EXTENDING A DATA BASE SYSTEM WITH PROCEDURES

Michael Stonebraker, Jeff Anton and Eric Hanson

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper suggests that more powerful data base systems (DBMS) can be built by supporting data base procedures as full fledged data base objects. In particular, allowing fields of a data base to be a collection of queries in the query language of the system is shown to allow complex data relationships to be naturally expressed. Moreover, many of the features present in object-oriented systems and semantic data models can be supported by this facility.

In order to implement this construct, extensions to a typical relational query language must be made and considerable work on the execution engine of the underlying DBMS must be accomplished. This paper reports on the extensions for one particular query language and data manager and then gives performance figures for a prototype implementation. Even though the performance of the prototype is competitive with that of a conventional system, suggestions for improvement are presented.

1. INTRODUCTION

Most current data base systems store information only as data. However older data base systems (e.g. [DBTG71]) specifically allowed data base procedures written in a general purpose programming language to be called during command execution. Moreover, Lisp [WILE84] supports objects which are interchangeably either procedures or data. In this paper we suggest that supporting a restricted form of data base procedures in a DBMS allows complex data base problems to be easily and naturally addressed. In particular, we propose that a field in a data base be allowed to have a value which is a collection of commands in the query language supported by the DBMS (e.g. SQL [SORD84] or QUEL).

Our proposal should augment a field-oriented abstract data type (ADT) facility (e.g. [ONG84]). Such an ADT capability appears useful for supporting relatively simple objects which do not require shared subobjects (e.g. lines, points, complex numbers, etc.). On the other hand, data base procedures are attractive for more complex objects, possibly with shared subobjects (e.g. forms, icons, reports, etc.).

We begin in Section 2 by presenting the data definition facilities for procedural data along with several examples of the use of this construct. Then, in Section 3 we review briefly how to extend one query language with necessary facilities to use procedures. Our choice is QUEL [STON76], but the extensions are easy to map into most other relational query languages. The definition of this language, QUEL+, is indicated in Section 3 and is based on suggestions in

---

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

[STON84]. Substantial changes to the query execution code of a data base system are required to process QUEL+. In Section 4 we indicate the changes that were necessary to support our constructs in the University of California version of INGRES [STON76]. Then, in Section 5 the performance of our prototype on several problems with complex data relationships is indicated. Lastly, Section 6 discusses ways in which the performance of the prototype could be improved.

2. DATA BASE PROCEDURES

The motivation behind using procedures as full-fledged data base objects was to retain the ‘‘spartan simplicity’’ of the relational model, while allowing it to address situations where it has been found inadequate. Such situations include generalization, aggregation, referential integrity, transitive closure, complex objects with shared subobjects, stored queries, and objects with unpredictable composition. The main advantage of our approach is that a single mechanism can address a large class of recognized deficiencies. We discuss the data definition capabilities of our proposal along with examples of its application to some of the above problems in the remainder of this section.

2.1. Objects with Unpredictable Composition

The basic concept is that a field in a relation can have a value consisting of a collection of query language commands. Consider, for example, a conventional EMP relation with the requirement of storing data on the various hobbies of employees. Three relations containing hobby data might be:

```
SOFTBALL (emp-name, position, average)
SAILING    (emp-name, rating, boat-type, marina)
JOGGING  (emp-name, distance, best-time, shoe-type, number-of-races)
```

Each gives relevant data for a particular hobby. For example, Smith could be added as the catcher of the softball team by:

```
append to SOFTBALL (name = ‘‘Smith’’, position = ‘‘catcher’’, average = 0)
```

The desired form of the EMP relation would be:

```
create EMP (name = c10, age = i4, salary = f8, hobbies = procedure)
```

Then, for example, Smith could be added as an employee by:

```
append to EMP (
                name = ‘‘Smith’’
                age = 40
                salary = 10000,
                hobbies = ‘‘retrieve (SOFTBALL.all)
                         where SOFTBALL.name = ‘‘Smith’’’’
               )
```

In this case, the first three values are conventional fields while the fourth is a field of data type ‘‘collection of commands in the query language’’. The value of this last field is obtained by executing the command (s) in the field. As such the ultimate value of each hobbies object is an arbitrary collection of records of arbitrary composition. A procedural field has the flexibility to model environments where there is no predetermined structure to objects. A second example of the need for procedural fields is indicated in the next subsection.

2.2. Stored Queries

Most data base systems which preprocess commands in advance of execution (e.g. System R [ASTR76] and the IDM [EPST80]) store access plans or compiled code in the data base system. Such systems already manage a data base of compiled queries. Their implementations would become somewhat cleaner if data base commands became full-fledged data base objects. For example, the precompiler for a programming language could run a conventional APPEND command to insert a tuple into the following relation for each data base command found in a user program:

TODO (id, command)

Then, at run time the program would use the EXECUTE command to be introduced in Section 3:

execute (TODO.command) where TODO.id = value

To substitute parameters into such a command, one requires an additional operator ‘‘with’’ to specify:

execute (TODO.command with param-list) where TODO.id = value

In this way, the compile-time and run-time interfaces to the data base system are the same, resulting in a more compact implementation.(**) Moreover, in Section 6 we discuss how to asynchronously build query processing plans for user commands between the time that the preprocessor inserts then in the TODO relation and the time that the user executes them. Hence, there is no performance penalty to our approach compared to current technology. In fact, our approach may well run faster because in Section 6 we also propose caching the answers to commands as well as their execution plan.

A second use of stored queries is to support the definition of relational views. Each view can be stored as a row in a VIEW relation as follows:

VIEW (name, query)

Here, the retrieval command that defines the view can be stored in the ‘‘query’’ field while the name of the view is stored in the ‘‘name’’ field. The query modification facilities of [STON75] are needed to support the extensions that we propose to a query language in the next section; consequently, it will be seen that views require very little special case code if implemented as procedural fields.

Lastly, many applications require the ability to store algorithms made up of data base commands in the data base. An example of this kind of application is [KUNG84]. Our proposal contains exactly the facilities needed in such environments.

2.3. Complex Objects with Shared Subobjects

Another example where procedures are helpful is in modeling of complex objects. Suppose an object is composed of text, line segments, and polygons and is represented in the following relations:

```
OBJECT    (Oid, text, shape)
LINE      (Lid, l-desc)
TEXT      (Tid, t-desc)
POLYGON   (Pid, p-desc)
```

Subcomponents of objects would be inserted into the LINE, TEXT or POLYGON relation, and we assume that l-desc and p-desc are of type ‘‘line’’ and ‘‘point’’ respectively and utilize a field-oriented ADT facility (e.g. [ONG84]). For example:

```
append to LINE (Lid = 22,
                l-desc = ‘‘(0,0) (14,28)’’)
```

** Of course, authorization must be done for the above command to support access control. It would be beneficial to avoid reauthorizing a command each time it is executed from an application program. A mechanism to accomplish this task is beyond the scope of this paper.

```
append to POLYGON (Pid = 44,
                   p-desc = ``(1,10) (14,22) (6,19) (12,22)'')
append to TEXT (Tid = 16,
                t-desc = ``the fox jumped over the log'')
```

Then, the ‘‘text’’ and ‘‘shape’’ fields of OBJECT would be of type procedure, and each tuple in OBJECT would contain queries to assemble a specific object from pieces stored in the other relations. For example, the following query would make object 6 be composed of all line segments with identifiers less than 20, polygon 44, and the first 9 text fragments.

```
append to OBJECT( Oid = 6,
                  shape = ``retrieve (LINE.all) where LINE.Lid < 20
                           retrieve (POLYGON.all) where POLYGON.Pid = 44'',
                  text = ``retrieve (TEXT.all) where TEXT.Tid < 10'')
```

Notice that sharing is easily accomplished by inserting queries into multiple ‘‘shape’’ or ‘‘text’’ fields which reference the same subobject.

