QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson
and Brad Rubenstein

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

ABSTRACT

This paper explores the use of commands in a query language as an abstract data type (ADT) in data base management systems. Basically, an ADT facility allows new data types, such as polygons, lines, money, time, arrays of floating point numbers, bit vectors, etc., to supplement the built-in data types in a data base system. In this paper we demonstrate the power of adding a data type corresponding to commands in a query language. We also propose three extensions to the query language QUEL to enhance its power in this augmented environment.

I 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.

The use of ADTs in a relational data base context has been discussed in [ROWE79, SCHM78, WASS79]. In these proposals a relation is considered an abstract data type whose implementation details are hidden from application level software. Allowable operations are defined by procedures written in a programming language that supports both data base access and ADTs. One use of this kind of data type is suggested in [ROWE79] and involves an EMPLOYEE abstract data type with related operations hire-employee, fire-employee and change-salary.

In [STON82, STON83] we presented an alternate use of ADTs. Instead of treating an entire relation as an ADT, we suggested that the individual columns of a relation be ADTs. This use of ADTs is a generalization of data base experts [STON80].

In Section II we briefly review our proposal and then in Section III we introduce QUEL as a data type and indicate desirable operators for this new type. Section IV turns to a discussion of three extensions to the QUEL language that are useful in this environment. In Section V we consider optimization issues related to QUEL ADTs. Lastly, we indicate that several data base problems including referential integrity, non-first normal form relations, and generalization hierarchies can be solved by defining QUEL as an abstract data type. Section VI presents our approach to these problems. Section VII concludes by summarizing the paper.

This Research was supported by the Navy Electronics Systems Command under Contract N00039-83-C-0243 and by the Air Force Office of Scientific Research under Grant 83-0021.

II ABSTRACT DATA TYPES

We explain our use of ADTs with an example concerning geometric objects. In computer aided design of integrated circuits, objects are often made up of rectangular boxes. For a VLSI data base one would like to define a column of a relation as type "box". For example, one might create a boxes relation as follows:

```
create boxes (owner = i4,
              layer = c15,
              box-desc = box-ADT)
```

Here, the boxes relation has three fields: the identifier of the circuit containing the box, the processing layer for the box (polysilicon, diffusion, etc.) and a description of the box's geometry. All fields are represented by built-in types except box-desc which is an ADT.

Tuples can be appended to this relation using QUEL [STON76] as follows:

```
append to boxes (owner = 99,
                 layer = "polysilicon",
                 box-desc = "0,0,2,3")
```

The built-in data types are converted to an internal representation and stored in a data base system. The string "0,0,2,3", represents the box bounded by x=0, y=0, x=2, y=3 and requires special recognition code. An input procedure must be available to the DBMS to perform the conversion of the character string "0,0,2,3" to an object with data type box-ADT. Such a routine is analogous to the procedure ascii-to-float which converts a character string to a floating point number.

It is desirable to have special operators for box-ADTs. for example, one would clip box dimensions as follows:

```
range of b is boxes
replace b (box-desc = b.box-desc * "0,0,4,1")
        where b.owner = 99
```

The * operator represents box intersection. In this case "0,0,4,1" will be converted to an object of type box-ADT, and a procedure must be available to perform box intersection between this ADT and b.box-desc.

In addition, one might want to define new comparison operations. For example, one might wish to define || as an operator meaning "overlaps". The || operator could then be used to return the boxes overlapping the unit square based at the origin as follows:

```
range of b is boxes
retrieve (b.box-desc)
        where b.box-desc || "0,0,1,1"
```

Again, a procedure is required for the overlap operator.

As a result an ADT contains the following elements:

a) a registration procedure to inform the DBMS of the new type, giving the length of its internal representation.

b) a collection of routines which implement operators for this type and perform conversions to other types. These routines must obey a prespecified protocol for accepting arguments and returning results. Once defined by the ADT implementor, the new type and operators become available to other users of the DBMS.

c) modest changes to the parser and query execution routines to correctly parse commands with new operators and call the routines defined by the ADT implementor during execution.

