APPLICATION OF ARTIFICIAL INTELLIGENCE TECHNIQUES TO DATABASE SYSTEMS

By

Michael Stonebraker

Memorandum No. UCB/ERL M82/31

6 May 1982

This research was supported by the National Science Foundation through grant number 8007683, the Navy Electronics Systems Command through grant number N00039-81-C-0569, and the Air Force Office of Scientific Research through grant number 78-3596.

APPLICATION OF ARTIFICIAL INTELLIGENCE TECHNIQUES

TO DATABASE SYSTEMS

by

Michael Stonebraker
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

This paper suggests two mechanisms for adding semantic knowledge to a data manager, namely inclusion of an AI oriented rules system and a particular use of abstract data types.

I INTRODUCTION

Research effort on expert systems by the Artificial Intelligence (AI) community has largely focused on selecting a domain of discourse and then attempting to make a computer program competitive with a human expert in that domain. Systems such as MYCIN [SHOR76] in the domain of medical diagnosis and PROSPECTOR [DUDA78] in the area of geological exploration are examples of this approach. As a subsequent step, one might attempt to broaden the scope of an expert system to other fields. E-MYCIN is an example of a project with this focus.

On the other hand, database systems strive to be general purpose and provide limited intelligence in a broad arena. Much of the effort on DBMS semantics attempts to make a general purpose data manager "smarter". This paper discusses two mechanisms along these lines. First, we will discuss the inclusion of an AI oriented rules system in a data manager. Then, we will discuss the use of abstract data types (ADTs).

II RULES SYSTEMS

2.1 Introduction

Many services of a database management system (DBMS) can be interpreted as rules systems. For example, integrity constraints [STON75, HAMM76] specify conditions which must be guaranteed by a data manager. One such constraint for the relation

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

is that employee salaries be greater than 1000. It can be expressed in the current INGRES DBMS [STON76, STON80] as:

```
range of E is EMP
define integrity E.salary > 1000
```

This condition is automatically enforced by modifying each incoming salary update to one which is guaranteed not to violate the constraint. For example, the command

```
range of E is EMP
replace E(salary = .8 * E.salary) where E.name = "Smith"
```

is changed to

```
range of E is EMP
replace E(salary = .8 * E.salary) where
```

E.name = "Smith" and .8 * E.salary > 1000.

The last clause ensures that Smith's updated salary cannot violate the constraint.

This modification procedure is triggered by an incoming command and performs a collection of actions which alter the command. Hence, it is of the form

On condition
Then action

As such, it is a special purpose rules system. In addition, alerters [BUNN79], triggers [ESWA76], protection services [GRIF76, STON74], and support systems for views [CHAM75, STON75] follow the same paradigm. Consequently, they are also rules systems.

Many DBMS implement such database services individually. For example, INGRES implements integrity control, protection and views with three independent modules; each of which is a special purpose rules system. The purpose of this section is to propose a single rules system which can provide all such database services. In this way only one mechanism need be implemented, and an economy of database code may result. Moreover, many rules not possible with existing DBMS services can also be formulated.

The next subsection indicates our thoughts in this area.

2.2 RAISIN

The language by which a database administrator or user specifies rules is called RAISIN (Rules from AI Specified for INgres). Its basic structure is a sequence of ON-THEN clauses. That is,

ON (condition) THEN (action)

ON (condition) THEN (action)

:

:

For each ON-THEN clause, the condition will specify constraints to be met by an incoming data manipulation command before the action can be applied. Moreover, the condition can depend on data in the database system. The action will be a set of operations to be performed on the command as well as other possibly new operations on the database.

In this section we specify the allowable conditions and actions in RAISIN. The general form of a condition is the following:

```
ON          command(s)
TO          relations(s)
AFFECTING   field(s)
QUALIFYING  field(s)
BY          user-name(s)
DURING      time-range
FOR         day-range
WHERE       qualification
```

Hence a condition is a collection of terms, each of which is a keyword followed by a parameter. We give a few examples of conditions then explain the general syntax.

```
ON          replace
TO          EMP
```

```
ON          replace
TO          EMP
AFFECTING   salary
WHERE       EMP.name = "Smith"
```

```
ON          append, replace
TO          *
BY          Jones
DURING      8:00-17:00
```

FOR mon-fri

The first condition applies to all replace operations to the EMP relation while the second applies to a salary update for an employee named Smith. Lastly, the third condition applies to all database modifications made by Jones during normal working hours.

