FUTURE TRENDS IN EXPERT DATA BASE SYSTEMS

Michael Stonebraker and Marti Hearst
EECS Dept.
University of California, Berkeley

Abstract

In this paper we discuss how we see the capabilities of DBMSs evolving over the next several years to meet the needs of expert data base applications. We also present some of the research thrusts which we see as important that appear to be receiving insufficient attention in the research community.

1. INTRODUCTION

We begin in Section 2 with an example application which we think characterizes the need for expert data base management systems. This application will set the context for the discussion in the rest of the paper.

This paper then turns in Section 3 to the architectural alternatives that can be adopted for the construction of expert data base systems such as the one described in Section 2. We begin with **loose coupling** between a rule manager and a data manager. The serious flaws in this architecture are then discussed followed by the possibility of **tight coupling** between rule and data management. We then point out that tight coupling will allow simple expert systems to be entirely implemented within the data manager while more complex ones can be written in a **cooperative** style whereby a data manager with an embedded inference engine and an expert system shell work cooperatively to implement an expert data base application.

Then we turn in Section 4 to the two thrusts which research on DBMSs with an embedded rule system have taken. A collection of researchers have concentrated on the efficient solution of rules systems that are recursive in nature. These naturally lead to the construction of query optimizers for a DBMS that can deal with recursive queries. On the other hand, other researchers have focused on more mundane rule systems where recursion is not present and have worked on efficiently discovering when one must "fire" rules from a large rule set. We conclude Section 4 by discussing the importance of these two thrusts and their likely commercial exploitation.

In Section 5 we discuss the possible technical approaches to deciding when to "fire" rules from a large rule set. We discuss approaches based on theorem provers, indexing techniques and physical markers in the data base. We conclude that none are satisfactory alternatives, and we are enthusiastic about research on other possible mechanisms.

Section 6 then concludes by indicating that the construction of real world expert systems entails much more than adding rule management to a DBMS. Specifically, expert data base applications seem to entail a collection of difficult object management problems. In Section 6 we discuss a simple expert system which we have been working on and indicate why a conventional relational system with an inference engine is not adequate for our application.

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

2. AN EXAMPLE EXPERT DATA BASE APPLICATION

Consider a computer program which is intended to give a person directions from any starting location in a particular geographic area to his destination. Such a program would be useful to car rental companies who must give directions from an airport location to the traveler's final destination. It would also be useful to route delivery vehicles in unfamiliar territory, etc. In fact, simple systems with the capabilities which we will describe are under development commerically at the present time.

The intended program will run off a data base which consists of a street map of the area in question. For each adjacent pair of nodes, the data base must store:

the travel time along the arc (perhaps by time of day)
the legal house numbers along the arc
whether it is a one-way street
the presence of stop signs
the road surface

For a geographic area such as the San Francisco Bay area, there are (say) 5 million people, two million legal addresses, and 200,000 arcs. This data base is clearly tens or hundreds of megabytes, and it will be difficult to compress it into a main memory structure. Moreover, the data base grows larger if the geograpic are is expanded or if an off-the road vehicle (such as a tank) is the target vehicle. In this latter case, we must store a topographic map and not a street map. As such, the application fundamentally has a data base problem.

In addition, one can solve this application by writing simple heuristic algorithms which might, for example, first try to find the best route to the nearest expressway interchange. Algorithms with this flavor are discussed in [PEAR86]. However, it seems more natural to approach the problem using a rule driven solution. For example, in the Bay Area, all the residents use the following rule periodically:

To get from anywhere in Berkeley to anywhere in San Francisco, get from
the current location to the Bay Bridge, then get from the Bay Bridge to
the destination.

This rule specifies that any good route from Berkeley to san Franscisco must include the Bay Bridge. Most of us navigate our local geographic areas by making use of hundreds or even thousands of such rules. Lets suppose that we wish a solution to our application that can make use of the (perhaps) thousands of rules that could control navigation in an area the size of the San Francisco Bay Area.

We will define an **expert data base application** as any application such as the one above that must interact with a sizeable data base and a sizeable rule base. We turn in the next section to the possible architectures by which such applications can be supported.