This abstraction has been constructed in about 2500 lines of code for the INGRES relational data base system. Implementation details are addressed in [FOGG82, ONG82], and ADTs execute with a modest performance degradation [FOGG82]. Suggestions concerning how to integrate new operators into query processing heuristics and access methods are contained in [STON83, ONG83].

III QUEL AS A DATA TYPE

We turn now to utilizing the ADT mechanism to define commands in a query language as an ADT. Hence, a column of a relation can have values which are one (or more) commands in the data manipulation language, QUEL. We explain our proposal using the following relations:

EMP (name, salary-history, hobbies, dept, age, bonus)
DEPT (dname, floor)
SALARY (name, date, pay-rate)
SOFTBALL (name, position, average)
MUSIC (name, instrument, level)
RACING (name, auto, circuit)

A tuple exists in the EMP relation for each employee in a particular company. Employees can have zero or more hobbies. For those employees who have softball as a hobby, a tuple in the SOFTBALL relation gives their position and batting average. If an employee plays an instrument, a tuple in MUSIC indicates the instrument he plays and his skill level. Lastly, those employees who race sportcars are listed in the RACING relation along with the type of car they drive and the circuit they race on.

The SALARY relation contains employees salary histories. Each time the salary of an employee is modified, a tuple is appended to the SALARY relation indicating the date of the modification and the new pay-rate. The DEPT relation contains the floor number of each department. Lastly, the EMP relation contains three fields, salary-history, hobbies, and dept which are of type QUEL. The hobbies field holds a query (or queries) which, when executed, will yield information on the employee's hobbies. The dept field contains a query which will return the name of the department for which the employee works, and the salary-history field contains a query that finds all records in his salary history. An example insert to the EMP relation might be:

```
append to EMP (
 name = "Fred",
 salary-history = "range of s is SALARY
                   retrieve (s.all)
                   where s.name = "Fred"",
 hobbies = "range of m is MUSIC
            retrieve (m.all) where m.name = "Fred"
            range of r is RACING
            retrieve (r.all) where r.name = "Fred"",
 dept = "range of d is DEPT
         retrieve (d.dname) where d.dname = "toy"",
 age = 25,
 bonus = 10)
```

The appropriate additional insertions are:

```
append to MUSIC(
        name = "Fred",
        instrument = "piano",
        level = "novice")

append to RACING(
        name = "Fred",
        auto = "formula Ford",
        circuit = "SCCA")
```

This collection of inserts will append Fred as a new employee in the toy department with racing and music as hobbies.

In a later section we will propose an implementation of this data type. In this section we specify desirable operators this type and their intended semantics.

The current implementation of ADTs [FOGG82, ONG82] allows operators to be overloaded. INGRES currently allows "." as an operator with two operands, a tuple variable and a column name, e.g E.name. Our first ADT operator overloads the operator ".". First, we propose that "." allow a left operand which is a field of type QUEL and a right operand of type column name. For example:

```
range of e is EMP
retrieve (e.hobbies.instrument)
        where e.name = "Fred"
        and e.hobbies.level = "novice"
```

In this case "name" is a column in the relation indicated by e while "level" and "instrument" are columns in the relation (or relations) specified by the QUEL in e.hobbies. This command is interpreted as follows:

1) Find all values for e.hobbies which satisfy the qualification "e.name = "Fred".

2) For each value found, ignore all commands which it contains except RETRIEVE and DEFINE VIEW. For each RETRIEVE command which the value contains, replace the keyword RETRIEVE with the keyword DEFINE VIEW and execute it to form a legal view. For each view definition which the value contains, execute it directly to form a legal view. Then, define t to be a tuple variable which will iterate over the this collection of views. For each one, execute:

```
retrieve (t.instrument) where t.level = "novice"
```

The result of the overall query is the union of the results of the individual commands executed in step 2.

In general, if X is a tuple variable, Y is a field of type QUEL, and Z is a field, then X.Y.Z is a field in a collection of views, one for each RETRIEVE and DEFINE VIEW command contained in a qualifying value for X.Y. Moreover, "." can be arbitrarily nested and the above semantics apply recursively at each level. Also note that this use of "." is similar to that proposed in GEM [ZANI83], and we comment further on the relationship of our proposal to GEM in a later section.