Additional examples of complex objects include forms (such as found in a system like FADS [ROWE82]), icons, reports, and complex geographic objects (e.g. a plumbing fixture which makes a right angle bend).

When objects can have a variety of subobjects and those subobjects can be shared, most contemporary modelling ideas are flawed. For example, the proposal of [HASK82, LORI83] does not easily allow shared subobjects. Semantic data models (e.g. [HAMM81, MYLO80, SHIP81, SMIT77, ZANI83]) lack the flexibility to deal with uncertain structure. The proposal of [COPE84] allows sharing by storing subobjects as separate records and connecting them with pointer chains. Our sharing is accomplished without requiring a specialized low level storage manager, and we will show in Section 6 how caching can be used to make performance competitive with pointer based proposals.

2.4. Generalizations to Arbitrary Procedures

Our proposal should be easily generalizable to procedures written in a general purpose programming language. An example that can utilize more general procedures is a graphics application that wishes to store icons in the data base (e.g. [KALA85]). Icons should be stored in human readable form, so their description can be browsed easily. However, display software requires icons to be converted into a display list for a particular graphics terminal. An icon could be a complex object, and its components assembled by a query. However, the components must then be turned into a display list by a procedure in a general purpose programming language which appears in an application program. Efficiency can be gained by caching icons as noted in Section 6; however, further efficiency results from caching the actual display list. Such a capability requires general procedures rather than just data base procedures.

A second example of the need for general procedures is in the support for extended data type proposals (e.g. [ONG84]) They require user-defined procedures to implement new operators. Such procedures must be called by the DBMS as appropriate, and it would be more natural if they were full fledged data base objects.

A last example of the use of general procedures would be in the system catalogs of a typical relational data base system where the following two relations appear.

```
RELATION (relation-name, owner, ...)
ATTRIBUTE (relation-name, attribute-name, position, data-type, ...)
```

Whenever a relation with N attributes is ‘‘opened’’, a ‘‘descriptor’’ must be built by accessing one tuple in RELATION plus N tuples from the ATTRIBUTE relation. In order to allow ‘‘browsing’’ of the system catalogs, it is desirable to store the catalogs in the above fashion; however the penalty is the lengthy time required to open a relation.

An alternate solution is to add a procedural field to RELATION, e.g:

RELATION (relation-name, owner, ... , descriptor)

The ‘‘descriptor’’ field contains queries to retrieve the appropriate tuples from the ATTRIBUTE relation and the current tuple from the RELATION relation. These queries are surrounded by code in a general purpose programming language to build the actual descriptor in the format desired by the run time system.

In Section 6 we will discuss a technique that allows the value for a procedural field to be cached in the field itself. If this is accomplished, then the N accesses to the ATTRIBUTE relation are avoided, and the descriptor can be accessed directly from the RELATION relation. Writes to tuples in the ATTRIBUTE relation which make up an object (an infrequent event) will cause the cached value to be invalidated as explained in Section 6. The next time a relation is opened, the contents of the cached value must be reassembled.

Alternate implementations of complex objects (e.g. [COPE84]) store subobjects as individual records. Hence, pointers must be followed to assemble a composite object. Sophisticated clustering will be required to avoid extra disk reads in this environment. Moreover, if subobjects are shared, it will be impossible to guarantee clustering. Our caching implementation should offer superior performance to one based on pointers when updates are infrequent. It should be noted, however, that our caching idea can be applied to any DBMS to improve performance. Hence, a pointer based DBMS that also implemented caching might be an attractive alternative.

We now turn to a special case of procedural data types and indicate its utility.

2.5. Referential Integrity

Consider the standard EMP and DEPT example as follows:

EMP (name, age, salary, dept)
DEPT (dname, floor, budget)

Here, one often wants to guarantee that the values that occur in the column ‘‘dept’’ of EMP are a subset of the values that occur in the field ‘‘dname’’ in DEPT. This concept has been termed **referential integrity** in [DATE81] and occurs because ‘‘dept’’ is, in effect, a pointer to a tuple in DEPT and is represented by a foreign key.

Procedural data can alleviate the need for special case syntax and implementation code to support referential integrity in the following way. Suppose the ‘‘dept’’ field for each employee in the EMP relation contains the following procedure:

retrieve (DEPT.all) where DEPT.dname = ‘‘the-appropriate-dept’’

In this case the following semantics are automatically enforced. Whenever, an employee is hired and assigned to a non-existent department, then the procedure in the ‘‘dept’’ field evaluates to null, and the employee is effectively placed in the null department. Moreover, whenever a department is deleted from the DEPT relation, then all employees who were previously in that department now have a procedural field which evaluates to null and are thereby placed in the null department. Although [DATE81] has several other options, procedural data captures the main thrust of that proposal.

Notice that all fields in the ‘‘dept’’ column have the same basic query as their value, differing only in the constant used in the qualification. Consider an implementation of this special case whereby the parameterized command(s) is stored in the system catalogs and only the parameter(s) stored in the field itself. Hence, in the example above, only the department name of the employee’s department would appear in the field ‘‘dept’’, while the remainder of the query:

retrieve (DEPT.all) where DEPT.dname = parameter-1 would appear in the system catalogs. Moreover, an update to the ‘‘dept’’ field would only need to specify the parameter and not the entire query, e.g:

append to EMP (name = ‘‘Joe’’, age = 25, salary = 10000, dept = ‘‘shoe’’)

To specify this special case syntactically, one could proceed in two steps. First, one could **register** the procedure containing the parameter(s) with the data manager and give it some internal name, say DEPARTMENT, with the following command:

define DEPARTMENT as retrieve (DEPT.all) where DEPT.dname = parameter-1

Then, one could create the EMP relation as:

create EMP (name = c10, age = i4, salary = f8, dept = DEPARTMENT)

Alternatively, one could avoid the registration step for commonly used procedures such as the one above by accepting the following syntax:

create EMP (name = c10, age = i4, salary = f8, dept = DEPT[dname])

The syntactic token DEPT[dname] signifies that the procedure

retrieve (DEPT.all) where DEPT.dname = parameter-1

should be automatically defined and associated with the ‘‘dept’’ field.

The data type ‘‘pointer to a tuple’’ suggested in [POWE83, ZANI83] can be effectively supported by another special case. Suppose each relation automatically contains a unique identifier (UID), a feature commonly requested in some environments. Moreover, suppose in the syntax:

create EMP (name = c10, age = i4, salary = f8, dept = DEPT)

the DEPT token is automatically associated with the query:

retrieve (DEPT.all) where DEPT.UID = parameter-1

In this way procedures can be used to support the capability that a field in one relation can be a uniquely identified tuple in another relation.

2.6. Aggregation and Generalization

Procedural fields can support both generalization and aggregation as proposed in [SMIT77]. For example, consider:

PEOPLE (name, phone#)

where phone# is of type procedure and is an aggregate for the more detailed values area-code, exchange and number. As such, the following parameterized procedure can be used for the phone# field:

retrieve (area-code = parameter-1, exchange = parameter-2, number = parameter-3)

A simple append to PEOPLE might be:

append to PEOPLE (name = ‘‘Fred’’, phone# = ‘‘415-841-3461’’)

Here, ‘‘-’’ is the assumed separator between the values of the three parameters.