It should be noted that all terms in a condition except the first are optional and the wild card "*" is a valid parameter standing for "always". The TO clause specifies a list of relations in the current database to which this rule applies while the AFFECTING term indicates what fields must be updated for the condition to apply. Moreover, the QUALIFYING clause indicates what fields must be present in the qualification of a user command for the condition to apply. For example, the command which gave a 20 percent salary decrease to Smith uses name in the qualification. A rule which included the term

QUALIFYING name

would apply to this update.

The day-range, time-range and user-list constructs are self-explanatory. Lastly, the WHERE clause qualifies data to which the rule applies. Hence, it should be a valid qualification in a data manipulation language. In this exposition, we assume that qualification is a QUEL WHERE clause modified in one important way. In QUEL, all field names must have an attached range variable. Hence, E is declared to range over EMP in the above QUEL example and fields are designated by E.name and E.salary. In RAISIN qualifications we assume that a relation name is prepended to a field name instead of a range variable. Hence,

EMP.name and EMP.salary would be valid field names.

For any incoming data manipulation command, the first condition of any rule is either true or false. If false, the rule does not apply. However, if true the action part of the rule is executed and the remainder of the ON-THEN statements (if any) are checked for applicability. We now turn to the legal actions which can appear in a RAISIN statement

The action portion of an ON-THEN statement is an ordered collection of commands from the following list.

1) EXECUTE

The user command is performed automatically as the last action of a rule. If a user wants the command done earlier, he must use an EXECUTE statement. Two EXECUTE statements in a row would cause the user command to be run twice.

2) CANCEL

This action cancels the execution of the user's command.

3) UNDO

This action undoes all changes to the database since the beginning of the rule. With the inclusion of this action there is the implicit assumption that transactions are supported.

4) CHANGE relation-1 TO relation-2,..,relation-N

This action will change the scope of the user command from relation-1 to

relation-2,.., relation-N. More precisely, whenever one has

```
range of var-1 is relation-1
```

this is changed to

```
range of var-2 is relation-2
  .
  .
  .
range of var-N is relation-N
```

Var-2, .., var-N are internally assigned by a RAISIN implementation. Moreover, for any given field name, F, in relation-1, it is assumed that only one relation, say relation-j, has a field of the same name. Hence,

```
var-1.F
```

is changed to

```
var-j.F
```

For example, one can deflect all operations on the EMP relation to the NEW-EMP relation by the following rule.

```
ON      *
TO      EMP
THEN
CHANGE  EMP to NEW-EMP
```

5) RENAME field-1 TO field-2

This action causes all references to field-1 to be changed to field-2. If, for example, NEW-EMP has a salary field named dollars, the action statements of the above rule should be extended to the following:

```
RENAME salary   TO dollars
CHANGE EMP TO NEW-EMP
```

6) MESSAGE {TO user-name} "message text"

A message is returned to the person who issued the command that activated the rule. If the optional clause TO user-name is included, the message is directed to another user. The MESSAGE action is useful when a command must be aborted and an error message returned.

7) ILLEGAL "message text"

This action inspects the current command to see if it is syntactically valid. If not, it will perform a CANCEL and generate a message. Consequently, it has the following effect:

```
ON       syntax error
THEN
CANCEL
MESSAGE "message text"
```

8) QUEL command

Any QUEL command is a legal action. For example, suppose RULES is a relation with two fields, a rule number and a count field indicating how many times any given rule has been executed. The action statement needed to correctly update this relation for rule number 16 follows.

```
range of R is RULES
replace R (count = R.count + 1) where R.number = 16
```

Unfortunately, this action statement must be repeated for each rule currently being enforced.

One extension is needed to QUEL commands in a RAISIN context. Portions of the user command which activated the rule can be substituted into a QUEL statement which is applied as an action. The following key-

words indicate the needed portions.

qualification	-	a keyword for the qualification in the users command
command	-	a keyword for the whole user command
new.field-name	-	a keyword for the value being assigned to field-name by the user command.