Our second use of "." has a left operand which is a field of type QUEL and a right operand which is a QUEL statement, e.g.:

```
range of e is EMP
retrieve (e.salary-history.
      retrieve (date, pay-rate) where pay-rate < 400)
where e.name = "Fred"
```

Here, e.salary-history is a field of type QUEL and the inner RETRIEVE command is the right hand operand for the intervening ".". This use of "." is a short-hand notation for the equivalent expression:

```
range of e is EMP
retrieve (e.salary-history.date,
          e.salary-history.pay-rate)
where e.name = "Fred" and
e.salary-history.pay-rate < 400
```

In this nested retrieval context "." has a similar meaning to the one discussed above. In particular, the left hand operator evaluates to the collection of views mentioned earlier, and a range variable, say t, is created to iteratively span this set. The QUEL command which is the right hand operand in then executed for each view by appending t as the tuple variable to any field name which does not have an explicit variable.

When the right hand operand is a RETRIEVE command, the result of this operator is a collection of result relations. The semantics of "." when the right hand operand is a QUEL update command are unclear, and we expect to support this form of nesting only for retrieves.

We now turn to several other operators on QUEL data items. First, all the normal character string operators can be overloaded. For example:

```
range of e is EMP
retrieve (e.name) where e.dept =
        "range of d is DEPT
                retrieve (d.dname) where
                d.dname = "toy""
```

In this context, "=" simply implies character string equality between e.dept and the constant string containing the query.

Consider an operator, ==, which has two fields of type QUEL as operands and returns true if they specify the same collection of tuples. For example,

```
range of e is EMP
range of f is EMP
retrieve (e.name, f.name) where e.salary-history ==
                        f.salary-history
```

This query will return pairs of employees with identical names and salary histories. A containment operator, <<, can be specified similarly for operands which are fields of type QUEL. Additionally, all operators in a relational algebra (e.g join, union, intersection) can be easily defined between fields of type QUEL.

Any relational algebra operators will produce a result of type relation. Since QUEL allows cascaded operators, we require operators for data of type relation. It is straight forward to overload all operators for the QUEL data type to apply to data of type relation. For example to find pairs of employees with different names and the same salary history, we would execute

```
range of e is EMP
range of f is EMP
retrieve (e.name, f.name)
    where e.salary-history.
            retrieve (date, payrate)
    == f.salary-history.
            retrieve (date, payrate)
```

Here, == has relations as both operands and returns true if the two relations are equal.

The last generalization is to allow any operator for fields of type QUEL to be overloaded to apply to operands which are QUEL statements or tuple variables. For example, suppose a relation STANDARD contains a collection of dates and payrates. The following command would find all employees with the same salary history that appears in STANDARD:

```
range of e is EMP
range of s is STANDARD
retrieve (e.name)
    where e.salary-history.
            retrieve (date, payrate)
    == retrieve (s.all)
```

Here the right hand operand of == is a simple QUEL statement. A shorthand for the above statement would have a tuple variable for the right operand of == as follows:

```
range of e is EMP
range of s is standard
retrieve (e.name)
    where e.salary-history.
            retrieve (date, payrate)
    == s
```

Our complete set of proposed operators appears in Table 1. Most can be applied interchangeably to operands which are fields of type QUEL, tuple variables, QUEL statements, and relations.

IV EXTENSIONS TO QUEL

There are three main extensions which we propose for inclusion in QUEL to enhance its power in the ADT environment of Section III. In addition, we endorse the proposal made in [ZANI83] to have default tuple variables. In this situation, a command such as:

```
retrieve (EMP.age) where EMP.name = "Fred"
```

would be interpreted as:

```
range of EMP is EMP
retrieve (EMP.age) where EMP.name = "Fred"
```

This suggestion simplifies many QUEL commands and was inserted into one version of QUEL [RTI83].

operator name	description	left operand	right operand	result
.	referencing	field of type QUEL	field-name	field
.	referencing	field of type QUEL	QUEL statement	relation
=	character string compare	*	*	boolean
==	relation compare	*	*	boolean
>>	relation inclusion	*	*	boolean
<<	relation inclusion	*	*	boolean
U	union	*	*	relation
!!	intersection	*	*	relation
JJ	natural join	*	*	relation
OJ	outer join	*	*	relation

* denotes a field of type QUEL, a QUEL statement, a relation or a tuple variable

Proposed Operators

Table 1

In addition to default tuple variables we propose three other extensions. First, we suggest the possibility of executing data in the data base rather than retrieving or updating it. The syntax is as follows:

```
exec (EMP.hobbies) where EMP.name = "Fred"
```

The target list must be a field of type QUEL and instances which satisfy the qualification are found and executed. In this case, the hobbies which Fred engages in are returned.

