22
Triggers and Inference In Database Systems

Michael Stonebraker[1]

ABSTRACT *There is a collection of database applications (such as real time control) which may be best accomplished using collections of triggers. The paradigm in which an initial action recursively triggers dependent actions is often called forward chaining. In addition, database support for large knowledge bases requires at least a simple inferencing capability. When a retrieve command cannot be satisfied using only stored data, a data manager must determine if a rule in the knowledge base can be used to reformulate the query. In this way, one works from the desired data toward database facts which must be ascertained using backward chaining.*

In this paper we show how forward chaining and backward chaining can both be supported by simple extensions to a relational query language. Moreover, we demonstrate extensions to a conventional lock manager which will efficiently implement the new constructs. Lastly, the extensions to support backtracking are described.

1. Introduction

In real time control applications such as power plant monitoring and control of sonar systems, a collection of sensors periodically (or continuously) present data to a database system. When certain data patterns are sighted, appropriate actions must be taken. For example, if the flow of coolant past a given sensor falls below a certain value, then

[1] University of California, Berkeley, CA.

an auxiliary pump should be activated. One way to implement such applications is through a collection of trigger [ESWA75] or alerters [BC79]. In business data processing applications, triggers may provide a useful mechanism for propagating updates to dependent (or replicated) data elements. (However, there are many who dispute this point of view).

In a trigger driven application an initial update to the database will cause dependent updates, which may recursively cause further updates. This forward chaining process quiesces when there are no new dependent triggers.

Conventional database management systems (DBMS) are useful for efficiently storing large quantities of rigidly formatted data. However, when DBMS's are applied in applications containing both large amounts of data and "knowledge", they usually fail to provide facilities to manage the knowledge-based portion. Applications which have a knowledge base generally require the capability to store rules of the form:

if (condition) then (result)

For example:

if a person is a manager
then he has a key to the executive lounge

Moreover, such applications require automatic inference on the rules in the knowledge base. For example, if a user asks:

Does John have a key to the executive lounge?

then the data manager must detect that the information is not stored in the database, that there is an appropriate rule to consult, and that the desired query is:

Is John a manager?

The process whereby a desired retrieval iteratively causes auxiliary retrievals until one yields an appropriate result is called backward chaining, and is the basic control tactic used in Prolog [CM81] and many other artificial intelligence (AI) languages.

When multiple rules can be applied, then backtracking is often a useful feature. Suppose there is an additional rule:

if a person is disabled
then he has a key to the executive lounge

If John is not a manager, the system should backtrack to determine whether John is disabled. Backtracking on failure is a common control tactic in AI languages, and can be applied in both forward chaining and backward chaining systems.

In this paper simple extensions to a relational query language QUEL [STON76] are proposed to support the storage of rules and both forward and backward chaining as control strategies. As such capabilities similar to those in Planner [HEWI71] can be integrated into a data base management system. Moreover, simple extensions to a lock manager to efficiently implement the language proposals are suggested. Section 2 treats forward chaining and its proposed locking implementation, while Section 3 discusses backward chaining. Then, Section 4 indicates how to extend both facilities to support backtracking, while Section 5 presents two alternative locking systems which can support the necessary locking constructs. In the next section the proposed mechanisms are contrasted with a alternate implementation tactic based on views. Lastly, Section 7 contains some concluding remarks.

Even though the context for our proposal is the data sublanguage QUEL and the INGRES database system, the ideas can be easily used in any relational DBMS with minor modifications. Although an expert system example might be a more appropriate vehicle for illustrating our constructs, in the interest of simplicity and brevity the standard EMP relation:

EMP (name, salary, age,dept, manager)

will be used instead. Other work on coupling knowledge bases and databases is described in [BJ84, CW84, JCV84, MW84, SW84, WARR81].

2. A Proposal For Triggers

2.1. Language Constructs

A QUEL command such as:

```
range of E is EMP
replace EMP (salary = E.salary)
where EMP.name = "Mike"
and E.name = "Bill"
```

will set Mike's salary equal to that of Bill. The command can be rerun at any time to reperform the replacement.