These can appear where they are semantically valid in a QUEL command. For example, in the command which gave a 20 percent pay decrease to Smith, qualification has the value

```
E.name = "Smith"
```

while new.salary has the value

```
.8 * E.salary
```

9) ADDQUAL qualification

This action will perform query modification [STON75] on the current command. Specifically it will add the indicated qualification to the one specified by the user. This extra qualification follows the syntax of QUEL WHERE clauses except each field name has a relation name prepended instead of a range variable. Since the user's command will have a range variable in front of each field name, the qualification must be preprocessed to find each field name, remove the prepended relation name and substitute the user's range variable.

For example, we can restrict Jones to the subset of employees under 30 by the following rule:

```
ON      *
TO      EMP
BY      Jones
```

```
THEN
ADDQUAL  EMP.age < 30
```

If Jones issues a query such as

```
range of E is EMP
retrieve (E.salary) where E.name = "Smith"
```

then it will be modified to

```
retrieve (E.salary) where E.name = "Smith"
                              and
                          E.age < 30
```

Notice that EMP.age is preprocessed to E.age before being added to the command. One other processing step must take place. The keywords noted in command (8) are also valid here, and the appropriate substitutions must take place.

2.3 Examples

We indicate the use of RAISIN to accomplish integrity constraints and protection statements. Additional examples are presented in [STON82]. If employees must make more than 1000, then the following integrity constraint in INGRES expresses this desire.

```
range of E is EMP
define integrity E.salary > 1000
```

In RAISIN this rule can be expressed as:

```
ON       replace, append
TO       EMP
THEN
ADDQUAL new.salary > 1000
```

Note that new.salary refers to the value assigned to salary by the user command.

Suppose Jones is only allowed to update salaries of employees for whom he is the manager between 8 A.M. and 5 P.M. This can be expressed in INGRES as:

```
range of E is EMP
define permit replace of E(salary) to Jones
FROM 800 to 1700  WHERE E.manager = "Jones"
```

This can also be specified in RAISIN as:

```
ON          replace
TO          EMP
AFFECTING   salary
BY          Jones
DURING      8:00-17:00
THEN
ADDQUAL     EMP.manager = "Jones"
```

2.4 Conclusions

The above section has specified a rules system which can be used to obtain all popular database services. It is anticipated that this system can be made as efficient as providing the same database services with special purpose code. Moreover, a single rules system is easier to implement than the same collection of services individually.

Three questions remain unresolved. First, storage of rules in a relational DBMS is not appealing. A storage structure for RAISIN rules is suggested in [STON82]; however, it is neither particularly efficient nor easy to understand. It is an open question how to extend a relational DBMS to be a more attractive storage system for rules. The same sort of deficiencies arise in attempting to use a relational DBMS to store the parsed representation of a program in a general purpose programming language [POWE82].

The second question concerns the specification of rules. In general, integrity constraints, protection statements and view specifications are easier to understand than the comparable RAISIN statements. Of course, it is a simple matter to internally map these these classes of statements to RAISIN rules. However, the design of a more appealing syntax for triggers and alerters should be studied.

Moreover, in a RAISIN context it is possible to design applications with a large collection of triggers. In fact, some applications can be primarily specified by triggers. An application specified in this manner will not be particularly easy for a human to understand, debug or maintain. Office automation languages containing messages, e.g. [ROWE82] have the same readability problem. It is an open question how to design a specification system for triggers which is easy to understand.

The third issue concerns commands containing a join of a relation to itself. For example, one can give a 10 percent pay cut to all employees who earn more than their managers as follows:

```
range of E is EMP
range of M is EMP
replace E (salary = 0.9 * E.salary) WHERE
       E.salary > M.salary AND
       E.manager = M.name
```

The RENAME action can be applied to this command; however it will change the command so that no employee can possibly qualify. Moreover, the statement

```
ADDQUAL EMP.salary > 1000
```

is ambiguous. One is uncertain whether this qualification applies to

employee salaries or manager salaries. Consequently, application of rules to reflexive joins is an open issue.

III THE USE OF ABSTRACT DATA TYPES

3.1 Introduction

Abstract data types (ADTs) [LISK74, GUTT77] have been extensively investigated in a programming language context. Basically, an ADT is an encapsulation of a data structure so that its implementation details are not visible to an outside client procedure along with a collection of related operations on this encapsulated structure. The canonical example of an ADT is a stack with related operations: new, push, pop and empty.