3. ARCHITECTURES FOR EXPERT DATA BASE APPLICATIONS

The first alternative that we could use for expert data base applications has been termed **loose coupling** and is illustrated in Figure 1. Here, one writes the application by making use of an expert system shell, such as Prolog, OPS5, KEE, ART, etc. Any application logic, presentation services, and the rules which control navigation are supported by this subsystem. For our purposes, the expert system shell manages the rule base as indicates in Figure 1. Since expert system shells are programming environments and not data base systems, they have no capabilities to store the map and legal addresses which our application requires. Hence, many shells have been extended to support calls on an external data manager. the KEE Connection [ABAR86] is an example of a product that supports such external DBMS access. Hence, the

Loose Coupling Architecture
Figure 1

data base portion of the application is managed by a second software system. The two subsystems are loosely coupled in that the shell makes calls on data base services in the same way as any other DBMS user.

There are several severe disadvantages to this architecture. First, the the rule base is main memory resident. Hence, if the address space in which the shell resides goes away, then the rule base also disappears. If the application makes any changes to the rules, these would not be preserved unless manual procedures to save them were utilized. Although this characterisitic is not an issue in our example application, there are many situations where the rules must be changed dynamically. Moreover, the rule base is not shared. If one user changes a rule, other concurrent users are not informed of the change.

Another severe disadvantage concerns **dynamic** data. Suppose the shell extracts a fact from the data base and then subsequently the value of the fact changes. This might happen if the shell performed a complex inference that took considerable time. It might also occur if the shell **cached** the fact in main memory for the duration of an inference session to obtain faster performance. In either case, the shell is maintaining a cache of data base objects, and the consistency of the cache must be ensured. The only possible techniques are to run all data base extractions inside a transaction and then to end the transaction only when the desired inferences have been completed. This will ensure that the extracted data base objects are locked for the duration of the inference. This approach will disallow updates to extracted objects until the inference engine has finished, and may seriously impact performance. On the other hand, the shell can **poll** the data manager periodically to obtain the new contents of extracted values. This approach will require the shell to needlessly run DBMS queries even when the data items in question have not changed. it also may severely impact performance. Hence, an application where the shell must deal with dynamically changing facts is sure to be a problem for a loose coupliing architecture.

The second problem area is so-called **non-partitionable** applications. Suppose the shell must run a query to fetch the entire fact base before the user's desired inferences can be completed. In this case, the size of the shell's cache may be gigantic and the resulting performance of the application may be poor because of the sheer size of address space required to hold the cache.

Because of these disadvantages, many DBMS researchers have been investigating ways to integrate a rule system into a data manager. We will term this approach **tight coupling.** Using this architecture, the DBMS manages both the data base and the rules base. In this case, the rule base is automatically shared and persistent. Moreover, the DBMS can deal with dynamic data easily by waking up rules only when the data they require actually changes. Lastly, because a DBMS is tuned to manage very large amounts of data, there should be no particular problem with non-partitionable data. Hence, in theory, the disadvantages of loose coupling can be corrected by moving the rule system inside the DBMS. The next section discusses the two approaches that have been taken in this regard.

However, it is likely that an embedded rules system will lack certain functions routinely present in expert system shells. For example some rules system, such as emycin [xxx], allow a user to attach a certainty factor to each rule. The inference engine is then coded to combine certainty factors together to give an indication of the strength of the composite inference. Moreover, most expert system shells are capable of giving explanations for any chain of inferences accomplished. As will be seen in the next section, few DBMS researchers are contemplating the inclusion of uncertainty or explanations. A user who requires such capabilities must code them in an expert system shell. In this case we have an example of a **co-operative** architecture in which two subsystems, the DBMS and the shell, each have an inference engine and co-operate to solve the ultimate application. We expect such co-operation to become more prevalent in the future.

