OBJECT MANAGEMENT IN POSTGRES USING PROCEDURES

Michael Stonebraker

Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This paper presents the object management facilities being designed into a next-generation data manager, POSTGRES. This system is unique in that it does not invent a new data model for support of objects but chooses instead to extend the relational model with a powerful abstract data typing capability and procedures as full-fledged data base objects. The reasons to remain with the relational model are indicated in this paper along with the POSTGRES relational extensions.

1. INTRODUCTION

This paper presents the mechanisms in POSTGRES [STON86a] to support object management. This system does not invent a new data model for manipulation of complex objects, but rather extends the relational model with a powerful abstract data typing system and support for procedures as a fundamental data type. With these constructs, most application specific data models can be easily simulated. A companion paper illustrates this fact by showing how a shared, multiple-inheritance, object hierarchy can be implemented on POSTGRES [ROWE86]. Hence, POSTGRES appears to easily support a wide variety of application specific needs without compromising the simplicity of the relational model for conventional business data processing applications.

In Section 2 we briefly review a collection of data modeling proposals intended for support of non-traditional applications. We also argue that there is no small common collection of ideas on which to base the data model of a general purpose next-generation data base system. Consequently, the thrust of next-generation systems should be to efficiently simulate a variety of application specific data models.

In Section 3 and 4 we discuss the specific approach taken in POSTGRES which utilizes an abstract data typing capability and procedures as full-fledged data base objects. Section 5 closes with a summary of the capabilities of our approach.

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0021 and by the Naval Electronics Systems Command Contract N00039-84-C-0039.

2. THE CASE FOR THE RELATIONAL MODEL

This section briefly discusses three reasons to retain the relational model as the backbone of a next-generation system.

2.1. The Semantic Poverty Argument

It is often argued (e.g. [ZANI83]) that the relational model is semantically impoverished, and should be replaced by a data model with additional semantic constructs. Over the last ten years there has been considerable research toward identifying such a model, and in this section we briefly list some of the constructs proposed.

Without attempting to be very rigorous at classification or exhaustive in coverage of proposals, the following list is easily assembled from the literature.

ENRICHED COLLECTION OF OBJECTS

entities, attributes and relationships [CHEN76]
classes [HAMM81]
roles [BACH77]
objects with no fixed type or composition [COPE84]
set valued attributes (repeating groups) [ZANI83]
unnormalized relations [LUM85]
class variables (aggregation) [SMIT77]
category attributes and summary tables [OZSO85]
molecular objects [BATO85]

TYPES OF RELATIONSHIPS

"is-a" hierarchies [SMIT77, GOLD83]
"part-of" hierarchies [KATZ85]
convoys [CODD79, HAMM81]
associations [WONG80]
referential integrity (inclusion dependencies) [DATE81]
grouping connections [HAMM81]
equivalence relationships [KATZ86]

OTHER CONSTRUCTS

ordered relations [STON83a]
long fields [LORI83]
hierarchical objects [LORI83]
multiple kinds of nulls [KENT83]
multiple kinds of time [SNOD85]
versions [WOOD83]
parameterized versions [BATO85]
snapshots [ADIB80]
synonyms [LOHM83]
table names as a data value [LOHM83]

automatic sampling [ROWE83]
recursion or at least transitive closure [ULLM85]
windows or universal relations [KORT84]
semantic attributes [SPOO84]
unique identifiers [CODD79, POWE83]
demons [STON85a]

Two conclusions are evident:

1) There is a large collection of constructs, each relevant to one or more application specific environment.

2) The union of these constructs is impossibly complicated to understand and probably infeasible to implement with finite resources.

Hence, it appears inappropriate to look for a single universal data model which will support all non-traditional applications. In short, what the CAD community wants is different from what the semantic modeling community wants which is different from what the expert data base community wants, etc. Consequently, such users should build application specific data models containing the constructs needed in their environment.

The thrust of a next-generation data base system should be to provide a support system that will efficiently simulate these constructs. The next section discusses the POSTGRES capabilities which will be seen to be considerably more powerful than other proposals with the same general intent.

Since the relational model has found such widespread acceptance, it should be the task of the proponents of some other data model to demonstrate that their choice provides the same degree of simplicity and simulation power as provided within the relational context by POSTGRES.

