MECHANISMS FOR CONCURRENCY CONTROL AND RECOVERY IN PROLOG - A PROPOSAL

by

Michael J. Carey
David J. DeWitt
Goetz Graefe

Computer Sciences Technical Report #560
October 1984

Mechanisms for Concurrency Control and Recovery in Prolog - A Proposal

Michael J. Carey
David J. DeWitt
Goetz Graefe

Computer Sciences Department
University of Wisconsin - Madison

This research was partially supported by the National Science Foundation under grant MCS82-01870 and by the IBM Corporation through an IBM Faculty Development Award.

ABSTRACT

Prolog is beginning to receive a great deal of attention as a vehicle for creating intelligent database management systems. This paper proposes concurrency control and recovery mechanisms which are particularly well-suited to a Prolog environment. The proposed concurrency control mechanism, query-fact locking, is based on a database concurrency control algorithm known as precision locking. The proposed recovery mechanism is based on a differential file scheme known as hypothetical databases. The combined effect of the proposed algorithms for concurrency control and recovery is a mechanism for correctly and reliably executing sequential Prolog programs and Concurrent Prolog programs in a multi-user environment with a shared knowledge base.

1. Introduction

During the past two years, researchers have begun to address the task of adding deductive capabilities to database management systems [Dahl82, DBE83, Pars83, Jark82]. Prolog [Cloc81] is frequently mentioned as a suitable implementation language for such systems. First, however, Prolog must be extended in a number of ways before it can used to process large databases in a multiuser environment. One extension is to augment Prolog to include some notion of file input and output. Second, concurrency control and recovery mechanisms need to be added. In this paper, we propose concurrency control and recovery mechanisms which are well-suited for a Prolog environment.

There are several forms of concurrency that must be controlled in a Prolog-based intelligent database management system, as Prolog semantics allow queries that update the facts and/or rules in the knowledge base (via the *assert* and *retract* operators). The first form of concurrency occurs in a Concurrent Prolog environment when a user initiates a query whose subqueries can be executed concurrently [Shap83]. This form of concurrency can occur only with subqueries whose variables are either already instantiated or will be instantiated by the subquery itself. For example, the subqueries of the rule:

parent(X,Y) :- father(X,Y): mother(X,Y).

can be executed concurrently. On the other hand, the subqueries of the rule:

grandfather(X,Z) :- father(X,Y), parent(Y,Z)

cannot be run concurrently, as father(X,Y) must instantiate Y before parent(Y,Z) can be executed. While one might control this form of concurrency with a formal mechanism, for the present time we have decided to resolve conflicts among concurrently executing subqueries by adopting before-image semantics and a differential-file based recovery mechanism. These will be addressed in more detail below.

The second form of concurrency that must be controlled in a Prolog-based intelligent database management system is the concurrent execution of queries initiated by different users. The queries themselves may or may not display the type of concurrency discussed above. The following example illustrates the need for a concurrency control mechanism in a multiuser Prolog environment. Assume that the database consists of facts of the form child(x,y) indicating that x is a child of y:

child(sue,larry)
child(carol,larry)
child(fred,larry)
child(joe,larry)

the rule:

grandchild(X,Y) :- child(Z,Y), child (X,Z).

and the two queries:

Q_1: grandchild(X,larry) Q_2: assert(child(john,sue), assert(child(alice,joe)

If Q_2 is run after Q_1 finishes, Q_1 will find that larry has no grandchildren. If Q_2 is run before Q_1 starts, Q_1 will produce the result set {john, alice}. The objective of *all* database concurrency control algorithms is permit the concurrent execution of operations from different queries while insuring that the final state of the database corresponds to *some* serial schedule of the transactions [Eswa76]. Thus either of the above results is correct.

Execution of Q_1 consists of a number of subqueries. First, Prolog will instantiate Z with the value "sue" and attempt to prove the subgoal child(X,sue). Following the outcome of this goal, Z will be instantiated with "carol". Execution proceeds until all of larry's children are tested. Assume rather than Q_2 being executed completely before or after Q_1, that it is instead executed after execution of the subgoal child(X,sue) and before instantiating Z with carol. The outcome of Q_1 in this case is that alice is the only grandchild of larry. Since this result it not equivalent to any serial schedule of Q_1 and Q_2, it is not correct.

In addition to illustrating the need for a concurrency control mechanism in a multiuser Prolog-based database system, this example illustrates a classical concurrency control problem known as *phantoms* [Eswa76]. When Q_1 began executing larry had no grandchildren. If Q_2 had added grandchildren for anybody other than larry, running Q_2 in the middle of Q_1 would not have caused any problems. It is interesting to note that concurrency control in Prolog is different than concurrency control in conventional database management systems as Prolog provides no vehicle for modifying facts already in the database. Thus, while conventional concurrency control mechanisms deal primarily with controlling access to the partial updates by

a query and pay little or no attention to the problems of phantoms, a concurrency control mechanism for Prolog must deal exclusively with the problem of phantom facts. In Section 2 we present a concurrency control mechanism that solves the problem of phantom facts.