4. TIGHT COUPLING ALTERNATIVES

There are two main thrusts to a DBMS rule system. We illustrate the first approach by an example. Consider the standard PROLOG program concerning grandparents, e.g:

grandparent (X,Y) := parent (X,Z), parent (Z,Y)
parent (Joe, Sue) :=
parent (Sam, Bill) :=

parent (Sam, Joe) :=

Clearly, the parent facts can be placed into a PARENT relation

PARENT (older, younger)

in a conventional relational DBMS. The rule defining grandparents can be stated as a relational view, i.e:

```
range of P, P1 is PARENT
define view grandparent (P.older, P1.younger)
where P.younger = P1.older
```

Then a query about grandparents, i.e:

```
retrieve (grandparent.older)
where grandparent.younger = "Sue"
```

can be processed by the view algorithms in a conventional system to:

```
range of P, P1 is PARENT
retrieve (P.older)
where P.younger = P1.older and P1.younger = "Sue"
```

The resulting query can be optimized in the conventional way. Hence, relational views already provide support for some kinds of rules. However, the first thrust is to generalize the support provided by a DBMS so more complex rules can be expressed. Consider for example the rule:

ancestor (X,Y) := ancestor (X,Z), parent (Z,Y)

This rule is **linearly recursive** because the same clause appears on both sides of the rule. In this case, the view definition which corresponds to the rule is:

```
define view ancestor (ancestor.older, parent.younger)
where ancestor.younger = parent.older
```

Hence, the view definition contains a reference to itself and is recursive. A query to this view, i.e

retrieve (ancestor.older) where ancestor.younger = "Sue"

must be translated into the following statements:

```
retrieve into temp (parent.older) where parent.younger = "Sue"

append * to temp (parent.older) where parent.younger = temp.older
```

The second statement is an example of a **transitive closure** query, i.e. one that can be expressed by a single query language command with a "*" added to denote that logically the command should be executed over and over until it ceases to have any effect. There has been a significant amount of work on optimizing transitive closure queries, i.e. [ROSE86, IOAN87].

More generally, one can express rules that involve **general recursion.** For example the following rule defines common ancestors:

common-ancestor (X,Y) := ancestor (Z,X), ancestor (Z,Y)

Here, there is more than one recursive clause on the right side of the definition, and queries about common ancestors are clearly much more difficult to solve than queries involving tranitive closure. Research on efficiently solving queries involving general recursion has been presented in [BANC86, 5 more].

The first thrust to rule management within a DBMS is therefore a substantial research effort on linear and general recursion which is oriented to efficient solution of the above kinds of rules. The second thrust is oriented to non-recursive rules and is illustrated by the standard EMP relation:

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

The first four fields are conventional data, and we turn to focus on the last field, desk. Most companies have elaborate rules on who is allowed to have what kinds of desks. Hence, this field is largely determined by a collection of rules, typically written down in a company personnel manual or even stored in a key

employee's head. Consequently, if we simply store the desk column as ordinary data, we will not capture the rules that restrict admissible values. Rather, we would like to store a collection of rules that determine desk values.

We now discuss the POSTGRES rule system which allows such rules in a natural way. POSTGRES supports a query language, POSTQUEL, which borrows heavily from its predecessor, QUEL. Its main extensions are in the areas of user defined operators and functions, and facilities to deal with time and inheritance. The rule system uses this query language in an integral way. Specifically, any POSTQUEL command can be turned into a **rule** by prepending it with a keyword. Currently, we support the keywords, **always, never,** and **onetime.** For example, the following command sets the desk of Mike to that of Joe:

```
replace EMP (desk = E.desk)
using E in EMP
where EMP.name = "Mike" and E.name = "Joe"
```

At the time the command is run, this will make the appropriate update to Mike's desk. To turn this command into a rule, one can prepend this command with the keyword always, i.e.:

```
always replace EMP (desk = E.desk)
using E in EMP
where EMP.name = "Mike" and E.name = "Joe"
```

The semantics of this revisied command is that it should appear to be continuously running. In fact, the implementation of this command follows one of two plans.

