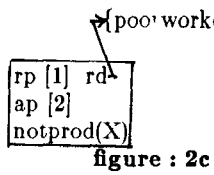


figure : 2c

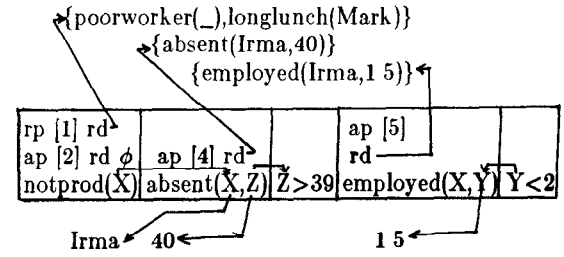


figure : 2d

The call longlunch(Mark) fails and so its activation record is removed and its read-set is transferred to the activation record that represents poorworker(X) (figure 2b) Since relation poorworker has only one tuple the activation record for poorworker is removed and its read set transfered to notprod(X)

4.2. Validation

At validation we form the read-set for the query, which is the union of read-set's of all the activation records, and pass it to the database manager to validate the transaction This corresponds to the read-set of the transaction as defined in section 3, for the following reasons Consider an element in the read-set of the transaction as defined in the previous section This must be an element of the read-set of some activation record, and was either successfully used by a procedure existing in the stack, or was applied and later rejected If the procedure still exists, it must be in the read set of the transaction If the procedure was rejected, the change in value of this tuple implies that additional answers may be available, and the transaction must restart

4.2 1. The Backtrack Algorithm

At validation, the concurrency control mechanism determines if the read-set of the transaction is consistent with previous transactions If the validation fails, it passes back the predicates that failed validation The backtrack algorithm, given below, depends on this information to find the activation record from which to restart

(1) The predicates that failed validation must belong to the read-set of some rejected or applied procedure in the stack Hence when validation fails, the read-set for some activation record has failed validation Determine the smallest offset from the bottom of the stack for any predicate that failed validation, which we denote N_1 One simple method to do this is keep a hash table of relations accessed, with pointers into the record which read the relation

(2) Remove all records above N_1 and delete read-sets for the removed records

(3) Determine the first procedure P, in order of appearance, for activation record N_1 whose read-set contains an object that caused validation to fail Return all procedures between the applied procedure for N_1 and P to the available list(including both P and the applied procedure) Delete the read-sets for procedures that were returned to the available list

(4) Restart the transaction from activation record N_1 and procedure P

In example 2, if predicate absent(Irma,40) fails validation because some other transaction modified it to absent(Irma,4), the failed predicate is in the applied procedure 4's read-set Hence we will have to backtrack the system until the stack looks as follows

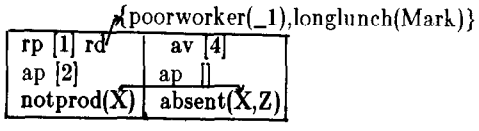


figure 3

Procedure 4 is now in the available list and we can read the modified relation absent(Irma,4) and correctly determne how many days Irma was absent Thus we save re-doing the computation that tried to check if Mark was a nonproductive worker

5. Implementation

In this section we describe in detail the changes needed to the Prolog routines to implement the backtack algorithm presented above In Prolog implementations the activation record contains space for the local variables, a pointer to the parent goal, pointers to the rejected procedures, the applied procedure, and the available procedures There is a current stack pointer which points to the top of the stack Our system assumes the same runtime system as described in [12] with some modifications described below

Existing prolog systems use a special procedure called **fail**, which is called when a procedure is rejected and alternative procedures have to be tried What **fail** does is try the next procedure in the av list from the top activation record to solve the goal If the av list is empty the system deletes the top record and sets the current stack pointer to the previous activation record All variable bindings done by the currently applied procedure in the activation record are removed , the procedure is moved to the rp list and another procedure from the av list becomes the applied procedure

For simplicity, we implement the read-set as a linked list of predicates, but it can also be implemented as hash tables or other data structure The list is a FIFO queue, so that the predicates read first are earlier in the queue than those that were accessed later The read-set is a global data structure visible by all activation records in the stack

To implement our scheme we propose the following changes

(1) Changes in the activation record

(2) Changes in the fail procedure

(3) Introduce a new fail procedure called **dbfail** which will be used to find the optimal activation record to restart when, validation fails

We explain the above three modifications to the system in detail below

5.1. Changes to the Activation Records

We implement read sets as a list of predicates and associate one pointer for each available procedure For each procedure P that has a non empty read-set we have one pointer, FinishIL, which points to the end of its read-set When the record N_1 is created the FinishIL pointer for all available procedures for that call are set to the current end of the global read-set Whenever a predicate read is done, the predicate is added to the end of the read-set and the FinishIL pointer for the current procedure points to the new end of the read-set In effect we have provided for a mechanism which keeps account of all reads done during each procedure

5.2. Changes to the Fail routine

We modify the **fail** routine as follows Whenever **fail** is called it undoes the variable bindings made by the applied procedure in the top activation record It then checks if there are any procedures in the av list and tries those If the av list is empty then the top activation record has to be removed By construction of the read-sets for activation records, the read-set associated with the rejected procedures must be transferred to the previous record when the top record is removed