The job of the database system recovery mechanism of recovery mechanisms is twofold. First, it must insure that all updates made by queries which commit (terminate normally) are durable. By durable one means that the changes made by the query will persist in spite of subsequent hardware and/or software failures. The second task of the recovery mechanism is to undo any changes to the database made by queries which partially execute before being terminated by the system (as the result of deadlock, for example) or aborting themselves (due to erroneous input data or the user hitting the delete key).

As discussed earlier, subgoals of Prolog queries may assert or retract database facts. When such a subgoal fails two options exist. The first is to undo its effects by removing the set of facts asserted by the query and replacing the set of facts retracted by it. The second option is to simply ignore the problem. This is the approach used by most Prolog implementations. We feel that this approach is unacceptable if Prolog is to be used to implement intelligent database systems. In Section 3, we present a recovery mechanism for Prolog based on the ideas of differential files [Seve76] and hypothetical relations [Ston80, Ston81, Wood83, Agra83b]. In addition to providing a mechanism for undoing the effects of subqueries that fail, this mechanism facilitates implementation of before-image semantics for parallel subqueries and insures that the updates made by a query are durable.

We have made the following assumptions in developing our proposed concurrency control and recovery mechanisms:

(1) *undo semantics for subqueries that fail* - The effects of a subquery that fails will be undone. Since a subquery that fails may use the assert and/or retract operators to cause a side effect, we will also present a mechanism which permits delaying undoing the effects of such subqueries until the query terminates.

(2) *before-image semantics for parallel subqueries* - Facts asserted and retracted by subqueries executed concurrently are not visible to one another.

(3) *'all' semantics for goals connected with conjuncts* - These are the standard Prolog semantics. Our approach assumes that updates made by the subqueries are applied in the sequence corresponding to the left to right ordering of the goals in the rule.

(4) *'any' semantics for goals connected with disjuncts* - While sequential Prolog specifies that disjuncts are to be evaluated in a left to right order, we have assumed that a goal is satisfied by the first successful subquery in a concurrent Prolog environment. Furthermore, once one subquery succeeds the remaining active subqueries are terminated.

(5) *fixed rule base* - For the present time we have assumed that queries do not assert new rules or retract existing ones. At the end of the paper we briefly outline the changes that would be required to handle subqueries that update the rule base.

2. Concurrency Control

In this section we propose a concurrency control mechanism for Prolog. We begin with a discussion of the concurrency control problems posed by Prolog, and then we describe the proposed concurrency control mechanism. After examining our proposal in detail, we reflect on why we prefer our mechanism over several alternatives which were also considered. We conclude this section with a summary of the salient features of our concurrency control proposal.

2.1. The Problem

As described in Section 1, Prolog transactions are user queries which access *facts* (or unit clauses) and *rules*, possibly also *asserting* or *retracting* facts during their execution. The job of the concurrency control mechanism is to prevent transactions submitted by multiple users from interfering with one another. In particular, it should make transactions *serializable* [Eswa76] — the effect of the concurrent execution of a set of transactions should be equivalent to some serial execution of the transactions. There are several ways in which Prolog transactions can conflict and produce behavior which is non-serializable:

Fact − Fact Conflicts. This problem arises when two concurrent transactions attempt to perform updates involving the same fact(s). For instance, consider the following pair of transactions:

T_1 :- assert(foo(a,b)), retract(foo(a,c)).

T_2 :- assert(foo(a,c)), retract(foo(a,b)).

The outcome of serially executing T_1 and then T_2 is that foo(a,b) is false and foo(a,c) is true. The outcome of executing the transactions serially in the opposite order is that foo(a,b) is true and foo(a,c) is false. However, if the two transactions are interleaved, executing their assert steps and then their retract steps, the outcome is non-serializable — both foo(a,c) and foo(a,b) are false.

Query − Fact Conflicts. This problem arises when one transaction asserts or retracts a fact used by another concurrent transaction. For example, suppose that the queries Q_1 and Q_2 from Section 1 are run

concurrently without concurrency control. We saw that it is possible for Q_1 to see one, but not both, of Q_2's updates to the Prolog knowledge base, leading to non-serializable behavior. In this example, the problem is due to the concurrent execution of a read-only query (Q_1) and a query that asserts new facts (Q_2), but a similar problem would arise if Q_2 were to retract facts used by Q_1.

Query–Rule Conflicts. This problem arises when one transaction asserts or retracts a rule used by another concurrent transaction. Since we do not address queries which assert or retract rules in this paper, we need not concern ourselves with this type of conflict for now. We will return to this issue when we discuss future work at the end of the paper.