First, at the time that Joe receives a new desk, POSTGRES will awaken this command and propagate the change to Mike. We call this **early evaluation.** Hence, the command is awakened whenever any data item which it reads is modified. On the other hand, the second strategy for rule evaluation is to do nothing at the time that Joe receives a new desk. Rather, one waits until somebody requests the desk of Mike. At this time, rather than giving the person a stored value, POSTGRES runs a modified version of rule to fetch the actual data item from Joe's record. We term this strategy **late evaluation.** The intent in POSTGRES is to automatically choose between early and late evaluation on a rule by rule basis.

Of course collections of rules can interact. For example, suppose there is a second rule which sets the desk of Joe to be that of Sam:

```
always replace EMP (desk = E.desk)
using E in EMP
where EMP.name = "Joe" and E.name = "Sam"
```

If both rules are evaluated early, then a **forward chaining** control flow will result because when Sam gets a new desk, the second rule will fire which will in turn fire the first rule. On the other hand, late evaluation corresponds to **backward chaining** because a request for Mike's desk will cause a request for Joe's desk which in turn will cause a request for Sam's desk. As a result, POSTGRES is automatically choosing between backward and forward chaining control flows.

Two other points about the rule system are noteworthy. First, it is permissable to have conflicting rules. For example, one might have a rule that all employees over 35 get a wood desk, i.e:

```
always replace EMP (desk = "wood")
where EMP.age >= 35
```

In the case that Mike or Joe is over 35 but Sam is not, then this rule will give both employees a different desk than the previous rules. POSTGRES deals with this situation by allowing a user to specify a **priority** for each rule and then enforces the one with higher priority if a conflict occurs.

Lastly, if a user specifies rules that would cause the rule system to loop, e.g:

```
replace EMP (salary = 1.1 * E.salary)
using E in EMP
where EMP.name = "Mike" and E.name = "Joe"
```

```
replace EMP (salary = 1.1 * E.salary)
using E in EMP
where EMP.name = "Joe" and E.name = "Mike"
```

Here Joe's salary is 10 percent more than Mike's which in turn is 10 percent more than Joe's. POSTGRES would ordinarily go into an infinite loop if either early or late evaluation was selected. However, POSTGRES simply remembers a stack of the previous rule activations. It examines the stack on rule activation and notices if a previously invoked rule is being awakened in the same state. If so, it aborts the current transaction to break the infinite loop.

This strategy breaks all infinite loops as well as some additional non-looping situations. For example, if one adds an extra clause to each of the above rules of the form:

```
and EMP.salary < 5000
```

then the rules will terminate after a finite number of iterations. However, POSTGRES will fail to recognize this fact and abort the requester of a salary (late evaluation) or the updater of a salary (early evaluation).

A complete set of rules for desk allocation culd be written in POSTGRES and might be:

```
always replace EMP (desk = "steel")
where EMP.age >= 35

always replace EMP (desk = "wood")
where EMP.age < 35
```

These are examples of the **general rules** which indicate that employees under 35 get a wood desk while those over 35 receive a steel desk. However, there are invariably **exceptions** to the general rules. Suppose that Bill is a younger employee who should receive a wood desk while Sam is an older employee who refuses to sit at a wood desk. Lastly, suppose that Joe has been allowed to have the same kind of desk as Sam and Mike the same desk as Joe. These exceptions would be expressed as:

```
always replace EMP (desk = "wood")
where EMP.name = "Bill"

always replace EMP (desk = "steel")
where EMP.name = "Sam"

always replace EMP (desk = E.desk)
using E in EMP
where EMP.name = "Joe"
and E.name = "Sam"

always replace EMP (desk = E.desk)
using E in EMP
where EMP.name = "Mike"
and E.name = "Joe"
```

Consider a query to the EMP relation, i.e:

```
retrieve (EMP.desk) where EMP.name = "Joe"
```

Although 6 rules can conceivably yield the answer to this query, we require the fifth rule to be applied. This will yield a new query:

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

Again 6 rules might apply but we require the fourth rule which will yield "steel" as the desired answer to the query.

This rule set is characterized by three properties:

1) Because there are exceptions, the rule set is fundamentally inconsistent. Hence, a mechanism must be found to deal with this situation.

2) There are a substantial number of rules, and none of them are recursive.