2.2. The Simplicity Argument

There are many drawbacks to using a complex tool rather than a simple one in a data base environment. First, the user manual is longer and harder to write, and training customers to use the tool is more costly. Second, a more complex tool has inherently higher technical support costs than one which is simpler. Additionally, there is often more than one construct that can be used to model a particular real world situation. Hence, advice is needed on which one to use and the performance implications of the choice. A logical and physical design tool is thereby harder to construct.

All other data models are more complex than the relational model. Clearly, constructs should be included in a data model only if the power provided overwhelms the cost of the added complexity in a variety of application environments. In my opinion, this power/complexity case has not been persuasively made by the advocates of most of the specific constructs in the previous section.

Stated differently, this author considers simplicity a good idea. The remarks of [CODD70] on the subject seem as valid now as they did when written fifteen years ago.

2.3. Compatibility

It is conceded by most that the relational model provides a good fit to the needs of the business data processing community. Such data will clearly gravitate from older technology data managers into relational data bases over the next decade.

It is also obvious that users will demand the ability to correlate data in multiple data bases managed by multiple software packages. For example, consider a CAD data base containing the design of a particular printed circuit board. This PC board contains packages which are bought from outside suppliers. Hence, it is certainly appropriate to ask the total cost of packages contained in the PC board. This query requires the ability to correlate data in the CAD data base with data on suppliers and parts. This latter data base is business data processing data and will presumably be in a relational system. As a result, one will need to correlate a relational data base with whatever data base system manages the CAD data.

This problem was addressed by Multibase. Moreover, it is obvious that problems of heterogeneity become increasingly severe the further one strays from the relational model. Hence, compatibility issues are an additional reason to retain the relational model unless an overwhelming case can be made to displace it.

3. THE POSTGRES ADT SYSTEM

POSTGRES supports object management within the relational model with two facilities, an abstract data type (ADT) facility and procedures as a data type. The ADT system has been described in [STON83b, STON86b], and is briefly reviewed in this section. POSTGRES support for procedures is considered in detail in the next section.

POSTGRES allows a user to implement a new data type which can then be used as the type of any column in any relation in the data base. Moreover, operators specific to the new data type can be included in the query language by writing a procedure to evaluate the operator. This capability is useful for all kinds of objects normally found in engineering applications (e.g. boxes, lines, polygons, points, line-groups, complex numbers, vectors, bitmaps, etc). For example, the proposal of [STON83b] discusses the inclusion of box as a data type along with a collection of operators (e.g. intersection, area-of, to-the-left-of, etc.) appropriate to the new type. The facility is also useful in business data processing applications. For example, many commercial system implement date and time as a data type (e.g. INGRES, FOCUS, NOMAD) along with operators on this type (e.g. subtraction). Unfortunately, the normal definition of subtraction for dates is not appropriate for some segments of the financial community which utilize a 360 day year and 12 equal length months. Only an ADT system allows a user community to implement a different definition of subtraction.

In summary the collection of data types and operators provided by most current data base systems are appropriate to the needs of business data processing applications. One need only allow an extensible type system to support the needs of others.

This ADT proposal is extended in [STON86b] with constructs that allow a heuristic query processor to optimize query language expressions containing new operators and new data types. Preliminary discussion of support for new access methods was also included. In the interest of brevity, these proposals are not summarized as they are not relevant to the following discussion.

An ADT facility meets the needs of a variety of object management applications. However, it fails in three important situations:

objects with many levels of subobjects

objects with unpredictable composition
objects with shared subobjects

Consider a mechanical CAD application which stores a particular building in a data base. An object in such a data base might be an office desk. However, the desk is in turn constructed of subobjects (e.g. drawers), which are in turn constructed of subobjects (e.g. handles). This "part-of" hierarchy is prevalent in many engineering applications. A user often wishes to "open up" an object and access specific subobjects. For example, he might want to find the handle on the lower left-hand drawer. The ADT proposal noted above would force a user to write an operator for each such access he wanted to perform. A very large number of operators would result that would be exceedingly hard to use. In summary, a user wants the query language to assist with "opening up" complex objects and searching for qualifying subobjects; he does not want an operator for each particular search.