In order to turn a QUEL command into a trigger, it must logically execute indefinitely. We propose the following syntax:

```
range of E is EMP
replace ALWAYS EMP (salary = E.salary)
```

```
where EMP.name = "Mike"
and E.name = "Bill"
```

This command sets Mike's salary equal to Bill's salary, and then logically continues to execute indefinitely. Hence, whenever Bill's salary is changed, the new value will immediately propagate to Mike's salary. Using this feature, any QUEL operation, postpended with ALWAYS, becomes a trigger.

2.2. Support For Triggers

Any ALWAYS command can be processed with the assistance of a special kind of lock, called a "trigger-me" lock (T lock). The compatibility between T locks and normal read and write locks is specified in the following table:

	R	W	T
R	ok	no	ok
W	no	no	##
T	ok	no	ok

An ALWAYS command is executed repeatedly by the user's INGRES process until it no longer has an effect. Then, INGRES reexecutes the command, and sets T locks on all the database objects read or written by the command (i.e. all the objects satisfying the where clause). Then the command is placed in a relation:

DORMANT (t-id, quel-command, user)

When a user U, submits an INGRES command which attempts to write an object on which a T lock is held (case ## above), U's INGRES process obtains the requested lock and continues processing. In parallel the lock manager releases all T locks held by the command in the DORMANT relation and activates it using the stored user-id to achieve access control. The only required DBMS facility is the ability to submit a QUEL command from inside the lock manager.

All commands which hold a T lock on the object written by the user command are awakened. In this way, the DBMS performs a (logically parallel) breadth-first exploration of tree of dependent commands. Section 4 will present an alternate exploration method using a depth-first search and backtracking.

Triggers can be canceled by running an INGRES delete command on the DORMANT relation. At this time, all T locks held by the command must be removed. Alternatively, one could extend QUEL with a

CANCEL command as follows:

CANCEL trigger-id

In this case, the user must be informed of the trigger-id for later use in a CANCEL command.

Of course, T locks must be persistent (i.e., survive crashes of the hardware or DBMS). Moreover, very fine granularity locks (on records or even on fields of records) will be helpful in avoiding unnecessary "wake-ups" of triggers caused by updating some other object inside a lockable granule. In addition, lock escalation is desirable to prevent triggers which read many data elements from setting an enormous number of fine granularity locks. The locking system must also be carefully designed to deal correctly with "phantoms" [GRAY78]. Not only must the locking system guarantee that an update to any qualifying object will awaken the trigger, but also it must guarantee that any insertions to the database that happen to qualify will also awaken the trigger. Such insertions are called phantoms. For example, the insertion of a new employee with a name equal to "Mike" must cause the example ALWAYS command to be awakened. Section 5 presents two implementation alternatives which satisfy these goals.

The above facilities will set Mike's salary equal to Bill's whenever Bill's salary is changed, and in addition, if Mike's salary is inadvertantly changed, it is immediately reset to Bill's value. In some circumstances one might not want a direct update to Mike's salary to be undone in this fashion. Such an effect can be accomplished by a different modifier on QUEL commands (e.g., WEAKLY) and trigger-me locks which are held on objects which are read but not written. Unfortunately, the phantom problem appears difficult to solve with WEAKLY commands.

The above trigger system is based primarily on QUEL, and requires no new syntax or query processing extensions. Also, it is as efficient as the fineness of lock granularity of the locking system. Moreover, only modest changes are required to a DBMS to support ALWAYS commands and T locks. Lastly, no theorem prover or exotic data structures are required to identify which commands to trigger; the lock manager simply identifies W-T lock conflicts.

ALWAYS commands support one form of inferencing. For example, a database update can trigger a command in the DORMANT relation. This awakened command can trigger a third, and so forth. This forward chaining will stop when no new triggers are awakened by active commands. In this way, one can find all consequences of a particular update. However, the above mechanism is not able to perform backward chaining from a desired goal to a set of facts. The mechanism in the next section supports this alternate construct.

3. Backward Chaining

3.1. Language Constructs

Consider another modifier to QUEL commands, DEMAND. For example,

```
range of E is EMP
replace DEMAND EMP (hair = E.hair)
where E.name = "Bill"
and EMP.name = "Mike"
```

The normal meaning of this command is to make Mike's hair color the same as Bill's. However, DEMAND commands are not directly executed; rather, a form of "lazy evaluation" is used. When someone requests the hair color of Mike, he is made aware of the effect of the DEMAND command in a way to be described. DEMAND commands are called "lazy triggers," and require a slight modification to the relational model.