The problem at hand, then, is to prevent fact-fact conflicts and query-fact conflicts through the use of an appropriate concurrency control mechanism. Readers familiar with the concurrency control literature will recognize that these conflicts are similar to the write-write and read-write conflicts of database concurrency control. However, there are some differences as well. The main salient features of the Prolog concurrency control problem have to do with *phantoms* and *parallel subqueries*.

We focus first on the issue of phantoms. All updates to the knowledge base in Prolog occur through the built-in *assert* and *retract* predicates. Thus, in database terms, Prolog programs can insert and delete facts, but they cannot modify existing facts.* As a result, all conflicts in Prolog are instances of the problem known as the *phantom tuple* problem (illustrated in the example in Section 1). As described in [Eswa76] and [Jord81], this problem arises when one transaction calls for a fact, or a tuple in database terminology, which does not (does) exist at the time of the request, but which is later created (deleted) due to the action of another transaction.

We next consider the second difference between database concurrency control and Prolog concurrency control, the issue of parallel subqueries. Consider the following Prolog rule:

```
sibling(X,Y) :- brother(X,Y); sister(X,Y).
```

This rule says that X is the sibling of Y if X is the brother or sister of Y. Suppose we now present the system with the query sibling(john,martha) in order to find out if john is a sibling of martha's. In a Concurrent

*To modify a fact, a Prolog program must retract the old version of the fact and then assert an appropriately modified version.

Prolog implementation, the two sub-queries in the disjunction, brother(john,martha) and sister(john,martha), can be executed concurrently. The same is true for similar conjunctive queries. As we shall see, the possibility of concurrent subqueries will complicate the concurrency control problem somewhat.

2.2. A Solution: Query-Fact Locking

The Prolog concurrency control problem can be dealt with by an algorithm that we call *query−fact locking*. This algorithm is a Prolog-oriented variant of a database concurrency control algorithm known as *precision locking* [Jord81]. Precision locking is based on the popular notion of two-phase locking [Eswa76, Gray79], where transactions set read locks and write locks on the data items that they read and write (respectively) and hold locks until end of transaction. Most database systems apply locks to *physical* objects such as files, pages, or records. Precision locking, on the other hand, sets read locks on groups of *logical* objects, which are specified by predicates, and it applies write locks to physical objects (tuples or records). The algorithm is related to the predicate locking scheme proposed by Eswaren et al [Eswa76], but it is much more efficient (and thus practical) because it avoids the problem of having to decide whether or not a pair of predicates are jointly satisfiable. We will describe the reasoning that led to our selection of precision locking later on, but the basic reason is that the mechanism effectively and efficiently handles the concurrency control problems of interest, including phantoms. Physical locking algorithms do not correctly deal with phantoms [Jord81].

2.2.1. The Basic Algorithm

In query-fact locking, each transaction T_i maintains two sets: a set of queries $\mathbf{Q_i}$, and a set of facts $\mathbf{F_i}$. The set $\mathbf{Q_i}$ is the set of queries $\{Q_{ij}\}$ that the transaction has executed, specifying the readset of the transaction. Read locks will be set on each query in this set, as we will see shortly. The set $\mathbf{F_i}$ is the set of facts $\{f_{ij}\}$ that the query has asserted or retracted, specifying the writeset of the transaction. Write locks will be set on each fact in this set. In addition, the concurrency control algorithm will maintain two global sets, $\mathbf{Q_L}$ and $\mathbf{F_L}$. $\mathbf{Q_L}$ is the global set of queries for which read locks have been granted and not yet released, and $\mathbf{F_L}$ is the set of facts for which write locks have been granted and not yet released. Entries in $\mathbf{Q_L}$ are of the form (Q_{ij}, T_i), where Q_{ij} is the locked query and T_i is the locking transaction. Similarly, entries in $\mathbf{F_L}$ are

of the form (f_{ij}, T_i), where f_{ij} is the locked fact. These sets are equivalent to the sets of predicates and tuples in the precision locking algorithm [Jord81], and the union of the global sets $\mathbf{Q_L}$ and $\mathbf{F_L}$ is the equivalent of a lock table.

We are now almost ready for the details of the query-fact locking algorithm. First, though, we need one definition: We will say that one Prolog query Q_p *covers* another Prolog query Q_c if (1) they have identical predicate names, and (2) every constant or instantiated variable in the i^{th} argument position of Q_c has the same value as the corresponding argument of Q_p if Q_p's i^{th} argument is a constant or an instantiated variable. For instance, child(X,larry) covers child(bob,larry) and child(Y,larry), but it does not cover child(bob,X). Said another way, Q_p covers Q_c if Q_c could arise as a subquery of Q_p through variable instantiation during the execution of Q_p. Note that a special case of this definition is a query covering a fact. We are now ready to proceed with the specification of the query-fact locking algorithm.