This extension frees a user from having to know the field names in the QUEL in e.hobbies. Also, it allows one to store updates in the data base and execute them at a later time. Such data base procedures are discussed in Section VI.

Notice that EXEC complicates the extended interpretation of "." in the previous section. For example, it is reasonable to have a value for e.hobbies which is an EXEC command. For example, one could change Fred's hobbies to be the same as John's by the following update:

```
range of E is EMP
replace e (hobbies =
        "range of f is EMP
        exec (f.hobbies) where f.name = "John"")
 where e.name = "Fred"
```

If X is a tuple variable, Y is a field of type QUEL and Z is a field and if a qualifying value for X.Y contains an EXEC command, then the semantics of X.Y.Z from the previous section must be extended. In particular X.Y.Z can be a column in an additional set of views. For each EXEC contained in a qualifying value of X.Y, replace the EXEC by RETRIEVE and run the command. If the result contains values of type QUEL, then X.Y.Z must span any views which result from these values by executing DEFINE VIEW commands, replacing RETRIEVE commands by DEFINE VIEW commands and recursively applying the above meaning to EXEC commands.

The second extension is to generalize the range statement. We propose to allow a tuple variable to range over a collection of one or more relations. Then we use this facility to support the further generalization illustrated below:

```
range of e is
        EMP.salary-history where EMP.name = "Fred"
retrieve (e.date) where e.pay-rate = 1000
```

The intent is to allow e to range over the result of a query specification. Because RETRIEVE is the only reasonable QUEL command to put in a range statement, we leave it out of the syntax and include only the target list and qualification. Moreover, the query specification must return data items of type QUEL. The purpose of the second extension is to allow the above expression rather than the less natural equivalent command:

```
range of e is EMP
retrieve (e.salary-history.date)
   where e.salary-history.pay-rate = 1000
     and e.name = "Fred"
```

If X and U are tuple variables and Y a field of type QUEL, then the semantics of

```
range of U is X.Y where qualification
```

are the following:

1) Run the query

```
retrieve (X.Y) where qualification
```

to find qualifying data items of type QUEL.

2) For each RETRIEVE, DEFINE VIEW or EXEC command, perform the steps indicated earlier to define the appropriate collection of views, C1,...,Cn.

3) Replace the range statement by

```
range of U is C1,...,Cn
```

The third extension is to allow update commands to have a generalized target relation as suggested by the following example:

```
append to EMP.salary-history
   (date = "6/81", payrate = 2000, name = "Fred")
where EMP.name = "Fred"
```

Currently QUEL only supports a target which is a relation. In this generalization, the target can also be a column of a relation in the data base which is of type QUEL.

The intent of the third extension is to allow the above expression rather than the equivalent extended command:

```
range of e is
        EMP.salary-history where EMP.name = "Fred"
append to e (date = "6/81",
             payrate = 2000,
             name = "Fred")
```

Notice that extended range statements and extended targets automatically introduce views. The usual semantic problems occur in updating these views.

V SPECIAL CASES OF QUEL AS A DATA TYPE

Three special QUEL data types will be suggested in this section to allow either increased performance or a more natural syntax. First we suggest relations as a special case of the QUEL ADT. Clearly, a value of type QUEL can be a relation, i.e.:

```
range of R is any-relation
retrieve (R.all)
```

Since the interpretation of the QUEL extensions in Section IV required that the query be treated as a view, we must invoke view processing to support such functions. A data type of relation as a special case of a QUEL data type will allow such operators to be optimized by ignoring the view processing.