Generalization is also easy to support. If all employees have exactly one hobby, then the hobbies field in the example EMP relation from Section 2.1 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.

2.7. Summary

In summary, data base procedures are a high leverage construct. Not only can they be used to simulate a variety of semantic data modelling ideas such as generalization and aggregation, but also they can be used to support objects that have unpredictable composition and shared subobjects. In addition, they are useful in simplifying the design of current relational systems by allowing a more uniform treatment of compiled queries and views. Lastly, support for procedures written in an arbitrary programming language is a natural and valuable extension, and a preliminary proposal in this direction appears in [STON86]. Hence, a single construct is useful in a wide variety of circumstances.

3. THE QUERY LANGUAGE, QUEL+

In order to make procedures a useful construct, several extensions must be made to QUEL and these are indicated in the next several subsections. This language, QUEL+, contains slight modifications to the facilities proposed in [STON84], and a concise summary of its extensions to QUEL appears in Appendix 1.

3.1. Execution of the Data

A procedural field can be interpreted in two ways, namely it has a **definition** which is the QUEL code in the field and a **value** which is obtained by executing the QUEL commands. Since a user needs to gain access to both representations, we use the convention that a normal retrieval returns the definition. For example, the query:

retrieve (EMP.hobbies) where EMP.name = ‘‘Smith’’

will return a collection of QUEL commands. Execution of a procedural field is accomplished by an additional QUEL+ command which allows one to execute data in the data base. For example, one can find all the hobby data for Smith by running the following command:

execute (EMP.hobbies) where EMP.name = ‘‘Smith’’

This command will search for qualifying tuples and then execute the contents of the hobbies field.

Two points should be noted about the above command. First, notice that a user program must be prepared to accept the tuples returned from the above query. Since the composition of these tuples may vary from tuple to tuple, the run time system must send output to an application program using a more complex format than often used currently. In particular, each tuple must either be self-describing or a tuple descriptor must be sent to the application which describes all subsequent tuples until a new descriptor is sent. Run time support code in the application program must be prepared to accept this more complex format and deal with the more complex buffering and communication with variables in an application program that this entails. Second, a user must note which fields contain procedural data, since retrieving a procedural field does not yield the ultimate data value. We considered automatic evaluation of procedural fields, but this option requires a second operator to ‘‘unevaluate’’ the procedure and seemed no more user-friendly. Also, it would have required the application program to accept unnormalized relations. For example, automatic evaluation of procedural fields for the query:

retrieve (EMP.name, EMP.hobbies) where EMP.age > 35

would yield an unnormalized relation as a result.

In some applications, it is desirable to execute only one of a collection of qualifying tuples. The following command will execute the hobby description for one employee over 70.

execute-one (EMP.hobbies) where EMP.age > 70

The intent of this command is that query processing heuristics along the lines of [SELI79] would be run on each candidate hobby description. The one with the expected least cost would be selected for execution. The use of this construct in a particular expert system application is discussed in [KUNG84].

3.2. Multiple-Dot Notation

Our second extension to QUEL allows the components of a complex object to be addressed directly. For example, one could retrieve the batting average of Smith as follows:

```
retrieve (EMP.hobbies.average) where EMP.name = ‘‘Smith’’
```

This **multiple-dot** notation has many points in common with the data manipulation language GEM [ZANI83], and allows one to conveniently access subsets of components of complex objects. More exactly, QUEL+ allows an indirectly referenced column name of the form:

```
relation.column-name-1.column-name-2 ... column-name-n
```

wherever a normal column name:

```
relation.column-name
```

is allowed in QUEL. The only restriction is that ‘‘column-name-i’’ must be a procedural data type for 1 <= i < n-1. Moreover, column-name-(i+1) is a column in any relation specified by a RETRIEVE command contained in the field specified by column-name-i. Of course, the same construct is allowed for relation surrogates (tuple variables).

The above QUEL+ command returns the average of Smith for any hobby that has a field with name ‘‘average’’. Since there may be several hobbies with this field defined, one requires a notation to restrict the average only to the SOFTBALL relation. This is easily accomplished with another operator, i.e:

```
retrieve (EMP.hobbies.average)
where EMP.name = ‘‘Smith’’
and EMP.hobbies.average in SOFTBALL
```

Here ‘‘in’’ expects an indirectly referenced column name as the left operand and a relation name as the right operand and returns true only if the column is in the indicated relation. Additional operators associated with procedural objects may be appropriate and will be added to QUEL+ as a need arises.

3.3. Extended Scoping

To change the position of Smith from catcher to outfield, one could make a direct update to the SOFTBALL relation. However, it is sometimes cleaner to allow the update to be made through the EMP relation as follows:

```
replace EMP.hobbies (position = ‘‘outfield’’) where EMP.name = ‘‘Smith’’
```

The desired construct is that a procedural field (in this case EMP.hobbies) can appear as the target of a DELETE, REPLACE or APPEND command. In general, this procedural field is identified by an arbitrary multiple-dot expression of the form discussed in the previous section, and we term this expression the **scope** of the update.

The semantics of an **extended scope** command are that the RETRIEVE commands in the procedural field used as the target of the update command define conventional relational views. Once a specific instance of such a procedural field has been identified, for each view, Vi, associated with a RETRIEVE command, Ri, one need only replace the the update scope by Vi in every place it appears in the user command, and then standard query modification [STON75] using Ri should be performed on the qualification and the target list of the resulting user’s command.

For example, if Smith's ‘‘EMP.hobbies’’ field contains the single query:

retrieve (SOFTBALL.all) where SOFTBALL.name = ‘‘Smith’’

then the above command to move Smith to the outfield will have the form

replace EMP.hobbies (position = ‘‘outfield’’)

once the clause

where EMP.name = ‘‘Smith’’

has been evaluated to identify a specific ‘‘EMP.hobbies’’ value. Hence, this query is turned into:

replace V1 (position = ‘‘outfield’’)

and then query modification converts it to:

replace SOFTBALL (position = ‘‘outfield’’) where SOFTBALL.name = ‘‘Smith’’

Notice that this construct allows a very simple means for supporting relational views. If the definition of each view appears in the VIEW relation as suggested in the previous section, e.g:

VIEW (name, query)

then any command involving a view, V, need only be modified to replace every reference to V with VIEW.query and then the clause

VIEW.name = V

must be added to the qualification. The resulting command will be one containing multiple-dot clauses and extended scoping statements and can be executed as a conventional QUEL+ command.

3.4. Extended Scoping with Tuple Variables

In addition to allowing the above construct, QUEL+ also allows a tuple variable to be used whenever a relation name or a field of type QUEL is permissible. Hence, the example above can also be expressed as:

range of e is EMP.hobbies
replace e (position = ‘‘outfield’’) where EMP.name = ‘‘Smith’’

3.5. Relation Level Operators

In addition, QUEL+ supports relation level operators, including union, intersection, outer join, natural join, containment and a test for emptiness. We illustrate the use of this construct with an example from the previous section where objects were made up of lines, polygons, and text fragments. In this situation, one might want to find all pairs of objects, one of which contains all the shapes in the other. This would be formulated as:

range of o is OBJECT
range of o1 is OBJECT
retrieve (o.Oid, o1.Oid) where o.shape >> o1.shape

Here, the containment operator >>, accepts two procedural operands and returns true if the relation specified by the procedure in the left operand includes the relation specified by the procedure in the right operand. The relation on the left is found by constructing the outer union defined by the RETRIEVE commands in o.shape. If all commands have identical target lists, then the outer union is the same as a normal union. Otherwise, it is formed by constructing a relation with all columns appearing in any command, filling each target list with nulls to be the full width of the composite relation, and then performing a normal union. This resulting relation must be compared for set inclusion with the relation to which o1.shape evaluates. Our initial collection of operators is indicated in Table 1.