In order to change the stack to be consistent with Prolog execution and maintain correct read sets, the following steps need to be performed Undo all variable bindings done by the applied procedure in the top ictivation record Mark the applied procedure as rejected and look for the next procedure in the av list If the av list is empty the current stack pointer is reset to the next lower activation record and the top record is removed Since the removed activation record may have added tuples to the read set the FinishIL pointer in the current activation record and

applied procedure should be set to the end of the read set

The algorithm for fail is specified as follows

```
procedure fail,
  begin
   UndoBindings Of Applied Procedure,
   If applied procedure is write then remove
        last item from write-set,
   applied procedure FinishIL = Read Set End,
   include applied procedure in the rejected
     procedure set,
   if av list is not empty
     then
       use first procedure from that set,
     else {
       pop the top activation record,
       fail,
      }
   end fail,
```

5.2.1. A new dbfail procedure

When the goal is finally reduced to empty, or if **fail** backtracks to the root with the no alternatives to try, the validation phase is started The read and write sets are passed to the concurrency control mechanism and the read set is validated If validation succeeds, the transaction terminates If validation fails the transaction has to be restarted which is done as follows

(1) Assign a new stn to the transaction with the value of its present ftn

(2) Determine the first predicate in the intention list that failed validation The validation mechanism can pass this back when validation fails

(3) Determine the activation record that has the failed predicate in its read set Let N_1 be the record and P be the procedure in N_1 which has the failed predicate in its read set

(4) Call dbfail The algorithm for dbfail is given below

```
  procedure dbfail,
  begin
  while top activation record ≠ N1
    do

      delete bindings made by applied
           procedure in top most record,
      if applied procedure is write remove last
        item from write set,
      delete FinishIL pointers,
      pop the top activation record,
    end while,
    delete all bindings made by
      applied procedure,
    if applied procedure ≠ P
      then
       repeat
        if applied procedure is write remove
          last item from write set,
        delete FinishIL pointers,
        return applied procedure to
          available procedure list,
        assign the last procedure in the
          rejected procedure list as
          the applied procedure,
       until applied procedure is P,
     endif,
   end dbfail,
```

6. Overheads Involved

The overheads involved in our scheme are minimal A Prolog system that uses optimistic schemes for concurrency control has to maintain read-sets in any case We choose to keep the read sets in the Prolog system rather than with the concurrency control mechanism The only overhead our scheme imposes on the system is the newdbfail routine and the additional pointers in the activation records

7. Conclusion & Further Work

We have presented a scheme which allows transaction restarts from intermediate transaction states Our scheme also works in Prolog systems that allow cuts in programs Cuts restrict the forward search space of the program and hence does not affect our scheme Our scheme is valid for serial as well as parallel schemes of validation [6]

Our scheme avoids phantom problems [9] by associating read sets with failed reads from the database Consider a procedure p(Mark,X) where p is a database relation, Mark is a constant and X a variable If the relation p has no tuples with the first attribute as Mark then the call fails and in our scheme we associate the read set p(Mark _) with the call This locks all tuples, existing and nonexisting, in the relation p with the first attribute Mark and thus prevents the phantom problem This reduces the concurrency in the system

Optimistic schemes favor short-lived transactions Long-lived transactions which perform many updates to the database are generally not favored We can justify our use of optimistic schemes by arguing that we expect most of our transactions to be short-lived ones, but we cannot rule out long-lived update transactions from the system Our scheme, by not restarting transactions from the beginning, increases the probability that long transactions will go through

Prolog queries can be viewed naturally as being composed of subqueries Each subgoal can be con-

sidered to be a subtransaction with its own validation phase A concurrency control mechanism in such a framework would then not favor short lived transactions For example if p,q,r is our query we can solve and validate p, then solve and validate q, etc If q fails we need only backtrack q and not p since p has succeeded validation Optimistic schemes should be devised which allow validation of partial read-sets

REFERENCES

1 D Agarwal, A Bernstein, P Gupta and S Sengupta, "Distributed Multi-version Optimistic Concurrency Control ", *Unpubished Report* , SUNY Stony Brook,

2 P A Bernstein and N Goodman, "Concurrency Control Algorithms for Multiversion Database Systems", , 1982

3 K P Eswaran, J N Gray, R A Lorie and I L Traiger, "The Notion of Consistency and Predicate Locks in Database System", *Comm ACM*, **19**, 11 (Nov 1976), 624-633

4 H Gallaire, J Minker and J Nicolas, "Logic and Databases A deductive Approach", *ACM Computing Surveys*, **16**, 2 (June 1984), 153-186

5 T Haerder, "Observations on Optimistic Concurrency Control Schemes", *IBM Research Report*, Oct 1982

6 H T Kung and J T Robinson, "On Optimistic Methods for Concurrency", *ACM Transactions on Database Systems*, **6**, 2 (June 1981), 213-226

7 B Liskov and R Scheifler, "Guardians and Actions Linguistic Support for Robust, Distributed Programs", *Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982

8 D P Reed, *Naming and Synchronization in a Decentralized Computer System*, PhD Thesis, MIT Department of EECS, Sep 1978

9 M Reimer and R P Bragger, *Predicative Scheduling Integration of Locking and Optimistic Methods*, Institut fur Informatik, ETH, Zurich, July 1983

10 E Sciore and D S Warren, "Towards an Integrated Database-Prolog System", 84/079, SUNY Stony Brook, June 1984

11 A Silberschatz and Z Kedem, "Consistency in Hierarchical Database Systems", *J ACM*, **27**, 1 (Jan 1980), 72-80

12 D S Warren, "Prolog A Runtime Environment", *Unpublished Report*, SUNY Stony Brook, April 1982

13 D S Warren, "Database Updates in Pure Prolog", 84/073, SUNY Stony Brook, April 1984

14 M Yannakakis, "A Theory of Safe Locking in Database Systems", *J ACM*, **29**, 3 (July 1982) 718-740