2.2.1.1. Fact Assertion or Retraction

Before a transaction T_i can assert or retract a fact f_{ij}, it must do the following:

(1) Add the fact f_{ij} to its fact set $\mathbf{F_i}$.

(2) Check to see if there exists a fact f_{km} in the set of locked facts $\mathbf{F_L}$ such that $f_{ij} = f_{km}$, or if f_{ij} satisfies any query Q_{kn} in the set of locked queries $\mathbf{Q_L}$, where $i \neq k$. If so, Ti must block until f_{km} or Q_{kn} is unlocked.

(3) Add f_{ij} to $\mathbf{F_L}$ and proceed.

Step (1) serves to provide T_i with a list of the facts that it has locked so that they can be unlocked easily at end of transaction. Step (2) checks for fact-fact and query-fact conflicts, with T_i being blocked if either type of conflict is found. Step (3) records T_i's newly granted lock in the global list of locked facts. Steps (2) and (3) must be executed together as an atomic action (i.e., in a critical section).

2.2.1.2. Subquery Execution

Before a transaction T_i can execute a subquery Q_{ij}, it must do the following:

(1) Check to see if there exists a query Q_{im} in its query set $\mathbf{Q_i}$ such that Q_{im} covers Q_{ij}. If so, T_i can skip the remaining steps and simply execute Q_{ij}.

(2) Add the query Q_{ij} to its query set $\mathbf{Q_i}$.

(3) Check to see if there exists any fact f_{kn} in the list of locked facts $\mathbf{F_L}$ such that Q_{ij} covers f_{kn} and $i \neq k$. If so, T_i must block until all such facts f_{kn} are unlocked.

(4) Add Q_{ij} to $\mathbf{Q_L}$ and proceed.

Step (1) checks to see if T_i already has a lock which covers the facts to be dealt with by subquery Q_{ij}. If so, there is no need to lock a subset of these facts. This is an important performance optimization, as a query can lead to potentially many subqueries through variable instantiation. If each such subquery led to an entry in the set of locked queries, query-fact conflict testing could become overly expensive. Step (2) adds Q_{ij} to the list of locked queries for T_i for use at end of transaction. Step (3) checks for query-fact conflicts, blocking T_i if such a conflict is found. Step (4) records Q_{ij} in the global set of locked queries. Steps (3) and (4) must be executed together as an atomic action.

2.2.1.3. End of Transaction

When a transaction T_i terminates, it applies its knowledge base updates (as will be described in Section 3 of the paper), and then releases its locks as follows using the information in $\mathbf{F_i}$ and $\mathbf{Q_i}$:

(1) $\mathbf{F_L} = \mathbf{F_L} - \{T_i, F_{ij}\}$.
(2) $\mathbf{Q_L} = \mathbf{Q_L} - \{T_i, Q_{ij}\}$.

2.2.2. Blocking and Deadlocks

As described above, conflicts are handled by blocking transactions until the conflicts disappear. Thus, in addition to the global query and fact lock sets $\mathbf{Q_L}$ and $\mathbf{F_L}$, the concurrency control mechanism will have to manage queues of waiting transactions, as does the lock manager in a typical database system

[Gray79]. Also, as with most dynamic two-phase locking algorithms, query-fact locking may lead to occasional deadlocks. For example, it is possible for two transactions each to set query locks which cover a set of common facts, and then for each to try to assert or retract a fact included in their joint coverage. Deadlocks can be handled by maintaining a waits-for graph and checking for cycles in the usual way [Gray79, Agra83a].

2.2.3. Query-Fact Locking in Action: An Example

Let us now see how the query-fact locking algorithm solves the problem illustrated by the example in Section 1. As a refresher, the example involved the following collection of facts:

```
child(sue,larry)
child(carol,larry)
child(fred,larry)
child(joe,larry)
```

The set of rules for the example consisted of a single rule:

```
grandchild(X,Y) :- child(Z,Y), child(X,Z).
```

Finally, the two conflicting queries were as follows:

Q_1: grandchild(X,larry).
Q_2: assert(child(john,sue)), assert(child(alice,joe)).

When query Q_1 is executed, the following subqueries and associated query locks will be set (assuming only the facts given above are in the knowledge base):

	Subquery	Query Locked
1.	child(X,larry)	child(X,larry)
2.	child (sue,larry)	
3.	child(X,sue)	child(X,sue)
4.	child(carol,larry)	
5.	child(X,carol)	child(X,carol)
6.	child(fred,larry)	
7.	child(X,fred)	child(X,fred)
8.	child(joe,larry)	
9.	child(X,joe)	child(X,joe)

Steps 2, 4, 6, and 8 illustrate the benefit of the optimization included in the query locking algorithm: If a new query lock is set *every* time a subquery begins to execute, regardless of whether or not the

transaction already holds a covering lock, locks would have been set at each of these steps as well. The result would have been that nearly twice as many locks would have been set as are really needed to achieve the desired concurrency control effect in this example.