4. PROCESSING QUEL+

The purpose of this section is to explain how our existing prototype executes QUEL+ commands. This prototype supports the complete language noted in the previous section with the exception of execute-one and extended scoping statements. Moreover, it only implements general QUEL procedural fields. The optimization routines to support the special case that all queries in a given column differ only by a collection of parameters have not yet been implemented.

Although more sophisticated query processing algorithms have been constructed [SELI79, KOOI82], our implementation builds on the original INGRES strategy [WONG76]. The implementation of QUEL+ has been accomplished using this code because it is readily available for experimentation. Integration of our constructs into more advanced optimizers appears straightforward, and we discuss this point again at the end of this section.

Figure 1 shows a diagram of the extended decomposition process. Detachment of one-variable queries that do not contain multiple-dot or relation level operators can proceed as in the original INGRES algorithms [WONG76]. Similarly, the reduction module of decomposition is unaffected by our extensions to QUEL. In addition, tuple substitution is performed when all other processing steps fail. A glance at the left hand column of Figure 1 indicates that a test for zero variables must be inserted into the original flow of control after the reduction module. Then, new facilities must be included to process the ‘‘yes’’ branch of the test. These include a test for whether there is a relation to materialize and the code to perform this step. Lastly, the one-variable query processor must be extended to process relation level operators. We explain these extensions with a detailed example.

The desired task is to find the polygon descriptions with identifiers less than 5 for all objects which have the same collection of shapes as the complex object with Oid equal to 10, i.e:

```
range of o is OBJECT
range of o1 is OBJECT
retrieve (o.shape.p-desc)
where o.shape.Pid < 5
and o.shape == o1.shape
```

Operator	Function
U	union
!!	intersection
>>	containment
<<	containment
==	equality
<>	inequality
JJ	natural join on all common column names
OJ	outer (natural) join
empty	emptyness

Relation Level Operators

Table 1

```
|-----------------
|                  |
|                  V
|       -------------------------------
|       | extract and process one  |
|       | variable clauses which    |
|       | do not contain relation   |
|       | level or multiple-dot     |
|       | operators                 |
|       -------------------------------
|                  |
|                  V
|       -----------------------
|       | apply reduction |
|       |  algorithm       |
|       -----------------------
|                  |
|                  V
|       ---------------------------              ---------------------------
|       | is the qualification   |    yes        | are there relations   |
|       | free of variables?     |----------->| to materialize?        |
|       ---------------------------              ---------------------------
|              no |                              yes |          | no
|                 |                    -------------|          |
|                 |                    |                       |
|                 V                    V                       V
|       ---------------------     -----------------      ----------------------
|       |  do tuple       |       | materialize a |       | pass to extended     |
|       | substitution   |        | relation       |      | OVQP for relation  |
|       ---------------------     -----------------      | level operator       |
|                 |                    |                | evaluation            |
|                 |                    |                 --------------------
|                 |                    |                         |
|                 V                    V                         V
-----------------------------------------------------------------------------------
```

Extended Decomposition

Figure 1

and o1.Oid = 10

In the initial step of the reduction process the last clause in the query is found to have a single variable, so it can be executed as:

retrieve into TEMP-1 (o1.shape) where o1.Oid = 10

The original query is now:

retrieve (o.shape.p-desc) where o.shape.Pid < 5 and o.shape == TEMP-1.shape

The first clause above contains a multiple-dot attribute and should not be processed until later. At this point reduction fails and the query still has two variables in it, so processing falls through to the tuple substitution module. If TEMP-1 is selected for substitution, the resulting query is:

retrieve (o.shape.p-desc)
where o.shape.Pid < 5
and o.shape == ‘‘QUEL-constant-1’’

Notice that the variable ‘‘TEMP-1.shape’’ has been replaced by a constant ‘‘QUEL-constant-1’’ which is a collection of QUEL commands. Processing now returns to the top of the loop where the query still does not have any one-variable clauses. Processing again returns to tuple substitution where the variable o might be chosen. This results in the query:

retrieve (‘‘QUEL-constant-2’’.p-desc)
where ‘‘QUEL-constant-2’’.Pid < 5
and ‘‘QUEL-constant-3’’ == ‘‘QUEL-constant-1’’

Notice that o.shape has been replaced by two constants ‘‘QUEL-constant-2’’ and ‘‘QUEL-constant-3’’ which are identical. When o.shape is materialized, there will be a one-relation clause (o.shape.Pid < 5) that can be used to restrict and project the relation. Moreover, it is desirable to check this clause as early as possible because the current query will have no answer if this clause is false. On the other hand, o.shape must be retained as a complete object so that the the relation level comparison with QUEL-constant-1 can be performed if necessary. In order to avoid forcing the relation level operator to be executed first, we have duplicated the QUEL constant and thereby retained the option of performing the one-variable restriction first. Even though QUEL-constant-2 and QUEL-constant-3 define the same object, the caching discussed in Section 6 should avoid materializing this object more than once.

Now the command has zero variables and is passed to the materialize module. This processing step chooses one of the QUEL constants and materializes the outer-union of the RETRIEVE commands into a relation TEMP-2. If ‘‘QUEL-constant-2’’ is chosen, then the resulting query will be:

retrieve (TEMP-2.p-desc) where
TEMP-2.Pid < 5
and ‘‘QUEL-constant-3’’ == ‘‘QUEL-constant-1’’

This query now has a one-variable clause which can be detached and processed creating another temporary relation TEMP-3. If TEMP-3 is empty then the query is false and can be terminated. Alternately, processing must continue on the following command:

retrieve (TEMP-3.p-desc) where ‘‘QUEL-constant-3’’ == ‘‘QUEL-constant-1’’

The qualification is again free from variables, so another relation must be materialized. If ‘‘QUEL-constant-1’’ is chosen, we obtain:

retrieve (TEMP-3.p-desc) where ‘‘QUEL-constant-3’’ == TEMP-4

The qualification is still free from variables, so the final relation must be materialized as follows:

retrieve (TEMP-3.p-desc) where TEMP-5 == TEMP-4

After another trip around the processing loop, no further materialization is possible. Hence, the query must now be passed to the one-variable query processor. This module will process the operator == for the two relations involved.

Several comments are appropriate at this time. First, this algorithm delays materializing a relation until there is no conventional processing to do. In addition, it delays evaluating relation level operators until there is nothing else to do. This reflects our belief that expensive operations should never be done until absolutely necessary. The current prototype only materializes a procedural field if the desired columns actually appear in the result. This tactic avoids obviously unnecessary materializations. However, no attempt has been made to materialize only a subset of a procedural object by using qualification in the user command to advantage. For example, only the tuples where Pid < 5 could have been materialized from the query in ‘‘QUEL-constant-2’’ by modifying the qualification. Such restricted materializations would not allow the caching that we have in mind, and we did not consider them. A more sophisticated query planner would try to optimize the decision of whether to materialize the value of the whole procedural object or a qualified subset.