A normal relation consists of tuples, each with a collection of stored fields. In the EMP relation these are name, salary, age, dept, and manager. In addition, lazy triggers can provide data values for columns which are not stored (e.g., hair color in the EMP relation). Hence, each tuple in a relation has a collection of stored fields and a collection of fields with values provided by lazy triggers. Lazy triggers need not provide a value for a given field for each tuple in a relation. Consequently, the unstored field can vary from tuple to tuple. One view of lazy triggers is as a means of extending a relation with new fields which have values only for a subset of the tuples.

3.2. Support for DEMAND Commands

A DEMAND command is executed until the collection of objects which it will update is determined. Then execution is halted without any modification to the database, and the command is stored in a DORMANT-2 relation

DORMANT-2 (c-id, command, user-id)

Additionally, "Missing Data" locks (M locks) are set on the collection of objects which the command would have updated. The compatibility matrix for M locks is the following: When a read lock is set on an object which has an M lock set (case !! below), an R-M conflict algorithm is run. M locks are held indefinitely and are only withdrawn when the corresponding DEMAND command is deleted.

	R	W	M
R	ok	no	!!
W	no	no	no
M	no	no	ok

QUEL processing is slightly different if DEMAND fields are present. The regular parser will reject a command which requests data which does not appear as a stored field in a relation. This rejection must be delayed, because there may be a lazy trigger to provide the desired data. An execution plan is then carried out and a retrieval command is decomposed into a collection of single relation subcommands, S, of the form:

range of R is relation
retrieve (R.t1, ..., R.tj)
where Q(R).

Here, Q(R) is a qualification involving only the tuple variable R, and the target list contains a collection t1, ..., tj of fields of R. The DBMS will acquire locks during the processing of each S.

If S requests a read lock on an object which has a M lock set, the R-M conflict algorithm below must be executed. In this algorithm the DEMAND command D, holding the M lock is of the form:

replace DEMAND X (d1 = f1, ..., dn = fn)
where QUAL

Hence, D updates data items d1, ..., dn with values computed using the functions, f1, ..., fn whenever the qualification QUAL, is true. Also, X is the tuple variable which specifies the relation to update.

R-M Conflict Algorithm

1. Make a copy of the query S and delete from the qualification all clauses which have already been evaluated to true for the current record. Substitute into the query any data items which are stored in the current record, and call the resulting query S'. It will contain only references to non-stored data items. If D does not provide values for any data items, then terminate the algorithm, and continue query processing for S on the next DEMAND command D' which has an M lock on the current record, or on the next appropriate tuple.
2. In D replace each occurence of the tuple variable X by R. Add all range statements of D to those of S. Create S'' by replacing every reference to R.dj in S' by fj and adding the qualification QUAL

from D.

3. Execute the modified query S'' normally and return any qualifying tuples produced as part of the result for S. Continue query processing for S on the next DEMAND command with an M lock on the current record, or on the next appropriate tuple.

For example, suppose the user requests the hair color of Mike:

```
retrieve (EMP.hair) where EMP.name = "Mike"
```

Moreover, suppose the following two lazy triggers are in effect:

```
     range of E is EMP
     range of F is EMP
D1:  replace DEMAND E (hair = F.hair)
     where E.name = "Mike"
     and F.name = "Bill"

     range of G is EMP
D2:  replace DEMAND G (hair = "green")
     where G.name = "Bill"
```

The request for Mike's hair color will collide with the M lock held by D1. At this time the clause "EMP.name = "Mike"" has been evaluated. Hence, the remainder of the query, S', is simply:

```
retrieve (EMP.hair)
```

The algorithm will use D1 to produce the following command:

```
range of F is EMP
retrieve (F.hair)
where EMP.name = "Mike"
and F.name = "Bill"
```

(A smarter algorithm would avoid the redundant check for Mike's name, which is guaranteed to be true at this point.) This second retrieve command is run normally and will collide with the M lock held by D2. When the collision occurs, the system will be executing the subquery:

```
retrieve (F.hair)
where F.name = "Bill"
```

and will have evaluated "F.name = "Bill" " to true. Hence, the algorithm will be run again to produce:

```
retrieve (hair = "green")
where F.name = "Bill"
```

This query will return "green", which will be iteratively returned to the top level query and added to the answer. Since there are no other

qualifying records, the query processing plan will exit, producing a single answer "green" as the result of the query.