The internal representation of a QUEL data type may be anything from a text string for the command to a machine language procedure containing a compiled version of the access plan. The choice depends on trading off efficiency, flexibility and complexity of the underlying DBMS. Alternatively, it is also possible to precompute the answer to any RETRIEVE command. This collection of pointers to tuples would be stored as the value of the field. In the case that at most one tuple qualified, this value would be a pointer to a single tuple or the null pointer. This representation is exactly the data type "pointer to a tuple" suggested by Powell [POWE83] and by Zaniola [ZANI83]. More generally, the value could contain multiple pointers to tuples in different relations. Consequently, implementing the QUEL data type by precomputing answers for QUEL queries provides a generalized version of previous proposals. Storing such physical pointers in the data base has a clear speed advantages over storing the query. However, it also has the disadvantage that a pointer can be left "dangling" if the tuple it points to is moved. Moreover, no consistency guarantee is made if the tuple which is pointed to gets updated.

A mechanism to overcome these deficiencies is to create a new lock mode. Besides conventional read and write locks, one could support a "materialize" lock. Such a lock would be placed on any object which was used to precompute another data base object. Materialize locks would be compatible with read locks but not write locks. Moreover, any process which wished to set a write lock on an object for which a materialize lock had been previously set could "break" the materialize lock and invalidate the precomputed object. This procedure would succeed unless the precomputed object was locked. Such a policy has points in common with [BROW81] and can guaranteed data base consistency in this environment.

The third special case of a QUEL data type can be illustrated by appending a tuple to the EMP relation, e.g.

```
append to EMP (name = "Joe", dept = "shoe")
```

In this case dept is a field of type QUEL and we would prefer to simply enter the value "shoe" and not the remainder of the query. If dept is defined to be a new ADT which is special version of the QUEL ADT, then the routine which converts from external to internal format for this ADT can change "shoe" to:

```
retrieve (DEPT.dname) where DEPT.dname = "shoe"
```

Consequently, a user need not type all the extra pieces of the QUEL command.

VI USES OF QUEL AS A DATA TYPE

In this section we indicate several uses for the above facilities.

6.1 Unnormalized Relation

There has been much discussion surrounding normalization of relations, and several recent proposals have advocated unnormalized relations [HASK82, GUTT82, ZANI83]. One use of a QUEL ADT is to support hierarchical data as noted in the example use of salary-history.

6.2 Referential Integrity

The notion of referential integrity has been formalized for relational data bases in [DATE81]. Basically, a data item must take on values from the set of values in a column of another relation. Notice that our example use of the dept field in the EMP relation automatically has this property. Although not all of the options suggested in [DATE81] can be easily supported using QUEL as a data type, several of the more common ones can be.

6.3 Variant Records

Our use of queries in the hobbies field corresponds closely to the notion of variant records in a programming language such as Pascal. Frames oriented languages such as FRL [ROBE77] or KRL [BOBR77] also allow a slot in a frame to contain a value of an arbitrary type with arbitrary fields. Our use of QUEL queries with different ranges supports this notion.

6.4 Aggregation and Generalization

QUEL as a data type can support both generalization and aggregation as proposed in [SMIT77]. For example, consider:

```
PEOPLE (name, phone#)
```

where phone# is of type QUEL and is an aggregate for the more detailed values area-code, exchange and number. A simple append to PEOPLE might be:

```
append to PEOPLE (name = "Fred", phone# =
        "retrieve (area-code = 415,
                   exchange = 999,
                   number = 9911)")
```

Generalization is also easy to support. If all employees have exactly one hobby, then the hobbies field in the EMP example relation will specify a simple generalization hierarchy. In fact, our example use of hobbies supports a generalization hierarchy with members which can be in several of the subcategories at once.

6.5 Data Base Procedures

Stored commands are easily supported with the facilities described above. For example, suppose an employee is allowed to have only one hobby and we want a general data base procedure to change the hobby of an employee from playing softball to playing a musical instrument. Call this procedure "softball-to-music" and add it to a relation PROCEDURES as follows:

```
append to PROCEDURES(
name = "softball-to-music",
code = "delete SOFTBALL where SOFTBALL.name = $1
        append to MUSIC (name = $1,
                         instrument = $2,
                         level = $3)
        replace EMP (hobbies =
                "retrieve (MUSIC.all)
                where MUSIC.name = $1")
```

Now suppose we define a new ADT operator, WITH, that will substitute a parameter list given as the right hand operator into a query which is the left hand operator. With this operator we can make Fred play the violin at skill level novice as follows:

```
exec (PROCEDURES.code WITH (Fred, violin, novice))
    where PROCEDURES.name = "softball-to-music"
```

In this way we can store collections of QUEL commands in the data base and execute them as procedures.