Second, most current optimizers build a complete query plan in advance of executing the command. Such optimizers (e.g. [SELI79, KOOI82]) can construct a plan for the portion of the query without nested dot constructs. However, run-time planning will be required on remaining portions of commands. For example, the following query must be processed by tuple substitution for o or o1.

retrieve (o.shape.p-desc, o1.shape.p-desc) where o.shape.l-desc = o1.shape.l-desc

After substitution twice, the remaining query is:

retrieve (TEMP-1.p-desc, TEMP-2.p-desc) where TEMP-1.l-desc = TEMP-2.l-desc

The characteristics of TEMP-1 and TEMP-2 are not known until run time, so further query planning must be deferred to this time.

The only exception to run time planning would occur if all values in a procedural column contain the same query as discussed in Section 2. In this situation, a view translation algorithm can be run on the initial user command instead of applying the algorithm of this section. The algorithm is similar to the one presented in [STON75] and would translate a multiple-dot query into a conventional query which can be optimized in the conventional fashion. This ‘‘flattening’’ of a query will allow a compile time plan to be built and additionally will support a wide range of query processing alternatives to be explored, rather than just the ‘‘outside-to-inside’’ strategy discussed in this section. The details of this algorithm are straight-forward and are omitted for the sake of brevity.

Lastly, in our prototype the module that materializes a relation passes the RETRIEVE commands to another process which also runs the INGRES+ code. This second INGRES+ executes the command, stores the resulting relation in the data base, and then passes control back to the first INGRES+. A second process is required because the INGRES code will not allow a command to suspend in the middle of the decomposition process so that a new command can be executed. The ability to ‘‘stack’’ the execution state of a query would be a very desirable addition to the system.

5. BENCHMARK RESULTS

It would be clearly desirable to compare the performance of INGRES+ against various other approaches to object management. These could include using a conventional relational system as well as prototypes with other capabilities (e.g. [COPE84, LORI83]). Only a conventional relational system was easily available in our environment as a test case. Hence, a more detailed performance study is left as a future exercise and would require the acquisition of appropriate hardware to run other prototypes.

In this section we describe a collection of benchmarks which we performed on our prototype. We modeled three different tasks using QUEL+ and then compared them to a conventional relational system, namely INGRES [STON76]. In all cases we chose tasks which would result in different queries in the two systems. Running the same command in both systems would clearly result in equal performance. In all tests recovery and concurrency control has been turned off, and CPU time and total elapsed time in a single user environment were tabulated. For convenience, INGRES numbers are normalized to 1 while INGRES+ numbers are given as a multiple of the corresponding INGRES result. All tests are run on a single-user VAX 11/780.

Both systems contain substantial inefficiencies (e.g. run-time optimization, generation of an excessive number of temporary relations). However, it appears that such problems penalize both systems about equally. Only three issues excessively penalize INGRES+. First, the unnecessary communication with a second INGRES+ task adds unnecessary overhead that could be eliminated in a commercial implementation. Second, in many cases INGRES will be seen to execute a single two-variable query while INGRES+ runs a larger number of one-variable commands. Run time query planning of a larger number of commands imposes an excessive penalty on INGRES+. Lastly, the flattening of parameterized procedural fields has not yet been implemented in INGRES+. Consequently, execution of muliple-dot queries is constrained by the structure of the query, and an inefficient plan may be executed as a result. Hence, a compiled query implementation of QUEL+ which included parameterized procedual fields should yield results similar to or more favorable toward INGRES+ than those we present.

The three experiments are discussed in the following subsections.

5.1. Simulation of Simple Complex Objects

This experiment involves accessing simple variant records corresponding to hobbies in the EMP relation of Section 2. Each of 7000 employees has a collection of hobbies. From a total of 50 possible hobbies, each employee practices between one and eight.

Both an INGRES and an INGRES+ data base must store records on each of the 50 hobbies in relations:

```
SOFTBALL (emp-name, other data)
SAILING   (emp-name, other data)
JOGGING  (emp-name, other data)
 .
 .
 .
```

A normal DBMS would store in addition the relations:

```
EMP(name, age, salary)
HOBBIES(emp-name, hobby-name)
```

while an INGRES+ data base would only require a single relation:

```
EMP(name, age, salary, hobbies)
```

The field ‘‘hobbies’’ has a collection of queries, one per hobby as noted in Section 2.

The task is to find the information on all hobbies for a given employee and is expressed in INGRES+ as follows:

```
execute (EMP.hobbies) where EMP.name = ‘‘unique-emp’’
```

A normal DBMS query language cannot express this task, and the most reasonable option is to execute the following algorithm.

```
retrieve (HOBBIES.hobby-name) where HOBBIES.emp-name = ‘‘unique-emp’’
for each such hobby-name do
   retrieve (hobby-name.all) where hobby-name.emp-name = ‘‘unique-emp’’
end-do
```

Table 2 indicates a performance comparison for various numbers of hobbies per employee. The INGRES+ numbers result from running the above execute command while the INGRES number were obtained using the terminal monitor to retrieve the hobbies for a given employee and then executing the appropriate number of retrieve commands to obtain hobby data. This would simulate a user who ran a query to obtain the collection of hobbies and then ran the correct collection of queries on the various hobbies relations. Notice that the INGRES+ option is superior except when there is a single hobby per employee. This performance difference results from the fact that a large number of queries are passed through the INGRES terminal monitor, which has noticeable overhead. The INGRES+ solution runs the same collection of queries, but the hobby queries are generated internally by the system and do not go through a terminal monitor.

5.2. Simulation of More Complex Objects With Shared Subobjects

Consider the example from Section 2 where complex objects are composed of lines, text and polygons. Moreover, assume that these subobjects must be shared among various complex objects. Consequently, both schemas have relations for the subobjects as follows:

```
LINE (Lid, l-desc)
TEXT (Tid, t-desc)
POLYGON (Pid, p-desc)
```

Then, a normal schema must have three additional relations indicating which subobjects are in which complex objects:

```
T-obj  (Tid, Oid)
P-obj  (Pid, Oid)
L-obj  (Lid, Oid)
```

However, an INGRES+ schema needs only the single additional relation from Section 2, namely:

```
OBJECT (Oid, trim, shape)
```

The trim field in OBJECT is a collection of queries of the form:

```
retrieve (TEXT.all) where TEXT.Tid = value
```

while the shape field has a collection of queries of the form:

```
retrieve (LINE.all) where LINE.Lid = value
retrieve (POLYGON.all) where POLYGON.Pid = value
```

The benchmark query is to find the shapes of a particular complex object, e.g:

```
retrieve (POLYGON.all)
where POLYGON.Pid = P-obj.Pid
```

Query	INGRES-CPU	INGRES+-CPU	INGRES-total	INGRES+-total
one-hobby	1	1.23	1	1.28
four-hobby	1	.73	1	.61
eight-hobby	1	.59	1	.61

A Benchmark of Simple Complex Objects

Table 2

and P-obj.Oid = ‘‘unique-value’’

retrieve (LINE.all)
where LINE.Lid = L-obj.Lid
and L-obj.Oid = ‘‘unique-value’’

The QUEL+ query is simply:

execute (OBJECT.shapes) where OBJECT.Oid = ‘‘unique-value’’

The INGRES+ prototype limits the length of procedural fields to 255 bytes (about 9 queries); hence, multiple rows are required to express an object with a larger number of subobjects. This limitation affects INGRES+ performance marginally. The costs of the two systems for objects having respectively 1, 4, 8, 16 and 32 subobjects is shown in Table 3. INGRES must run 2 two-variable queries while INGRES+ runs a single query on the OBJECT relation followed by a one-relation query per subobject. If there is only one subobject, it is clear that INGRES+ will run 2 one-variable queries and have better performance than INGRES. This performance advantage deteriorates until there are 16 subobjects at which point 17 one-variable queries take more time than 2 two-variable queries. In a commercial implementation, two-variables queries would be better optimized and the crossover point might occur at a lower number of subobjects. On the other hand, run-time optimization of 17 commands in INGRES+ is a serious source of overhead which would not be present in a commercial system. Hence, it is not clear how Table 3 would look in a commercial environment.