The advantage of this scheme is that little new syntax is needed. If fine granularity locking is supported, the R-M conflict algorithm will be rarely activateed unless the lazy trigger affects the request. Hence, the scheme should be very efficient. Also, there is no need for a special indexing structure for DEMAND commands to identify relevant lazy triggers.

This mechanism will implement backward chaining because a retrieve command will activate those lazy triggers which provide required data. These in turn may activate other lazy triggers which will ultimately retrieve facts from the data base. This backward chaining is similar to that accomplished by PROLOG; however, it does not support backtracking. If there are two or more rules that apply at a given point, all are executed in a random order. The next section extends the previous constructs to support backtracking.

4. Priorities and Backtracking

4.1. Introduction

Suppose a general rule with some exceptions is desired. For example, all employees over 40 have a wood desk and others have a steel desk. However, Bill who is 45 has a steel desk, Mike who is 35 has a wood desk, and Sam has the same kind of desk as Bill. Using the proposed inference system this can be expressed as five DEMAND commands:

```
D1: replace DEMAND EMP (desk = "wood")
    where EMP.age > 40
D2: replace DEMAND EMP (desk = "steel")
    where EMP.age < = 40
D3: replace DEMAND EMP (desk = "steel")
    where EMP.name = "Bill"
D4: replace DEMAND EMP (desk = "wood")
    where EMP.name = "Mike"
    range of E is EMP
D5: replace DEMAND EMP (desk = E.desk)
    where EMP.name = "Sam"
    and E.name = "Bill"
```

Two problems must be solved. First, a priority system must be devised to support evaluating the last three DEMAND commands in preference

to the first and second. Second, only if a higher priority DEMAND command fails should a lower priority command be executed. For example, to determine the kind of desk that Sam has:

retrieve (EMP.desk) where EMP.name = "Sam"

command D5 and then D3 should be utilized, resulting in the answer "steel." However, if Bill is not an employee, these commands will return a null answer. In this case one should "backtrack" and try a lower priority command (D1 or D2 depending on Sam's age). In no case should all applicable DEMAND commands be used, since this will produce the answer "wood, steel" if Bill is an employee and Sam is under 40.

A similar situation exists with triggers. If two or more triggers apply, one might want to execute the one with highest priority. If that trigger fails to produce a desired answer, the DBMS should backtrack and try a lower priority trigger.

For example, suppose new employees are inserted into the EMP relation with no department specified. They are then assigned to a department using the following triggers:

```
T1:  replace ALWAYS EMP (dept = "shoe")
     where EMP.dept = null
     and EMP.age > 40

T2:  replace ALWAYS EMP (dept = "admin")
     where EMP.dept = null
     and EMP.manager = "Smith"

     range of E is EMP
T3:  replace ALWAYS E (dept = EMP.dept)
     where EMP.name = "Bill"
     and E.name = "Mike"
     and E.dept = null

T4:  replace ALWAYS EMP (dept = "trainee")
     where EMP.dept = null
```

The desired effect is to place new employees managed by Smith in the admin department, new employees over 40 not managed by Smith in the shoe department, Mike in the same department as Bill, and everyone else in the trainee department.

To achieve the desired effect, the triggering process should stop when a desired goal (in this case assigning a department to a new employee) succeeds. Moreover, all four triggers apply to a new employee over 40 named Mike who works for Smith; hence a priority system is required to activate the triggers in the order T3, T2, T1, T4.

Lastly, there may be situations where a collection of triggers have been executed, no new ones have been activated, and the goal has not been reached. In this case, the system should backtrack (i.e., undo the effect of one or more triggers) and try a lower priority alternate collection of triggers.