It has been pointed out that ADTs can be applied in a relational database context [ROWE79, SCHM78]. Briefly, a relation would be an abstract data type whose implementation details would be hidden from application level software. Then, allowable operations would be defined by procedures written in a programming language that supported both database access and ADTs. For example, one use of this kind of abstract data type is suggested in [ROWE79] and involves an EMPLOYEE abstract data type with related operations hire_employee, fire_employee and change_salary. This use of ADTs can serve to limit access to a relation in prespecified ways, thereby guaranteeing a higher level of data security and data integrity. Also, a view can be defined as an ADT. Consequently, the algorithm that transforms updates on views into updates on base relations can be encapsulated in the ADT and a high degree of data independence provided in this fashion.

This section presents a different use of ADTs. We will explore using ADTs for individual columns of a relation. The thrust will be to extend the semantic power of a data manager by defining new data types and related new operators on these data types using user written procedures obeying a specialized protocol. These ADTs are a generalization of database experts [STON80a].

We begin with a motivational example of the need for this kind of ADTs in Section 3.2. Then in Section 3.3 we define our use of abstract data types. Lastly, Sections 3.4 and 3.5 will close with an implementation proposal and some possible extensions of our definition of ADTs.

3.2 Time as an ADT

One would like to be able to define a column of a relation to have the data type "time". For example, one might create an event relation as follows:

```
create event (ename = c20,
              p-cancel = f4,
              type = c6
              date = time)
```

Here, an event relation is desired with four fields, the name of the event as a character string, its probability of cancelation as a float, its event type as a character string and the date of the event as a time field.

One would like to add events to this relation, e.g.

```
append to event (ename = "lunch",
                 p-cancel = 0,
                 type = "food",
                 date = "12:00 - 1:00")
```

Clearly, all fields can be correctly converted to an internal

representation and stored in a database system with the exception of the string "12:00 - 1:00". In order to be interpreted as a time, special recognition code will be required.

Moreover, one would like to use standard DBMS operators on the time domain, e.g.

```
range of e is event
replace e ( date = e.date + "1 hour")
        where e.ename = "lunch"
```

Here, one wishes to move lunch forward one hour. Somehow, a data manager must be instructed concerning the interpretation of addition between two times.

In addition, one would like to define new functions on the time column. Numerical columns have sin, cos, log, etc. defined as built-in functions. Each of these accepts an integer or float as input and returns a float. Similarly, one might want to define the length of a time interval and use it in data manipulation commands, e.g.

```
retrieve (e.ename)
where length(e.date) < "1 hour"
```

The problems here are twofold. First, one must actually define to a database manager the function "length" which accepts a time as input and returns a time. In addition, one must inform the database system of the meaning of "<" for the data type "time". Without this added semantic knowledge, a DBMS cannot know that "59 minutes" is less than "1 hour".

Moreover, one might like to define new comparison operations. For example, in the time domain, the semantic concept of "contained in" makes sense, and one might want an operator "¦=" defined for this pur-

pose. Then, one could ask if there was any event contained within the lunch period as follows:

```
range of e is event
range of f is event
retrieve (f.ename) where f.date ¦= e.date
                  and e.ename = "lunch"
```

Lastly, one would like to be able to define aggregate functions for the time column. For example, one would like to be able to find the first future event as follows:

```
retrieve (e.ename) where
e.date = min (e.date where e.date > "today")
```

Again, extra semantic knowledge is required to to define the "min" function for this new data type.

Consequently, we require a mechanism that allows a new data type to be defined with its own particular internal representation. In addition, normal comparison, arithmetic and aggregate operators may be desired for this new type. Lastly, new operators that obey the syntactic conventions of built-in functions, comparison operators, arithmetic operators and aggregate functions are desired. The next section proposes a simple mechanism to support all of the above capabilities.

3.3 ADTs

ADTs are a mechanism for adding procedural knowledge associated with a column to a data manager. Basically, an ADT consists of a registration process and a collection of functions. These are discussed in turn.

An ADT is registered with a data manager as follows:

```
define ADT name
(length = value{, optional-field = value})
```

Here, an ADT is registered by giving the length of its internal representation along with a collection of optional fields. For example,

```
define ADT time (length = 32)
```

will register a time ADT with a 32 byte internal representation.