3) The key performance measure of queries which involve employee desks will be to efficiently decide which rule (or rules) must be "fired". Although many rules **might** apply, few **actually** apply.

We will term rule systems with these three properties **mundane** rules.

Having talked to a considerable number of DBMS users, we have observed the following characteristics. Nearly everybody has mundane rules (such as the definition of desks). In addition some people have linearly recursive rules (such as the definition of ancestors). The most common example is parts explosion queries which occur frequently. There are probably a few people who have more complex general recursive queries. However, we have yet to meet such a person. Consequently, our feeling is that research effort should be proportional to the needs of real users. Hence, we would like to see more effort on mundane rules sets and less effort on general recursion.

Moreover, the vendors of commercial systems will clearly be oriented toward mundane rules because that will result in the broadest applicability of system constructs.

5. TECHNIQUES FOR RULE FIRING

We believe that the key to efficiency in DBMS supported mundane rule systems is firing only those rules which actually apply. There seem three approaches which can be exploited:

theorem proving
indexing
flags

We discuss each in turn.

When a query is entered, e.g:

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

the DBMS can provide a **theorem prover** which will provide the following service. For each rule, R, with qualification Q(R), the theorem prover will ascertain if:

(EMP.name = "Joe") intersect Q(R) is empty

Any such rule can be immediately discarded. The rules which cannot be discarded are fired. Hence, the focus could be on efficently building such a theorem prover, and research along these lines is discussed in [SCHM86]. We are not optimistic that this approach will work well. First, a theorem prover will sequence over all rules involving desks to deal with the above query. If there are 1000 such rules, then a sequential evaluation of them will be a major bottleneck. Second, it may well be just as efficient to "fire" the rule to see if it applies rather than trying to prove that it doesn't. Moreover, in the case of the desk rules, it will be necessary to retrieve the age of Joe in order to ascertain which general rule applies. In this case, the theorem prover will require extra information from the data base to do its job, thereby degrading performance.

The second approach is to index the predicates in the rules. The general idea is to build a data structure which will store for each rule, R, the qualification Q(R). Then, when a user query is given to the system, the qualification is entered into the index and in the process the rules whose qualification overlaps the one from the user will be found. Such an index is inherently multidimensional and might be a generalization of R-trees [GUTM84]. In addition RETE networks [FORG83] are a kind of indexing system long used in forward chaining expert system shells. Perhaps a variant on this approach could be adapted for data base use.

The final approach is to use a approach based on flags. The basic idea is to actually run the qualification from the rule through the execution system of the DBMS, and have it **mark** all the data items which the rule accesses or intends to update. Such **flags** can be subsequnetly used to assist in determining which

rules to fire. In the case of the desk example, the various rules would set flags on the desk data item for employees that they intend to specify a value for and also on the names or ages of employees which are the data items they read.

Subsequently, if a user wishes to read the desk data item of any particular employee, the run time system can notice the flags that are set on that data item, and wake up only those rules. This will provide very fine granularity discrimination of rule activation. A proposal along these lines is presented in [STON87], and an analytic model comparing the performance of a flag system to an indexing system is contained in [STON85].

Our experience with a flag implementation is that it is **very** complex. Perhaps it is overly complex. Hence, the complexity taxes the designers and implementors, and this makes us uneasy about the ultimate viability of this approach. On the other hand, we have not seen a predicate indexing scheme that has sufficient generality to allow all the predicates that a rule system requires. For example, it is not, in our opinion, reasonable to restrict predicates in rules to just those which involve a single relation. Lastly, theorem proving techniques do not appear very promising either. Hence, we feel that a major contribution can be made by a new idea, and we would encourage researchers to look in this area.

6. OBJECT MANGEMENT IS ESSENTIAL

We have been working with the State of California Department of Water Resources (DWR). This agency is repsonsible for much of the water delivery and husbandry in the state. They maintain a large collection of data sets on water related matters. Specifically, for a collection of wells in the state, they monitor the depth to groundwater on a periodic basis. This information is useful for tracking the current amount of underground storage and rate of undergroud pumping. The U.S. Department of Reclaimation maintains the same data for a different collection of wells. We have obtained both data sets. ,pp DWR also does "fly overs" on a periodic basis taking image data. From this raw data they try to infer crop patterns and amount of land in cultivation.