Consider the situation of hiring a new employee named Sam. First Fred must be moved to the toy department to make room for Sam. Moreover, George must be transferred from the toy department to the admin department to make room for Fred. If one of these changes fails (for example because George does not work in the toy department) then Sam must be placed in the trainee department and Fred restored to his original position. This situation can be expressed by the following collection of triggers:

```
        range of E is EMP
T1:     replace ALWAYS EMP (dept = "toy")
        where E.name = "Sam"
        and E.dept = null
        and EMP.name = "Fred"

        range of E is EMP
T2:     replace ALWAYS EMP (dept = "admin")
        where E.dept = "toy"
        and E.name = "Fred"
        and EMP.dept = "toy"
        and EMP.name = "George"

        range of E is EMP
T3:     replace always E (dept = "shoe")
        where E.dept = null
        and E.name = "Sam"
        and EMP.name = "George"
        and EMP.dept = "admin"

T4:     replace ALWAYS EMP (dept = "admin")
        where EMP.name = "Sam"
        and EMP.dept = null
```

In this case triggers T1, T2, and T3 will be executed in order to produce the desired effect. However, if trigger T2 or T3 fails, their effects should be backed out, and trigger T4 executed instead.

The mechanisms for DEMAND and ALWAYS commands are slightly different, and we consider them separately.

4.2. Support for DEMAND

Consider an extra modifier for DEMAND commands:

```
replace DEMAND PRIORITY EMP (desk = "wood")
where EMP.name = "Mike"
```

To use priorities and backtracking, the keyword PRIORITY must be added to all potentially conflicting DEMAND commands. The effect of a PRIORITY command is similar to an ordinary DEMAND command, except it activates the following prioritization scheme.

Although priorities can be "hardwired", a flexible system is probably more useful. When a PRIORITY DEMAND command is inserted, the database system can easily ascertain which other PRIORITY DEMAND commands conflict with the new one and on what objects the conflict occurs. (This information is available from the lock manager). The user would be required to specify for each such object the priority of his command relative to conflicting commands. This could be done by making insertions into a relation:

PRIORITY-M (id-of-higher, id-of-lower, object)

The composition of PRIORITY-M for the desk example of the previous section would be:

PRIORITY-M	id-of-higher	id-of-lower	object
	5	2	Sam
	3	2	Bill
	4	1	Mike

Using this priority information, the lock manager can form an ordered list by priority of command-ids for each object. In the R-M conflict algorithm, the highest priority conflicting command must be used first.

The user must specify backtracking with the keyword PRIORITY on a retrieve command, for example:

retrieve PRIORITY (EMP.desk) where EMP.name = "Sam"

With this modifier, the R-M conflict algorithm must be altered slightly:

Whenever the algorithm says:

```
continue on the next command C' which holds
an M lock on the current record
```

substitute

```
continue on the next command ONLY
IF THE CURRENT COMMAND PRODUCED
AN EMPTY ANSWER
```

The modified R-M algorithm, the use of a RETRIEVE PRIORITY command, and the insertion of DEMAND PRIORITY lazy triggers will provide the desired priority-backtracking scheme.

4.3. Support for ALWAYS

To use priorities and backtracking, an ALWAYS command must specify the keyword PRIORITY:

```
replace ALWAYS PRIORITY EMP (dept = "shoe")
where EMP.dept = null
and EMP.age > 40
```

When a user inserts such a trigger, the lock manager can ascertain conflicting triggers and the collection of conflicting objects. Registration of a trigger includes inserting information into a PRIORITY-T relation similar to the PRIORITY-M relation above. The lock manager uses the PRIORITY-T relation to order all conflicting T locks. On a W-T conflict, the highest priority trigger is awakened first.