One implementation [OVER81] used the first 16 bytes as the lower bound of a time range and the latter 16 bytes as the upper bound. The coding was:

```
year:        4 bytes
month:       2 bytes
day:         2 bytes
day-of-week: 1 byte
am/pm:       1 byte
hour:        2 bytes
minute:      2 bytes
second:      2 bytes
```

The optional fields are useful in query processing heuristics. For example, suppose one wants to find the names of events that happen at 9:30, i.e.:

```
retrieve (e.ename) where e.date = "9:30"
```

Once the string "9:30" is converted to a 32 byte internal representation, a data manager such as INGRES can find qualifying tuples. Suppose there is a secondary index on the date field. In this case INGRES can quickly find the tuples which have a date field which matches the 32 byte representation for "9:30". However, it must know that no other values for the date filed can possibly match "9:30". Without this

knowledge, a complete sequential scan of the events relation is required. Hence, the first optional field, opt1, is set to true if normal equality can be used by INGRES to limit the search space for qualifying tuples.

The second optional field, opt2, is set to true if the normal meaning of "<" and ">" can be use to limit the search for qualifying tuples. If additional query processing information is useful, more optional fields may be required.

These hints allow normal index processing to be performed for secondary indices which contain new data types. Hence, an ADT does not need to become involved in query processing heuristics.

Once an ADT has been registered, one can define various functions for it using one of two formats:

```
define procedure-type operator-name (adt-name)
                      as procedure-name
                      returning adt-name
```

or

```
define procedure-type adt-name operator adt-name
                      as procedure-name
                      returning adt-name
```

We will now sequence through the various types of procedures:

a) Conversion Routines

In order to process time columns, routines must be provided to convert between character string representation and 32 byte internal representation. For example:

```
define conversion input (character)
                  as my-proc-1
                  returning time
```

```
define conversion output (time)
                as my-proc-2
                returning character
```

These routines would be called as appropriate to convert lunch between its 32 byte internal format and the string "12:00 - 1:00". A user can define any collection of conversion routines to allow transformations of an ADT field to different data types.

b) Comparison Routines

Since the 32 byte internal representation for time may use a peculiar coding convention, it is necessary to provide procedures to interpret the standard INGRES comparison operators {<, <=, =, >=, !=}. One can also define new comparison operators which obey the syntactic conventions of the normal comparison operators. The syntax is:

```
define comparison adt-name operator adt-name
                as procedure-name
                returning adt-name
```

For example, one could define "<" and "¦=" as follows:

```
define comparison time < time
                as my-proc-3
                returning boolean

define comparison time ¦= time
                as my-proc-4
                returning boolean
```

Notice that operators can be defined which return other types than their operands. Moreover, one can define routines which have operands which are of different types.

c) Arithmetic Operators

In order to use DBMS arithmetic for a new data type, one must be able to procedurally define the arithmetic operators {+, -, *, **, /, mod}. Also, one might want to define new operators which obey the syntactic conventions of arithmetic operators for the data type "time". The syntax is the following:

```
define arithmetic adt-name operator adt-name
                  as procedure-name
                  returning adt-name
```

For example one can define the operator "+" and a new operator "/=\" which finds the intersection of two time ranges as follows:

```
define arithmetic time + time
                  as my-proc-5
                  returning time

define arithmetic time /=\ time
                  as my-proc-6
                  returning time
```

d) Functions

In order to specify user defined functions on new data types, one requires routines providing the semantics of such functions. The syntax is the following:

```
define function function-name (adt-name)
                as procedure-name
                returning adt-name
```

For example, one can define the length of a time interval as follows:

```
define function length (time)
                as my-proc-7
                returning time
```

We can also use this facility to expand the collection of allowable operations on numeric data types.

e) Aggregate Functions

Currently INGRES supports aggregate functions such as the following:

```
avg (e.p-cancel where e.type = "food")
min (e.p-cancel by e.type)
min (e.p-cancel by e.type where e.date > "today")
```

The first computes the average cancelation probability while the second computes the minimum cancelation probability for each class of events. The last aggregate computes the cancelation probability for future events. We require a mechanism to define such aggregates for user defined types as well as the ability to define new aggregate operators. The syntax is the following:

```
define aggregate agg-name (adt-name)
                 as procedure-name
                 returning adt-name
```

The following examples define the aggregate "min" and a new aggregate "third from the smallest".

```
define aggregate min (time)
                 as my-proc-8
                 returning time

define aggregate third-low (time)
                 as my-proc-9
                 returning time
```

Using the last aggregate, we can find the third food event in the future as:

```
retrieve (e.ename) where
e.date = third-low (e.date where e.type = "food")
```

Consequently, an ADT is a registration process followed by a collection of procedure specifications of the above form. A user need only

know the calling conventions used by the data manager in order to write an ADT. In the next section we turn to the impact of ADTs on a DBMS.