5.3. Simulation of Unnormalized Relations

Although QUEL+ is most useful when applied to applications with complex structure, it is also possible to provide multiple-dot addressing on conventional data. This will allow a more natural query formulation compared to conventional techniques; however, much of the same effect can be alternately achieved using relational views. This section is included to demonstrate that QUEL+ provides reasonable performance even in ordinary situations.

The normal way to store data for the standard EMP, DEPT and JOB data base is:

EMP (name, age, salary, dept, jid)
DEPT(dname, floor)
JOB (jid, jname, benefits)

Query	INGRES-CPU	INGRES+-CPU	INGRES-total	INGRES+-total
Q1	1	.54	1	.67
Q4	1	.75	1	.78
Q8	1	.99	1	1.0
Q16	1	1.35	1	1.44
Q32	1	2.17	1	2.94

Benchmarks of Complex Objects

Table 3

Here, employees have a name, an age, a salary, are in a department, and have a job identifier. The other two relations are self-evident. We assume that there are 7000 employees, 500 departments and 50 job descriptions. Moreover, EMP tuples are 32 bytes wide, DEPT tuples are 14, and JOB tuples are 24.

On the other hand, in INGRES+ one can use an alternate schema as follows:

EMP (name, age, salary, dept, j-emp)
DEPT(dname, floor, d-emp)
JOB (jid, jname, benefits)

Here, j-emp is a procedural field of the form:

retrieve (JOB.all) where JOB.jid = ‘‘value-for-this-employee’’

In addition, d-emp is a procedural field of the form:

retrieve (EMP.all) where EMP.dept = ‘‘this-dept’’

Consequently, all the employees in a specific department are accessible though the d-emp field while the job description of a particular employee can be obtained through the j-emp field.

We ran the following three queries in both INGRES and INGRES+

Query 1: a normal join returning a few tuples

The queries to run in the two systems are respectively:

retrieve (EMP.name, DEPT.floor)
where EMP.dept = DEPT.dname
and DEPT.dname = ‘‘unique-name’’

retrieve (DEPT.d-emp.name, DEPT.floor)
where DEPT.dname = ‘‘unique-name’’

In this case, we are comparing the processing speed of normal INGRES running a two variable query with that of INGRES+ which must execute a one-variable query and then a second one-variable subquery.

Query 2: the full join

The two queries are respectively:

retrieve (EMP.name, DEPT.floor)
where EMP.dept = DEPT.dname

retrieve (DEPT.d-emp.name, DEPT.floor)

In this case, we are computing the full join between EMP and DEPT. The comparison is between a single two variable query and a single one-variable query to scan the DEPT relation along with 500 one-variable queries to find appropriate information in EMP. This should be a poor query for INGRES+ because of the run-time optimization of 500 queries. Moreover, because of the structure of the query, INGRES+ will iterate over DEPT tuples and then access the EMP relation for each one. This may (or may not) correspond to the plan which would be selected by a conventional optimizer using a flat representation of the query. If iterative substitution for DEPT tuples is not a wise plan, then INGRES+ will have poor performance because of the structure of the query.

Query 3: a three way join to find the job of a particular employee in a particular department

The queries are:

```
retrieve (JOB.jname, EMP.name)
where DEPT.dname = ‘‘value-1’’
and EMP.dept = dept.dname
and EMP.job  = JOB.jid
and EMP.name = ‘‘value-2’’

retrieve (DEPT.d-emp.j-emp.jname, DEPT.d-emp.name)
where DEPT.dname = ‘‘value-1’’
and DEPT.d-emp.name = ‘‘value-2’’
```

Here, INGRES is running a single three variable query while INGRES+ will execute three one-variable queries. The extended system should be especially attractive in this case, because of the extra complexity required to process multivariable queries in a conventional system.

Table 4 presents the results for these three queries.

Query	INGRES-CPU	INGRES+-CPU	INGRES-total	INGRES+-total
Query 1	1	1.37	1	1.15
Query 2	1	15.1	1	17.2
Query 3	1	.29	1	.36

Benchmarks of Unnormalized Relations

Table 4

As can be seen, Query 1 performs at about the same speed in both systems. In this case two one-variable queries are comparable to a single two variable query. The full join was a factor 15-17 worse in INGRES+ because the overhead of running 500 queries to retrieve EMP tuples is overwhelming. Finally, Query 3 shows that three one-variable commands are faster by a factor of 3-4 than a single three-variable command. Although one would expect superior performance from INGRES+, the magnitude is surprising and reflects the fact that the normal INGRES optimizer is not especially good at three way joins.

The conclusion to be drawn is that INGRES+ is competitive except when it utilizes a poor query plan or is forced to run a large number of commands. Bad performance in the latter situation should be eliminated by compile time query planning. Bad performance in the former case can be alleviated by flattening out multiple-dot commands when an entire column has the same query structure.

The next section turns to a suggestion to dramatically improve the performance of procedural fields.

6. CACHING PROCEDURAL FIELDS

The performance of INGRES+ may be dramatically improved by caching frequently used objects so they will not have to be repeatedly rematerialized. This section explores the use of this tactic.

6.1. The Cache Model

Caching QUEL procedures should be thought of as a two step process:

1) compile a query plan for the command(s) (plan caching)
2) execute the plan (result caching)

The first step (plan caching) is often done at compile time in current systems; however, in our model it should be thought of as computing an intermediate representation of the object. This representation can be saved for later reuse. Moreover, in current systems a query plan is usually invalidated at execution time if the schema has changed in a way that compromises the validity of the plan. In our model query plans are cached until a compromising update forces invalidation. One can optionally support a ‘‘demon’’ which utilizes any idle CPU resources recompiling invalidated plans. Alternatively, one can simply recompile when the plan is executed, as in [ASTR76].

The size of a compiled plan depends on the target language of the compiler. If access plans are generated, then the size will be modest (e.g. hundreds of bytes). If machine code is generated, then the size will be thousands of bytes. If plans are of moderate size, they can be cached directly in the field that defines them. Larger plan representations can be cached in a separate relation, i.e:

CACHE (identifier, compiled-plan)

and only the identifier stored in the defining field.

The second step (result caching) involves materializing the object from the compiled plan. Again this step can be performed on demand and saved for reuse or even computed in advance. The size of a materialized object depends, of course, on the command(s) which is executed. Small objects can be cached directly in the defining field. Larger objects should probably be cached as individual relations, and the name of the relation(s) inserted in the defining field. When objects are hierarchically composed of other objects, the above constructs can be applied recursively; hence, small objects which are composed of small objects will be cached together in the field describing the enclosing object.