The user must indicate that he desires a priority backtracking trigger solution. At the time he makes an update that will activate triggers, he must specify the keyword PRIORITY, for example:

```
append PRIORITY to EMP (name = "Mike",
        age = 25, salary = 2000)
```

Additionally, he must specify what goal should stop the priority/backtracking algorithm. A simple solution would be to have a built-in goal which is satisfied if a trigger activated no new dependent triggers but modified the database. To obtain greater generality, we propose that a user will will state both his triggering update and his goal in a transaction. The goal is a retrieve command with the keyword PRIORITY which is reached when the command has a non-empty answer. The following transaction has a goal of placing Mike in a department:

```
begin transaction
     append PRIORITY to EMP (name = "Mike", ...)

     retrieve PRIORITY (EMP.dept)
     where EMP.name = "Mike"
end transaction
```

The following three modifications to DBMS processing must be used in the presence of PRIORITY retrieves and updates:

1. A savepoint [GRAY78] must be established for the current transaction before any trigger is activated as a result (either directly or through a chain of intermediate triggers) of a PRIORITY update.
2. Only the highest priority trigger is activated.
3. If an empty answer to a PRIORITY retrieve command is observed and there are no triggers still processing, then backup to the last transaction savepoint, and have the lock manager activate the next trigger in priority order.

The use of ALWAYS PRIORITY commands, the use of transactions with a PRIORITY update followed by PRIORITY retrieve, and modest changes to QUEL processing will implement a general priority-backtracking scheme. The highest priority trigger will be activated and cause a sequence of forwardly chained actions. If the goal is not achieved, the system will backtrack (by undoing changes back to a savepoint) and recursively try the next lower priority trigger, until a solution is found or until all applicable triggers are exhausted.

5. Implementation of T and M Locks

A straight-forward approach would be to place T and M locks in the same lock table holding R and W locks. In this case, one must cope with a lock table of widely varying size which must be made recoverable. Moreover, phantoms must be correctly handled. Lastly, all triggers must run inside the scope of the transaction which contained the initial update, and will thereby be backed out if the transaction is aborted.

The first objective can be satisfied by using extendable hashing for the lock table instead of conventional hashing. The second objective requires writing T and M locks into the log as part of the process of registering a new DEMAND or ALWAYS command. Moreover, the lock table must be checkpointed along with ordinary data. Database recovery code can now restore T and M locks after a crash. This

presents only modest implementation difficulties.

The phantom problem poses more serious issues. Systems which perform page level locking (e.g., [RTI84, CHEN84]) have few difficulties supporting correct semantics in the presence of phantoms. However, finer granularity locking is required for efficient T and M locks. Systems which perform record level locking can allow detection of phantoms by holding locks on index intervals in the leaf nodes of secondary indexes as well as on data records [ASTR76]. Hence, a transaction which modifies a tuple will hold a write lock on the tuple and on the appropriate index interval for any modified field for which a secondary index exists. An insert will, of course, add a new record and associated index entries. Such an insert must wait if it will fall in a locked index interval. Of course, a transaction which splits a B-tree index page must wait until there are no locks held on any index records in the page.

The general mechanism can be restated as one of holding record level locks on data and index tuples. Then, a write must be delayed if it will fall adjacent to a tuple which is locked. Adjacency means "logically adjacent in Tuple Identifier order" for B-tree data records and indexes; adjacency means "in the same hash bucket" for hashed records and indexes.

The same adjacency tactic can be applied to T and M locks. T locks must be held on data and index records, and a trigger will be awakened if a write lock is set on the adjacent index record or data record. Moreover, a DEMAND command will hold M locks on data records. If a write lock is set on an adjacent data record, the DEMAND command must be reregistered to potentially cover the inserted record(s).

The only problem with the adjacency approach is that a B-tree page split will cause all triggers holding locks on the page to be awakened and all DEMAND command holding write locks to be reregistered.

The phantom problem and the logging problem appear easier to solve if an alternate strategy is employed. Consider storing the M and T locks in data and index records themselves. Systems which support variable length records can simply add as many T and M locks to each record as necessary. Such locks are automatically recoverable by current conventional techniques. The phantom problem requires the above adjacency algorithm; however, structure modifications (e.g., B-tree page splits) do not cause extra overhead. Moreover, since the extra locks are stored separately from R and W locks, extendable hashing is not a prerequisite for the lock table. Lastly, lock escalation can be handled by storing an extra record at the front of each page or relation indicating that all enclosed objects are locked. This record is

automatically recoverable, and the run-time system must simply guarantee that it looks for this special record before accessing any enclosed object.

The only drawback of this second alternative is that a second implementation of a lock manager must be coded for T and M locks.