3.4 Implementation Considerations

Each define command would cause an entry to be inserted into the following relation

```
ADT( adt-name = c12,
     operator-type = c12,
     operator-name = c12,
     token-value = i2,
     first-operand = c12,
     second-operand = c12,
     result-type = c12,
     procedure-name = c12)
```

For simplicity we assume that the names of ADTs are unique. Each operator defined must be assigned a token for use by the parser when representing user commands as a tree structure. The registration information concerning the internal length and processing hints can be stored in the relation which contains column information. In INGRES this is the ATTRIBUTE relation.

There are two classes of implementation difficulties, parsing and query processing. We illustrate each by use of the following command:

```
range of e is event
retrieve (e.ename) where e.time + "1 hour" < "9:15"
            and e.time > avg (length (e.time) by e.type)
```

One must first build a parse tree for this command using standard parsing techniques. Then, type mismatches must be resolved. For example, e.time + "1 hour" requires adding a character string and a time. If there is an entry in the ADT relation that has input operands of type time and character string and an operator name of "+", then one can use

the associated procedure to evaluate "+" and return a result for

```
e.time + "1 hour"
```

However, suppose no such entry exists and a data conversion must be attempted. If a conversion routine exists from character string to time, then "1 hour" can be changed to a time. Supposing that procedures exist for evaluating "+" and "<" for times, then one can replace the command with a parsed version of:

```
range of e is event
retrieve (e.name) where
e.time + input ("1 hour") < input ("9:15")
and e.time > avg (length (e.time) by e.type)
```

This command has all operands of the same type and is ready for evaluation.

On the other hand, suppose there is no routine for converting between times and character strings. In this case, one must look for some data type for which there exist conversions routines from both time and character string fields. If such a unique one exists, it is chosen as the target type and a conversion of both fields performed. If either none or more than one is found, an ambiguity exists and an error message must be issued.

This analysis of types may be very costly. Optimizing the procedure by caching portions of the ADT relation in primary memory should probably be attempted. Also, built-in rules for the standard types are probably desirable.

Query processing entails evaluating the parse tree for for database tuples either by generating code for the tree or interpreting it at run time. In the former case one must simply generate code to call the

various system and user provided routines at the appropriate times during tree evaluation, passing arguments from descendent nodes and sending the result of the procedure call up the tree. Aggregate operators must be passed the partial result which is being accumulated and the data element in the current tuple. The routine must then return a new partial result. The query processing plans in [LORI77] appear to have no difficulty with this added generality.

If the parse tree is interpreted, then the DBMS must hold a procedure library of all possible ADT routines. Otherwise, dynamic linking to a library of such procedures must be supported. This may lead to a very large run time system, discouraging user defined types.

3.5 Extensions

3.5.1 Inheritance

It is very useful to specify that a new type inherits all the operations of an existing type. This concept has been used in generalization [SMIT77], classes [MCLE78], and is-a hierarchies [MYL078]. It can be specified as follows:

```
define equal adt-name-1 is-a adt-name-2
```

For example, one might specify:

```
define equal future-time is-a time
```

3.5.2 Functions With Multiple Arguments

Certain ADTs may require functions to be defined with multiple arguments. For example, one might want to create a new time interval between two other times, i.e.:

```
make-interval( "lunch", "5:00")
```

The problem is that current relational systems do not support any functions with multiple arguments. Hence, there are no query processing heuristics or parser templates for this sort of operation. It would be a useful extension for relational systems to support such functions.