When query Q_2 is executed, the following subqueries and associated fact locks will be set:

	Subquery	Fact Locked
1.	assert(child(john,sue))	child(john,sue)
2.	assert(child(alice,joe))	child(alice,joe)

Now let us examine what would happen if the two queries are run concurrently. Suppose that steps 1-3 of Q_1 have completed and now Q_2 begins running. When Q_2 attempts to execute its first step, it will first try to lock the fact child(john,sue). Since Q_1 has already locked child(X,sue), which covers child(john,sue), Q_2 will be blocked until Q_1 completes and releases its locks. What if Q_2 had begun first instead? Q_2 will lock the fact child(john,sue) in its first step, causing Q_1 to block at its step 3 because child(X,sue) covers child(john,sue). Thus, in this case, Q_1 will wait until Q_2 completes and releases its locks. Regardless of the attempted execution order, the query-fact locking algorithm prevents non-serializable behavior by blocking one of the two conflicting queries until the other has finished.

2.2.4. Concurrent Subqueries: A Complication

The execution of a Concurrent Prolog program forms an and-or tree of concurrent conjunctive and disjunctive subqueries. This permits us to optimize our locking protocol a bit, releasing some locks earlier than end of transaction without compromising serializability. Consider the following set of disjunctive subqueries:

f(X,Y,Z) :- f1(X,Y,Z); f2(X,Y,Z); f3(X,Y,Z).

Under the Concurrent Prolog semantics discussed in Section 1, f(X,Y,Z) will be evaluated in a way such that if either of the subqueries are true, the entire query will be true. Beyond this, however, no guarantees are provided. In particular, each subquery sees the knowledge base state prior to the execution of all concurrent subqueries, and only the results of one of the successful (true) subqueries are guaranteed to be applied to the knowledge base. Given these conditions, then, we can allow failed subqueries in a set of disjunctive

subqueries to release their locks. In addition, we can discard the results of all but one of the successful subqueries, and each of the discarded subqueries can also release its locks. We refer to the selected successful subquery as the *selected disjunct*, and the other queries in the disjunction are referred to as *insignificant disjuncts*. All queries which are not insignificant disjuncts (i.e., either conjuncts or selected disjuncts) are referred to as *significant queries*.

These observations lead to the following algorithmic changes: Each subquery Q in a set of concurrent subqueries inherits a copy of the lock sets $\mathbf{F_i}$ and $\mathbf{Q_i}$ from its parent query at the time of the subquery's initiation. It then proceeds to apply the query-fact locking algorithm as described in the previous section. When the subquery completes, if it turns out to be an insignificant disjunct, the recovery manager can discard its results, and all of the locks added to $\mathbf{F_i}$ and $\mathbf{Q_i}$ by this subquery may be released. Otherwise, the parent query's lock sets $\mathbf{F_i}$ and $\mathbf{Q_i}$ must be augmented with any new locks set by the subquery, as these locks must remain set until end of transaction (i.e., the parent query inherits locks from its significant child queries). Readers familiar with the concurrency control literature may notice that our mechanism has a flavor somewhat similar to that of nested transactions [Reed78, Moss81]. A difference is that our subqueries are not required to be serializable with respect to one another, with before-image semantics for concurrent subqueries being provided by the recovery mechanism (see Section 3).

2.2.5. Concurrent Rules

As discussed earlier, concurrent queries that satisfy the same goal can be regarded as a disjunct with an order imposed. Consider, for example, the following predicate specification:

$$\begin{aligned} &p(X) :- s_1(X).\\ &p(X) :- s_{i-1}(X).\\ &p(X) :- s_i(X).\\ &p(X) :- s_{i+1}(X).\\ &p(X) :- s_n(X). \end{aligned}$$

To evaluate p(X), all $s_k(X)$ are evaluated concurrently. If one of them returns with a success, say s_i, then the subqueries s_{i+1} to s_n are treated as insignificant disjuncts, whether or not they have already produced a result, but the subqueries s_1 to s_{i-1} must be continued. If the result of s_i is a failure, then this subquery is treated as an insignificant disjunct. In terms of subquery completion, this means that the first rule is always

completely evaluated and the other ones are evaluated only if all preceeding ones failed or a preceeding one took longer to execute.

2.2.6. Backtracking

During backtracking the effects of backtracked subqueries must be undone. However, locks are not released during backtracking. The reason for not releasing locks is that the database conditions that caused the query to backtrack must still hold at end-of-transaction to ensure serializability.