It is straightforward for the data base system to allow result caching for N1 ‘‘big objects’’ and execute a least recently used (LRU) algorithm to select big objects to be discarded. Of course this requires N1 relations to hold these objects and the corresponding extra entries in the system catalogs. Alternately, a data manager could reserve N2 blocks of storage for objects and select a victim based on a function of size and time-since-last-reference. Of course, N1 and N2 would be carefully chosen system-specific parameters. Objects must also be discarded upon an update to one or more tuples from which they are composed. We discuss a mechanism to accomplish this task presently. Alternately, it should be possible to incrementally update the cached object when a subobject is modified. Recent work on supporting materialized views (e.g. [BLAK86]) can be applied to this task. Moreover, if the object is an QUEL aggregate, it is straightforward to update the cached value. Additional effort in this direction is currently in progress.

For cached big objects, it is desirable to store their name and the query(s) which compose them in a main memory data structure. Then, if the same or another user materializes an object with the same description, the one already materialized can be used instead. An example of this situation occurred in Section 4 where our algorithm materialized the object represented by o.shape twice.

The prototype discussed in Section 4 caches big objects as discussed above, but small object caching as well as invalidation on update has yet to be implemented. We turn now to the efficient invalidation of objects.

6.2. Object Invalidation

Consider a new kind of lock mode called I mode. Hence, objects can be locked in R, W or I mode. A lock set in I mode has an associated identifier indicating the object which has been precomputed using the locked object. The compatibility of the various modes is indicated in Table 5. The * in that table indicates that a W requestor for an object locked in I mode will be allowed to proceed and set a W lock on the object. First, however the object with which the I lock is associated will be invalidated.

When INGRES+ materializes any object, it simply sets I locks on all objects read by the query(s) which materialize the object. These I locks are held until the object is deleted or invalidated through an update to a subobject.

6.3. Implementation of I Locks

A straight-forward approach would be to place I locks in the same lock table holding R and W locks. In this case, one must cope with a lock table of widely varying size since the number of I locks can change dramatically. Moreover, when a failure occurs, either all precomputed objects must be invalidated (which may be a costly alternative if the facility is extensively used) or I locks must be made recoverable. Lastly, phantoms must be correctly handled. Hence, if a new tuple is added which satisfies the qualification of some precomputed object, then this object must be invalidated.

The first objective can be satisfied by using extendible hashing [FAGI79] for the lock table instead of conventional hashing. The second objective requires setting I locks as part of a transaction and writing them into the log. If a failure occurs during this transaction, then recovery code must be extended slightly to back out the I locks which were set. Moreover, I locks must be periodically checkpointed to allow recovery from media failures. Hence, when recovery code is rolling forward from a checkpoint, I locks can be suitable updated. In summary, making I locks recoverable simply involves treating them as ordinary data and presents only modest implementation difficulties.

The phantom problem poses more serious issues. Systems which perform page level locking (e.g. [RTI86, CHEN84]) have few difficulties supporting correct semantics in the presence of phantoms. Hence, one can include I locks in such systems without concern. However, finer granularity locking is required to avoid an excessive number of unnecessary invalidations. Systems which perform record level locking (e.g. System R [ASTR76]) can allow detection of phantoms by holding locks on index intervals in the leaf nodes of secondary indexes as well as on data

	R	W	I
R	ok	no	ok
W	no	no	*
I	ok	no	ok

Compatibility Modes for I Locks

Table 5

records. Hence, a transaction which inserts a tuple will hold a write lock on the tuple and on the appropriate index interval for any field for which a secondary index exists. If I locks are also held on data records and index intervals, then all conflicts caused by phantoms will be detected by a collision in some index.

When access paths other than B-trees are present, a slight generalization to the above scheme will support phantom detection. Tuple level locks are held on index and data records which use a hashed or B-tree organization. Then, a precomputed object must be invalidated if an I lock which is held on its behalf falls adjacent to a tuple on which a W lock is held. For B-tree data records and indexes, adjacency means ‘‘logically adjacent in tuple identifier order’’; for hashed records and indexes, adjacency means ‘‘in the same hash bucket.’’

The only problem with the adjacency approach is that a B-tree page split will cause a write lock to be set on the page to be split, and thereby will cause an invalidation of all objects holding I locks on that page. Unless tuple identifiers are constructed so that they do not change when a tuple moves to a different page, this invalidation is required. Otherwise I locks will be held on the previous identifier for the moved tuple, and incorrect operation will result.

The phantom problem and the logging problem appear easier to solve if an alternate strategy is employed. Consider storing I locks in the data and index records themselves. Systems which support variable length records can simply add as many I locks to each record as necessary. Such locks are recoverable using conventional techniques which are automatically applied to data records. The phantom problem requires the above adjacency algorithm; however, structure modifications (e.g. B-tree page splits) do not cause unnecessary invalidations. Moreover, since the extra locks are stored separately from R and W locks, extendible hashing is not a prerequisite for the lock table. Lastly, this approach is easily extended to field-level locks, which may offer superior performance to record level I locks.

The drawbacks of this second alternative is that a second implementation of a lock manager must be coded for I locks. Moreover, setting an I lock on an object requires rewriting the object instead of just reading it. Hence, setting I locks will be expensive.

6.4. Performance of Cached Objects

The purpose of this section is to present a model which can be used to suggest when caching should be applied to a procedural object. Consider a ‘‘small’’ object which can be constructed in N1 page accesses and inspection of N2 records. Assume that this object occupies R1 = 1 page of space, consists of R2 tuples and is cached in the data record itself. In addition, consider a query pattern in which P percent of the accesses are reads of the complex object and 1-P are updates to subobjects from which the complex object is composed. Moreover, each update is applied to a randomly chosen subobject from the N2 candidates. Lastly, the record in which the description of the complex object is stored must be read during retrieval whether or not caching is employed. Hence, that access is not counted in the following analysis.

Consider the sequence of accesses to be a collection of **intervals,** each consisting of one or more consecutive writes followed by one or more consecutive reads. In a given interval, the expected length ER of the run of reads and the length EW of the run of writes is:

$$ER = 1 / (1 - P)$$
$$EW = 1 / P$$

Consider first the no-cache alternative. With the parameter K discussed in [SELI79] which weighs the CPU time used in evaluating a query plan relative to the number of I/O’s, the cost, M to materialize the object after its definition has been obtained is:

$$M = N1 + N2 * K$$

Without caching, this cost must be paid on each read access, and the cost, C1 of these accesses is:

C1 = (ER) * (M)

We now turn to the corresponding costs for cached objects. The cost to invalidate the complex object on the initial write is:

IN = 1

This assumes that I locks are stored in the records of the subobjects and that only the data record containing the object description must be rewritten to perform invalidation. All I locks are simply left in place. Subsequent writes prior to the first read also require the same cost even though the complex object has already been invalidated.

The first read to the complex object requires a materialization at cost, M. In addition, the cost, C to cache the object is:

C = 1

The materialized object must be written into the field occupied by the description of the complex object requiring a single record to be rewritten. Since I locks were never reset from prior materializations, no extra cost is required to ensure that they are set. Subsequent reads only require accessing the object in the cache. Since the access to the object description is not being counted, there is no additional I/O because the cached object is stored in this record. Hence, only the CPU cost to access the R2 records in the cached object must be paid, i.e:

A = R2 * K

Therefore, the expected cost of an interval using caching is:

```
C2 = EW * IN              /*cache invalidation on each write*/

   + M + C                /*materialize plus cache on first read*/

   + (ER - 1) * (A)       /*subsequent reads access the cache*/
```

Define Z = M - A to be the caching factor, i.e. the difference between the cost to materialize the object and the cost to access the object from the cache. This cost is in weighted CPU and page costs and is typically in the range of 10 - 1000. Algebraic manipulation now yields that caching will be preferred if:

Z > (1 / P ** 2) - 1

The consequence of this analysis is that caching will generally win unless P is very small. If P is 1/2, then Z must be greater than 3 for caching to be beneficial. If P is 1/10, then Z must exceed 99.