3.5.3 New Types of Aggregates

One would also like certain kinds of generalized aggregates. For example, suppose one wanted the food event that was closest to lunch, i.e.

```
retrieve (e.ename) where e.date =
     min (abs(e.date - f.date) where e.type = "food"
                                 and f.date = "lunch")
```

Although this is currently a legal aggregate, assuming that "abs", "-", and "min" are defined for times, it is clearly an awkward notation. One would prefer instead:

```
retrieve (e.ename) where e.date =
     closest(e.date TO "lunch" where e.type = "food")
```

Syntactically "closest" is similar to an aggregate function. The scope of the column being evaluated is limited by a "where" clause. Also, "closest" must keep a running tabulation of its answer. The only difference between normal aggregates and this one is the presence of a "TO" clause. The specification of more general aggregate operators and their associated query processing heuristics has yet to be investigated.

IV CONCLUSIONS

This paper has suggested the inclusion of a rules system in a relational data manager and a use of ADTs at the column level of a relation.

With a rules facility one can imagine a sophisticated user being able to extend his application environment with more elaborate integrity constraints, triggers, alerters, protection statements and view update algorithms than is possible with current relational systems.

One can also envison libraries of column ADTs and an individual installation using a subset of them in its run time system. One can also envison a sophisticated user writing his own ADTs, thereby extending a relational database system with knowledge specific to his domain of discourse.

Using these two facilities a data manager can be tailored to the needs of an individual application. It is hoped that other mechanisms can be discovered which will serve to further augment the power of contemporary data managers.

ACKNOWLEDGEMENT

This research was supported by the Naval Electronics Systems Command under Contract N00039-76-c-0022.

REFERENCES

[BUNE79] Bunemann, O. and Clemons, E., "Efficiently Monitoring Relational Databases," TODS, Sept. 1979.

[CHAM75] Chamberlin, D., et. al., "Views, Authorization and Locking in a Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

[DUDA78] Duda, R. et. al., "Development of the Prospector Consultation System for Mineral Exploration," SRI International, October 1978.

[ESWA76] Eswaren, K., "Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System," IBM Research, RJ 1820, San Jose, Ca., August 1976.

[GRIF76] Griffiths, P. and Wade, B., "An Authorization Mechanism for a Relational Data Base System," TODS, 2, 3, September 1976.

[GUTT77] Guttag, J., "Abstract Data Types and the Development of Data Structures," CACM, June 1977.

[HAMM76] Hammer, M. and McLeod, D., "A Framework for Data Base Semantic Integrity," Proc. 2nd. International Conference on Software Engineering, San Francisco, Ca., October 1976.

[LISK74] Liskov, B. and Zilles, S., "Programming With Abstract Data Types," ACM-SIGPLAN Notices, April 1974.

[LORI76] Lorie, R. and Wade, B., "The Compilation of a Very High Level Data Language," IBM Research, San Jose, Ca., RJ2008, May 1977.

[MCLE78] McLeod, D., "The Semantic Data Model and Its Associated Structural User Interface," Laboratory for Computer Science, M.I.T., October 1978.

[MYLO78] Mylopoulis, J., "A Preliminary Specification for TAXIS," CCA, Cambridge, Mass., January 1978.

[OVER81] Overmeyer, R., "A Time Expert for INGRES," Masters Thesis, University of California, Berkeley, August 1981.

[POWE82] Powell, M., private communication.

[ROWE79] Rowe, L. and Schoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass. May 1979.

[ROWE82] Rowe, L., et. al., "A Form Application Development System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fla., June 1982.

[SCHM78] Schmidt, J., "Type Concepts for Database Definition," Proc. International Conference on Data Bases, Haifa, Israel, August 1978.

[SHOR76] Shortliffe, E., "Computer Based Medical Consultations: MYCIN," Elsevier, New York, 1976.

[SMIT77] Smith, J. and Smith D., "Data Base Abstractions: Aggregation and Generalization," CACM, June 1977.

[STON74] Stonebraker, M. and Wong, E., "Access Control in a Relational Data Base System by Query Modification," Proc. 1974 ACM Annual Conference, San Diego, Ca., November 1974.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M., Retrospection on a Data Base System," TODS, September, 1980.

[STON80a] Stonebraker, M. and Keller, K., "Embedding Hypothetical Data Bases and Expert Knowledge in a Data Manager," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[STON82] Stonebraker, M., et. al., "A Rules System for a Relational Data Base System," Proc. 2nd International Conference on Databases, Jerusalem, Israel, June 1982.

h