2.3. Other Solutions Considered

In the process of designing the Prolog concurrency control mechanism just described, we considered and ruled out a number of other alternatives. In this section we briefly consider each such alternative in turn, explaining what led us to eliminate it. The alternatives considered include locking facts, locking predicate names, or using a non-locking-based algorithm such as optimistic concurrency control [Kung81].

Our first inclination was to set locks on individual facts. However, we quickly realized that this would not work, as this would not solve the phantom fact problem. In particular, having transactions lock facts read and written would not prevent the transactions in the example from Section 1 from interleaving in a non-serializable way — Q_1 can lock the facts that it reads, but it cannot possibly set read locks on facts before they are asserted.

Our second inclination was to set locks on predicate names, such as child in the example. This will work, but it has terrible performance implications: This solution is equivalent to locking entire relations in a relational database system, which is an unacceptably coarse granularity for locking. As an illustration, suppose that the set of child facts included information about 10,000 children, yet the only child facts relevant to queries Q_1 and Q_2 are the six facts which they deal with. Since transactions hold locks until end of transaction (with a few possible exceptions as outlined in the previous section), other transactions would have to block for potentially long periods of time. In the example, *no* other transactions would be able to access *any* of the child facts while Q_1 or Q_2 execute.

Having considered these two possibilities, we came to the realization that some form of predicate-like locking would be necessary. A desire for a practical mechanism led us to select precision locking

[Jord81] rather than true predicate locking [Eswa76] as a starting point.

We also considered other forms of concurrency control, such as optimistic algorithms [Kung81] or timestamp-based algorithms [Bern81]. Using either type of algorithm on physical objects, such as facts or predicate names, leads to the same sorts of problems as the rejected locking algorithms that we considered. However, it would be possible to use such algorithms in conjunction with queries and facts, as is done in query-fact locking. For example, if one wished to use the serial validation algorithm for optimistic concurrency control [Kung81], one could use $\mathbf{Q_i}$ as the readset for a transaction T_i, use $\mathbf{F_i}$ as its writeset, and validate T_i at commit time by checking that no recently committed transaction T_j had a writeset of facts $\mathbf{F_j}$ which is covered by any query in $\mathbf{Q_i}$. If T_i passed this validation test, it would be committed, its updates would be applied to the knowledge base, and $\mathbf{F_i}$ would be saved for use in validating future completing transactions. If not, it would be restarted. Thus, an optimistic algorithm would work equally well, at least in a logical sense. However, we chose locking over alternative mechanisms for performance reasons: A recent study has indicated that it is better to use locking algorithms instead of restart-oriented algorithms if conflicts are fairly likely, and that it does not matter what sort of mechanism is employed if conflicts are rare [Care84]

2.4. Summary

We have outlined a concurrency control mechanism which seems to nicely fit the concurrency control needs of Prolog. The mechanism, query-fact locking, is a form of two-phase locking in which transactions set read locks on queries and set write locks on facts. We expect that this mechanism can be implemented efficiently, as it is based on a notion of covering which is efficiently testable. The mechanism seems likely to perform at least as well as a physical locking mechanism would, as argued in the original precision locking paper [Jord81], and physical locking would not handle phantoms correctly. Since all conflicts in Prolog are instances of the phantom problem, and since query-fact locking correctly deals with this problem, we believe that our proposal is a promising one.

3. Recovery

In this section we propose a recovery mechanism for Prolog. The section begins with a discussion of the recovery problems posed by Prolog. Next we describe the operation of our proposed recovery mechanism. Finally, we discuss why we feel that it is the most appropriate recovery mechanism for the Prolog environment.

3.1. The Problem

A recovery mechanism for Prolog must provide three fundamental services. First it must make the effects of Prolog queries that update the database durable. By durable we mean that once a query commits its changes to the database will not be affected by subsequent hardware and/or software failures. The second service is to undo any changes to the database that are made by aborted queries. While both these services are found in conventional database systems, the recovery manager for Prolog must additionally provide a mechanism for undoing the effects of subqueries that fail in the process of answering a query. In the following section we will propose a mechanism that provides these three services.

3.2. Recovery using Differential Files

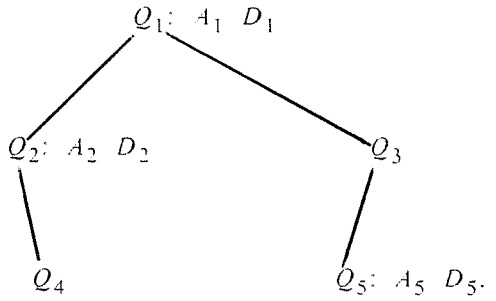
With the differential file scheme proposed in [Seve76], each logical file consists of two physical files: a read-only *base file* and a read-write *differential file* containing all changes to the file. The base file remains unchanged until reorganization. All updates are confined to the differential file. In [Ston80, Ston81, Wood83], Stonebraker extended the differential file idea to introduce the notion of hypothetical relations. Each relation R in the database consists of three parts: B, a read-only base portion of R, A, a file containing all additions to R, and D, a file containing all tuples deleted from R. From the point of view of a query, R is equal to (B $\cup$ A) - D. Recovery from software and/or hardware failures is simplified as A and D are both append-only files. A and D are merged with R only during database reorganization.