6.6 Triggers

Triggers have been widely suggested as a possible mechanism for implementing consistency constraints and for producing side effects for commands. They can be supported by using the features discussed in previous sections. Consider a relation:

```
TRIGGER (if, relname, command, then)
```

The field "then" is of type QUEL while "if" is of type QUEL qualification. Both "relname" and "command" are ordinary character string fields.

Currently INGRES performs deferred update [STON76] and writes a "side file" containing proposed changes to the data base as phase 1 of a command. In phase 2 the side file is processed and the changes are installed. Consider modifying the side file to be a relation SIDE and interrupting query processing at the end of phase 1 to perform:

```
exec (TRIGGER.then) where TRIGGER.if
        and TRIGGER.command = user-command
        and TRIGGER.relname = relation-from-user
```

Here, user-command is the type of command run by the user (e.g. replace, delete) and relation-from-user is the name of the relation being updated. These constants are readily available from the run time DBMS.

An example tuple in the TRIGGER relation might be:

```
append to TRIGGER(
    if = "SIDE.TID = EMP.TID and EMP.name = "Fred"
          and SIDE.age > EMP.age",
    relname = "EMP",
    command = "replace",
    then = "append to ALARM
          (message = "Fred got older")")
```

The TRIGGER relation is used to provide an alerting capability when Fred receives an update. Since TRIGGER may have a large collection of tuples, we require indexing on relname and command to restrict the set of TRIGGER.if terms that must be evaluated. It may be reasonable to have other extra fields in TRIGGER to provide further efficiency in TRIGGER selection.

6.7 Storing Data as Rules

Consider the requirement that all employees over 40 years old must receive a bonus of 1000. The relation in Section II showed both "age" and "bonus" as explicit data and an integrity constraint could easily be defined to enforce this constraint, e.g.:

```
range of e is EMP
define integrity E where E.bonus = 1000 or E.age <= 40
```

However, an alternative representation would be to remove "bonus" as a stored field in EMP and add the following rule to TRIGGER:

```
append to TRIGGER(
      relname = "EMP"
      then  = "replace SIDE( bonus = 1000)
              where SIDE.TID = EMP.TID and
                 EMP.age > 40"
```

If the QUEL parser was changed to allow queries that retrieve fields which are not stored, then this trigger will return the correct data by updating SIDE. Hence, the trigger mechanism can support storing data items as rules. Of course, the efficiency of this implementation is questionable, and it is awkward to ask questions about what rules are in effect.

6.8 Complex Objects

There has been substantial discussion concerning data base support for complex objects [LORI83, STON83].

Suppose a complex object is composed of text, lines, and polygons. It would be possible to construct the following relations:

OBJECT (Oid, description)
LINE (Lid, description, location)
TEXT (Tid, description, location)
POLYGON (Pid, description, location)

Here, the LINE, TEXT and POLYGON relations hold descriptions of individual objects and can make use of the abstract data types described in [STON83]. Then, the description field in OBJECT would be of type QUEL and contain queries to assemble the pieces of any given object from the other relations. This representation allows clean sharing of Lines, Text and Polygons among multiple higher level objects by allowing the same query to appear in multiple object descriptions.

Materializing an object from the OBJECT relation will be slow since it involves executing several additional QUEL queries. Hence, it is clearly desirable to precompute the value of frequently used objects and store the result in the OBJECT description field.

6.9 Transitive Closure

The facilities of this paper can be used to support transitive closure operations such as found in the "parts explosion" problem. Suppose one creates a PARTS relation as follows:

PARTS (pname, composed-of)

Consider a car which is made up of a drivetrain and a body. These are made up in turn of other smaller parts. The car would be inserted as follows:

```
append to PARTS(
pname = "car",
composed-of = "retrieve (pname = "car")
                exec (PARTS.composed-of)
                  where PARTS.pname = "drive-train"
                exec (PARTS.composed-of)
                  where PARTS.pname = "body"")
```

The command

```
exec (PARTS.composed-of)
where PARTS.pname = "car"
```

will find all the parts that make up a car.