They also monitor many of the waterways in the state for content of various impurities. A little known fact about irrigated land (which essentially all of California farmland is) is that salt and other impurities are natually present in stream water. When such water is poured onto fields the salt content gradually builds until the salinity of the soil renders the land unusable. This problem is addressed by "flooding" the field periodically, thereby dissolving the salt, and then "draining" the field (typically through underground drains) into an exit waterway. This drain water is high in salt content and must be disposed of. The problem with agricultural waste water has attracted substantial attention in the news media, and there are many stories on the subject in major California newspapers.

Our objective was quite modest. We wished to build a data base containing:

Bureau of Reclaimation data on depth to groundwater
DWR data on depth to groundwater
DWR data on impurities
DWR image data
a U.S.G.S. topographic map
electricity usage
newspaper clippings on agricultural water use

Then we wished to achieve two objectives. The first was to present to an end user a "seamless" view of this data. This required us to contruct a user view of the above data which for any possible plot of land (identified by an <X,Y> coordinate position) in our study area would have the following information:

depth to groundwater
land use (i.e. crop grown)
salt concentration
selenium concentration
amount of underground pumpage

For example, to infer the depth to groundwater at a particular location, one can use:

the depth to groundwater at neighboring wells
the difference in elevation between the various locations
the confidence in the source of the well measurement
the time of year

The second objective was to build a system that would be able to analyze the newspaper clippings and rate them on a collection of metrics. For example, one solution to drain water is to purify it and then put it into the normal system of waterways. We wished to be able to scale each news article on whether it was in favor of this solution or opposed, and we wished to use a 0-10 scale. Consequently, we wished to present to the end user a relation with a row for each news article with data items:

date
newspaper
author
score on metric 1
.
.
.
score on metric n

There are several problems which we ran into in trying to implement this application. First, it is necessary to store a topographic map of a portion of the State of California. This requires a system that can conveniently store polygons, line groups, points, etc. Moreover, many of the polygons have a large number of sides. Managing such objects tends to be very hard in current relational DBMSs. A system with more sophisticated object management capabilities would be a definite plus. Second, it is necessary to do queries of the form:

find me the N closest wells to the point (x,y)

This requires a two dimensional spatial index such as a K-D-B tree [ROBI80], R-tree [GUTM84] or quad-tree [SAME84]. No current general puirpose relational systems have such access methods, and it is necessary to use a spcial purpose system with geographic search capabilities or a system that can be extended with access methods written by a user, e.g. [STON86, LIND87, DEWI87].

Lastly, we tried to use the rules system to fill in values for the missing data in the two relations from the previous section. However, we encounered difficulties as illustrated by the depth-to-groundwater discussion which follows. Obviously, we don't want to have a tuple in the first relation for every conceivable <x,y> pair in California. Hence, there will be no tuple in the given relation for the point for which the user is interested in the depth to groundwater. Consequently, there is no data item for the rule system to fill in. As such, the obvious mechanism is for the user to write a function:

depth-groundwater (x,y)

that takes two arguments. This functions can be defined to POSTGRES and then used in subsequent queries. The natural query is:

retrieve (result = depth-groundwater (my-x, my-y))

Hence, user defined functions are a very valuable extension to a data manager.

In summary, object management capabilities such as extendible types, user defined access methods and user defined functions appear to be important in making expert data base applications successful.

7. REFERENCES

[ESWA75] Eswaran, K., *"A General Purpose Trigger Subsystem and Its Inclusion in a Relational Data Base System"*, IBM Research, Report No RJ 1833, San Jose, CA, July 1976.

[GUTT84] Guttman, A., *"R-Trees: A Dynamic Index Structure for Spatial Searching"*, Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data, Boston, MA, June 1984.

[ROBI81] Robinson, J. T., *"The K-D-B tree: A Search Structure for Large Multidimensional Dynamic Indexes"*, Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data, April 1981.