3.3. A Recovery Mechanism for Prolog

We propose to use the notion of hypothetical relations as a basis for a recovery mechanism for Prolog. For the duration of a query (or a group of queries designated as a transaction), the knowledge base,

KB, is treated as a read-only file. Whereas the A and D files in Stonebraker's hypothetical relation mechanism are permanently associated with a relation R, we propose to instead associate A and D files[1] with each query and to merge A and D with KB when the query commits. The A file is used to hold facts (and rules) asserted by the query and the D file is used to hold facts (and rules) which the query proposes to retract from the knowledge base.

Execution of a query may actually cause a number of A and D files to be created. Empty A and D files are created when a query first begins execution. In addition, whenever a subgoal is executed which asserts or retracts one or more facts, an additional pair of A and D files are created. This is illustrated by the following example:



In this example, queries Q_1, Q_2, and Q_5 assert or retract one or more facts that are stored in the A and D files local to the corresponding goal. From the viewpoint of goal Q_4, the knowledge base corresponds to $((((KB \cup A_1) - D_1) \cup A_2) - D_2)$. For Q_5, it is $((((KB \cup A_1) - D_1) \cup A_5) - D_5)$. A_5 and D_5 are empty until the goal asserts or retracts a fact. This approach quite obviously generalizes to queries at arbitrary depths.

When a subgoal succeeds, its associated A and D files, if any, are appended to those of its parent. If the parent does not have A and D files associated with it, it inherits those of the first child to succeed.

When a subgoal fails, two alternative courses of action are possible. The first is to delete its associated A and D files. This strategy preserves the undo semantics for subqueries that fail as assumed in Section 1. If this is not acceptable, the following alternative is feasible. First, a second set of files, Undo-A and Undo-D, is created whenever a set of A and D files is created. Whenever a goal fails, the union of the A file is formed with the A and Undo-A files of its parent. The D file is processed in the same manner. In

[1] A and D are probably more properly referred to as storage structures since they generally will reside in main memory

addition, the Undo-A and Undo-D files of the goal are combined with the Undo-A and Undo-D files of its parent. In this way, all assertions and retractions are visible for the duration of the query. However, before committing the effects of the query we form A = A - Undo-A and D = D - Undo-D in order to undo the effects of those subqueries which failed during the execution of the query.

When a query backtracks, the effects of its subqueries are undone by truncating those portions of its A and D files that were appended when its subqueries terminated.

Committing the effects of a transaction (query) requires updating the knowledge base by forming KB = (KB ∪ A) - D in such a manner that either all of the changes made by the query are reflected in KB or none are. The following algorithm can be used:

(1) Write A and D to stable storage [Gray79, Lamp79]
(2) Write a pre-commit record to stable storage. This record should contain the id of the query and pointers to the A and D files.
(3) Update the copy of KB on disk using the copies of A and D still in primary memory.
(4) Write a commit record to stable storage. This record contains the id of the query.

Once step 2 has been completed, the recovery software insures that the updates made by the query will eventually be applied to the knowledge base. If the system crashes during step 3, then the fact that the commit record for the query does not appear in stable storage will be noted during system restart and step 3 will be repeated.[2]

Finally, if a transaction is aborted by the user or the system, undoing its effects can be accomplished by simply deleting all its A and D files.

3.4. Discussion

Our proposed mechanism appears to provide a robust yet efficient recovery mechanism for Prolog. By structuring the A and D files in the same way as the KB, existing software can be used. Furthermore, $KB \cup A$ and $(KB \cup A) - D$ do not ever have to actually be materialized.

Generally, the proposed recovery mechanism preserves the semantics for use in a sequential Prolog system, while providing a reasonable set of semantics for a Concurrent Prolog environment. In

[2] See [Gray79] for a discussion on how idempotence can be achieved.

particular, by utilizing before-image semantics for parallel sub-queries, we can eliminate concurrency problems having to do with parallel sub-queries. (It is important to note that we do *not* demand or provide serializability for these sub-queries, as our before-image semantics prohibit it.)

If queries instead used in-place updates and logging [Gray79] as the recovery mechanism, updates by parallel subqueries would be visible to one another, leading to nonderministic results. Furthermore, since subqueries frequently fail, the cost of undoing updates made by these subqueries by reading the log from stable storage would be prohibitively high.