VII IMPLEMENTATION

If INGRES had been designed to support internal multitasking, then it would be a simple matter to implement EXEC by stacking the INGRES processing environment and executing the new command in a single INGRES process. However, at this point it would be very costly to change our code to be reentrant and support this kind of recursion. Other systems (e.g. System-R [ASTR76]) do not have this shortcoming.

Hence, our operational code to implement EXEC spawns a separate copy of the INGRES code and passes the QUEL command to the spawned version for execution. Returned data is redirected through the INGRES which did the spawning to the user who ran the original command. Since the passed command can be another EXEC, the total number of spawned INGRES's can increase without bound. Currently, the command is passed to the spawned process as a character string and all query processing steps are performed at run time by the second process.

We are currently implementing QUEL as an ADT. This data type is internally represented as a character string. Storing a preprocessed version of the command would entail a great deal more code. Operators which return a result of type relation will store the result in the data base and return the name of the object. This result can be involved in further processing or returned to the user. In the latter case, it is the responsibility of the internal-to-external conversion routine to accept the relation name, access the data base and return tuples to the calling program or user.

No thought has been given on how to optimize QUEL commands extended with the operators of Table 1. Integrating these new functions into query processing heuristics is left for future research. The design of a programming language interface supporting the objects generated by our proposal also remains to be studied.

VIII CONCLUSIONS

This paper has proposed a novel use of abstract data types and extended QUEL with three additional features. These extensions support added power, referential integrity, variant records, data base procedures, generalization and aggregation in a single facility.

Our proposal has points in common with GEM which supports new data types corresponding to "pointer to a tuple" and "set of values". Moreover, generalization hierarchies are supported and range variables can conveniently be defined over entities in this hierarchy. Our proposal effectively supports both of GEMs new data types as special cases of the QUEL ADT. Moreover, generalization is cleanly supported. Only GEM's use of range variables is not contained in our proposal.

REFERENCES

[ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM TODS, June 1976.

[BOBR77] Bobrow, D. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language," Cognitive Science, 1,1 1977

[BROW81] Brown, M., "The Cedar Database System," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., June 1981.

[DATE81] Date, C., "Referential Integrity," Proc. 6th VLDB Conference, Cannes, France, September 1981.

[FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, Ca Sept. 1982.

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

[GUTT82] Guttman, A. and Stonebraker, M., "Using a Relational Database Management System for Computer Aided Design Data", Data Base Engineering, June 1982.

[HASK82] Haskins, R. and Lorie, R., "On Extending the Functions of a Relational Database System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl, June 1982.

[LORI83] Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design

Transactions," Proc. Engineering Design Applications of ACM-IEEE Data Base Week, San Jose, Ca., May 1983.

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

[ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, Ca Sept. 1980.

[ONG83] Ong, J., et. al., "Implementation of Data Abstraction in the Relational Database System INGRES," to appear in SIGMOD Record.

[POWE83] Powell, M. and Linton, M., "Database Support for Programming Environments," Proc. Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983.

[RTI83] "INGRES Reference Manual, Version 1.4," Relational Technology, Inc., Berkeley, Ca., 1983.

[ROBE77] Roberts, R. and Goldstein, I., "The FRL Manual," MIT, AI Laboratory, Memo No. 409, Sept 1977.

[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.

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

[SMIT77] Smith, J and Smith, D., "Database Abstractions: Aggregation and Generalization," ACM TODS, June 1977.

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

[STON80] Stonebraker, M., and Keller, K., "Embedding Expert Knowledge and Hypothetical Data Bases in a Data Base System," Proc 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[STON82] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," Proc. NSF Workshop on Semantic Modeling, Intervale, N.H. June 1982 (to appear as Springer-Verlag book edited by M. Brodie).

[STON83] Stonebraker, M., et. al. "Application of Abstract Data Types and Abstract Indices to CAD Databases," Proc. Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983.

[WASS79] Wasserman, A.I., "The Data Management Facilities of PLAIN," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[ZANI83] Zaniolo, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.