6. Comparison With A View-Oriented Scheme

We illustrate the view based scheme suggested in [ULLM85, IOAN84] by performing the hair color example from Section 3 using views. Consider the following collection of view definitions:

```
range of E is EMP
define view EMP (E.all, hair = "green")
where E.name = "Bill"

range of E is EMP
range of E1 is EMP
define view EMP (E1.all, hair = E.hair)
where E.name = "Bill"
and E1.name = "Mike"
```

The interpretation is that EMP can be both a stored relation and a collection of view definitions. Moreover, the actual value of EMP is the union of the stored relation and the view definitions.

If the following query is specified

```
retrieve (EMP.hair) where EMP.name = "Mike"
```

it will be run on the stored relation to produce an empty answer. In addition, it will be run on both view definitions for EMP using the standard query modification procedure in [STON75]. This produces two queries:

```
retrieve (E.hair) where
E.name = "Mike"
and E.name = "Bill"

and

retrieve (E hair)
where E.name = "Bill"
and E1.name = "Mike"
```

A simple theorem prover can ascertain that the first query is false; alternately, the query can be run to produce an empty answer. The second query produces an empty answer when run on the stored

relation. However, when passed again through the view mechanism, it will be modified to produce the query which will yield the desired answer.

The problem with this approach is apparent. The view machinery contains no mechanism for indexing the collection of EMP views which are logically unioned. Previous work [ROUS82] has concentrated on indexing to speed materializing the tuples in a view and not on restricting the number of view definitions which must be evaluated. Therefore, the system must try all view definitions to find the subset which yield the answer. Hence, redundant queries will be executed leading to considerable inefficiency. Alternately, some sort of a theorem prover must be built into the view mechanism to limit the number of specifications which must be evaluated. This will be a sophisticated piece of software, and theorem provers are not noted for their execution efficiency.

In the case that there are a very small number of view definitions associated with a given relation, then a view-based scheme has obvious advantages. For example, if there is only one view definition which augments a stored relation, then at most two queries must be run to satisfy any command. Moreover, if views are cascaded, then the query modification algorithm [STON75] can be run iteratively producing at the end a small number of queries to optimize and execute. Plan optimization is performed only over these ultimate queries. On the other hand, the approach in Section 3 forces query planning to be done every time a new sub-query is constructed and query optimization will be performed over a larger collection of smaller queries. Optimization at the end is sure to result in a more efficient plan.

However, the technique in Section 3 is especially effective in discarding irrelevant rules in the case that there are a large number that might apply but only a few that actually do. A view oriented scheme performs this function much less efficiently. Hence, the composition of the application will determine which mechanism will work better.

6.1. Conclusions

This paper has suggested mechanisms to support rule processing using both forward and backward chaining in a relational database system. These require only minor extensions to the query language and locking system. Moreover, modest additional changes can support backtracking on failure in either environment. The proposed facilities are advantageous because they are easy to understand, easy to implement and can be efficiently supported. Moreover, no additional data structure is required to index either DEMAND or ALWAYS

commands. This can be contrasted with systems such as OPS5 [FORG81], which require such indexing.

Additionally, it appears possible to have simultaneous backward and forward chaining, if easily defined compatibility between T and M locks is established. One might suggest a solution such as the following:

replace (...) where ...
PARALLEL
retrieve (...) where ...

The replace command would start a collection of forward chaining triggers, and at the same time the retrieve would activate backward chaining. The forward chaining triggers could thereby provide data required by the backward chaining inference engine.

Our approach is not without drawbacks, however. For example, it is difficult to support a field which is physically stored for some records and computed by a lazy trigger for others. If the field has an index and a query uses this index as an access path, the locking system will fail to activate the appropriate DEMAND commands. Hence, lazy triggers are restricted to non-stored fields. Also, redundant clauses are often evaluated using the R-M conflict algorithm, leading to a loss of efficiency in processing DEMAND commands.

Acknowledgement

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 and the Naval Electronics Systems Command Contract N39-82-C-0235.

7. References

[ASTR76] [BC79] [BJ84] [CHEN84] [CM81] [CW84] [ESWA75] [FORG81] [GRAY78] [HEWI71] [IOAN84] [JCV84] [MW84] [ROUS82] [RTI84] [STON75] [STON76] [SW84] [ULLM85] [WARR81]