4. Conclusions

This paper has proposed concurrency control and recovery mechanisms which are designed specifically for use in a Prolog environment. The proposed concurrency control mechanism, query-fact locking, handles fact-fact and query-fact conflicts that can arise when concurrently executed Prolog transactions share a knowledge base. The mechanism requires transactions to set read locks on queries and write locks on facts to guarantee serializable behavior. In addition, a lock inheritance mechanism was outlined for dealing with concurrent subqueries in Concurrent Prolog programs. We expect that our algorithm will be reasonably efficient, as it is based on an easily-testable notion of queries covering other queries and facts.

The proposed recovery mechanism is based on a differential file scheme known as hypothetical databases. The scheme can handle normal sequential Prolog programs, sequential Prolog programs where the results of failed subqueries are to be backed out, and Concurrent Prolog programs where before-image semantics are desired for concurrent subqueries. The proposed scheme uses multiple levels of hypothetical relations to handle subqueries in Prolog programs, though a single set of hypothetical relations would be sufficient to handle the case of normal sequential Prolog programs.

One way in which this work can be extended is to handle rule updates as well as fact updates on a dynamic basis. This appears to be a simple extension: Rule assertions and retractions lead to two new types of conflicts, *rule* − *rule* conflicts and *query* − *rule* conflicts. Prolog transactions can be augmented with sets of updated rules, $\mathbf{R_i}$, analogous to their sets of updated facts. The global concurrency control information can be augmented with a set of locked rules, $\mathbf{R_L}$, analogous to the set of locked facts. These sets can then be

used to check for conflicts when subqueries are executed and when rules are asserted or retracted. The conflict-checking algorithm will be similar to the algorithms for checking for fact-fact and query-fact conflicts, except that the covering tests can be replaced by a simple test for matching predicate names in this case. As for recovery, rule updates can be recorded in the same files as the fact updates in the current scheme.

In summary, this paper has presented a fairly detailed proposal for adding concurrency control and recovery facilities to Prolog. These additions are necessary if Prolog is to someday be truly useful as a vehicle for the creation of practical, commercial, intelligent database management systems. The combined effect of our proposed algorithms for concurrency control and recovery is a mechanism for correctly and reliably executing either sequential Prolog programs or Concurrent Prolog programs in a multi-user environment with a shared knowledge base.

References

[Agra83a] Agrawal, R., Carey, M., and DeWitt, D., "Deadlock Detection is Cheap", *ACM SIGMOD Record* 13(2), January 1983.

[Agra83b] Agrawal, R., and DeWitt, D., "Updating Hypothetical Databases", *Information Processing Letters* 16(3), April 1983.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys* 13(2), June 1981.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proceedings of the Tenth International Conference on Very Large Data Bases*, August 1984.

[Cloc81] Clocksin, W., and Mellish, C., *Programming in Prolog*, Springer-Verlag, 1981.

[Dahl82] Dahl, V., "On Database Systems Development Through Logic", *ACM Transactions on Database Systems* 7(1), March 1982.

[DBE83] *Database Engineering* 6(4), Special Issue on Expert Systems and Database Systems, December 1983.

[Eswa76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM* 19(11), November 1976.

[Gray79] Gray, J., "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

[Jark84] Jarke, M., Clifford, J., and Vassiliou, Y., "An Optimizing Prolog Front-End to a Relational Query System", *Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data*, June 1984.

[Jord81] Jordan, J., Banerjee, J., and Batman, R., "Precision Locks", *Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data*, May 1981.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems* 6(2), June 1981.

[Lamp79] Lampson, B., and Sturgis, H., *Crash Recovery in a Distributed Data Storage System*, Research Report, Xerox PARC, 1979.

[Moss81] Moss, E., *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1981.

[Pars83a] Parsaye, K., "Database Management, Knowledge Base Management, and Expert System Development in Prolog", *Proceedings of the ACM-SIGMOD Database Week Conference (Databases for Business and Office Applications)*, May 1983.

[Reed78] Reed, D., *Naming and Synchronization in a Decentralized Computer System*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Seve76] Severence, D., and Lohman, G., "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Transactions on Database Systems* 1(3), September 1976.

[Shap83] Shapiro, E., *A Subset of Concurrent Prolog and Its Interpreter*, Technical Report TR-003, ICOT, January 1983.

[Ston80] Stonebraker, M., and Keller, K., "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System", *Proceedings of the 1980 ACM-SIGMOD International Conference on Management of Data*, May 1980.

[Ston81] Stonebraker, M., "Hypothetical Data Bases as Views", *Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data*, May 1981.

[Wood83] Woodfill, J., and Stonebraker, M., "An Implementation of Hypothetical Relations", *Proceedings of the Ninth International Conference on Very Large Data Bases*, August 1983.