6.5. Indexing Cached Fields

An extension of this caching tactic may be attractive when the queries in a field have certain compositions. Consider a situation, such as salaries in an EMP relation for which most of the fields are specified as constants while a few are specified procedurally. For example, Joe might make $10,000 and Bill is specified as having the same wages as Joe. In this case salary can be a procedural field and most rows have a value of the form:

retrieve (wages = constant)

while Bill would have a value of:

retrieve (EMP.salary.wages) where EMP.name = ‘‘Joe’’

In this case, one would desire a salary index on salaries for efficient processing of queries of the form:

retrieve (EMP.name) where EMP.salary.wages = value

This section indicates how to construct such indexes on procedural fields.

Instead of caching values as they are computed, consider caching all values in advance. If this is done, then it is straightforward to build an index on a field appearing in the cached objects using the conventional indexing utility. One can use this index to answer queries of the above form rapidly. On updates to subobjects, one must invalidate cached objects as discussed in the previous subsection. However, as part of the transaction which updates a subobject, the invalidated object must be rebuilt and the index updated. Conventional locking must be employed to guarantee consistency, and aborting a transaction must cause a backout of all updates or an invalidation of the new cached value.

In the case that most values are simply constants, such indexes should be beneficial, and be only marginally more expensive to maintain than conventional ones.

7. CONCLUSIONS

This paper has suggested that data base procedures are a natural way to model complex objects and to allow data base oriented algorithms and precompiled queries in the data base. Moreover, they appear to be easily generalizable to arbitrary programming language procedures which may be useful in certain applications. Lastly, they can be used to model aggregation, generalization and most other environments addressed by semantic data models. The advantage of data base procedures is that a user does not need to learn additional concepts to design his application. Since he must know the query language anyway, there is little extra complexity. Hence, this proposal is in the same spirit of the ‘‘spartan simplicity’’ stressed by the original advocates of the relational model.

A prototype implementation was described and initial performance studies explored. They indicated comparable performance between the extended and the non-extended prototypes except when many one-variable commands were processed by INGRES+ or when a bad query plan was indicated by the structure of the query. Bad plans can be avoided if procedural fields are restricted to parameterized queries, and compile time optimization should alleviate the overhead of one-relation commands. Moreover, a caching strategy was described which should accelerate performance of the prototype especially when objects are of moderate size. Our caching scheme should offer superior performance to pointer oriented implementations of complex objects when update frequency is low or moderate.

REFERENCES

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

[BLAK86] Blakeley, J. et. al., ‘‘Efficiently Updating Materialized Views,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[CHEN84] Cheng, J., et. al., ‘‘IBM Database 2 Performance: Design, Implementation, and Tuning,’’ IBM Systems Journal, February 1984.

[COPE84] Copeland, G. and Maier, D., ‘‘Making Smalltalk a Data Base System,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

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

[DBTG71] Data Base Task Group, ‘‘Report to the CODASYL Programming Language Committee,’’ April 1971.

[EPST80] Epstein, R., and Hawthorn, P., ‘‘Design Decisions for the Intelligent Database Machine,’’ Proc. 1980 National Computer Conference, Anaheim, Ca., May 1980.

[FAGI79] Fagin, R. et. al., ‘‘Extendible Hashing: A Fast Access Method for Dynamic Files,’’ ACM-TODS, Sept. 1979.

[HAMM81] Hammer, M. and McLeod, D., ‘‘Database Description with SDM,’’ ACM-TODS, September 1981.

[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.

[HELD75] Held, G. et. al., ‘‘INGRES - A Relational Data Base System,’’ Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

[KALA85] Kalash, J., ‘‘Implementation of a Data Base Browser,’’ Electronics Research Laboratory, University of California, Berkeley, Ca., Memo No. M85/22, May 1985.

[KOOI82] Kooi, R. and Frankfurth, D., ‘‘Query Optimization in INGRES,’’ Database Engineering, Sept. 1982.

[KUNG84] Kung, R. et. al., ‘‘Heuristic Search in Database Systems,’’ Proc. 1st International Conference on Expert Systems, Kiowah, S.C., Oct. 1984.

[LORI83] Lorie, R. and Plouffe, W., ‘‘Complex Objects and Their Use in Design Transactions,’’ Proc. Engineering Design Applications Stream of ACM-IEEE Data Base Week, San Jose, Ca., May 1983.

[MYLO80] Myloupoulis, J. et. al., ‘‘A Language Facility for Designing Database Intensive Applications,’’ ACM-TODS, June 1980.

[ONG84] Ong, J., et. al., ‘‘Implementation of Data Abstraction in the Relational Database System INGRES,’’ SIGMOD Record, March 1984.

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

[ROWE82] Rowe, L. and Shoens, K., ‘‘A Form Application Development System,’’ Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl, June 1982.

[RTI86] Relational Technology, Inc., ‘‘INGRES Version 5.0 Reference Manual,’’ November 1986.

[SELI79] Selinger, P., ‘‘Access Path Selection in a Relational Database System,’’ Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[SHIP81] Shipman, D., ‘‘The Functional Model and the Data Language Daplex,’’ ACM-TODS, March, 1981.

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

[SORD84] Sordi, J., ‘‘IBM Database 2: The Query Management Facility,’’ IBM Systems Journal, February 1984.

[STON75] Stonebraker, M., ‘‘Implementation of Views and Integrity Control 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,’’ ACM-TODS, September 1976.

[STON84] Stonebraker, M. et. al., ‘‘QUEL as a Data Type,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[STON86] Stonebraker, M., ‘‘Object Management in POSTGRES Using Procedures,’’ Electronics Research Laboratory, University of California, Memo M86/42, July 1986.

[WILE84] Wilensky, R., ‘‘The LISP PRIMER’’ W. Norton, Co, New York, 1984.

[WONG76] Wong, E., ‘‘Decomposition: A Strategy for Query Processing,’’ ACM-TODS, Sept. 1976.

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

APPENDIX 1

The syntax of QUEL has been described elsewhere [HELD75]. Here, we report only the extensions to the original language, QUEL, which define QUEL+. Let F be the construct ‘‘QUEL-col-1.,...,QUEL-col-n.field’’.

1) F can appear wherever a field of a relation can appear in QUEL.

2) The construct ‘‘tuple-variable.F can appear whenever a tuple variable or a relation name can appear in QUEL.

3) Clauses of the form:

G1 newop G2

are allowable if G1 and G2 are tuple variables or the construct ‘‘tuple-variable.F’’ and newop is in the set:

{ U , !! , >> , << , == , <> , JJ , OJ, empty }

4) EXECUTE and EXECUTE-ONE are added as commands

5) An operator ‘‘in’’ is added accepting an indirectly referenced column as a left operand and a relation name as a right operand.

6) A keyword ‘‘with’’ is added which is usable with the EXECUTE command to indicate the presence of a parameter list.