The second problem concerns unpredictable composition of objects. This issue is noted in [COPE84], and can be easily illustrated with the desk data. Suppose the data base contains objects that are on top of the desks in the example building. In particular, some desks have flowers, some have simple phones, some have switchboard phones, etc. In this case, a subobject of a desk may be one or more objects from a huge set of possible desk accessories. It is unreasonable to require a user to write an operator to extract any object from such an unpredictable collection.

The third problem concerns shared subobjects. Consider a heating duct in the building that is accessible from several rooms in the building. One would want to store the duct once, and then have it be a shared subobject in higher level objects (rooms). The ADT proposal noted above has no ability to share subobjects in this fashion.

To support objects with any of these requirements, POSTGRES supports procedures as full-fledged data base objects. In the next section we indicate the specific procedural support that we are constructing.

4. POSTGRES PROCEDURES

POSTGRES supports the notion of a registered procedure which can be used in query language commands as well as two different procedural data types, namely:

POSTQUEL procedure
parameterized POSTQUEL procedure

These are discussed in turn below.

4.1. Registration of Procedures

A procedure in a general purpose programming language can be **registered** to POSTGRES by indicating the following information

the name of the procedure
the implementor of the procedure
the data types of its parameters
the data type of its result
the programming language it is written in
the source language representation of the code for the procedure
a type-checking flag
a precomputation flag

Registration of a procedure is a POSTQUEL utility command which fills the above information

into two system relations, one for the procedure information and one for the parameters. After registration, the procedure is compiled asynchronously by POSTGRES and can be used in the POSTQUEL query language anywhere that a function is currently allowed in QUEL.

For example the code for "is-overpaid" could be registered as taking a float and an integer as arguments and returning a boolean. With this definition the following query can be expressed for the standard EMP relation:

retrieve (EMP.all) where EMP.age > 35
and is-overpaid (EMP.salary, EMP.age)

A second example would be a "progress" procedure which accepts a float and an integer and returns an integer between 1 and 10. The employees whose progress is greater than 4 who are over 35 would be expressed as follows:

retrieve (EMP.all) where EMP.age > 35
and progress (EMP.salary, EMP.age) > 4

This mechanism is a straighforward extension of hard-wired functions currently supported in QUEL (e.g. sin, cos, log, sqrt, etc.).

Registered procedures have the types of their arguments installed in a system relation. Consequently, type checking is done on the arguments to any registered procedure. If a type mismatch is discovered, then argument conversion takes place. This conversion is guaranteed to succeed, because part of registering a data type to POSTGRES is specifying two operators which will convert ascii to the new type and then back. Hence, if T is the type of argument expected and Y is the type of the actual argument, then POSTGRES need only apply the Y-to-ascii function followed by the ascii-to-T function.

In order to avoid a double conversion, we may experiment with a special class of functions called **conversion functions,** which convert between data types. If there exists such a registered function which has Y as the type of its argument and produces T as the type of its result, then that function can be used in place of the two functions noted above.

Note that the implementor of a registered procedure can turn type checking off by specifying the type checking flag as "no checking". This setting is appropriate in two situations. First, commands may come from an application program which can (somehow) guarantee that the arguments are the correct type. In this situation, run-time type checking of the parameters by POSTGRES generates needless overhead, and should be turned off. The second situation would be a user defined procedure which expected a variety of argument types and contained code to do its own type checking and coercion. In this case POSTGRES type checking should also be disabled.

The other flag that can be set by the implementor of a procedure declares it to be **precomputable.** In this case, POSTGRES is allowed to evaluate the procedure before receiving a request from a user. This precomputation is a central optimization for POSTGRES and is useful in a variety of circumstances as will be presently seen. In the present context, procedures with no arguments are sometimes precomputable. For example, consider the following functions:

user()
time()
group()
command()
machine-type()
factorial-10()

These functions return the current user, current time, the group of the current user (if defined), the

command he is currently running, the type of machine on which he is running, and the factorial of 10 respectively. Notice that the last two functions can be precomputed and the result of the procedure cached, while the others will generate the incorrect result if precomputed.

In the system relation containing registered procedures there is one additional flag besides those settable by the implementor. This flag declares a procedure to be **safe.** In this case, POSTGRES will call the compiled version of this procedure by linking the code into the POSTGRES address space and performing a local procedure call. This call is unprotected, and an errant or malicious procedure can bring down POSTGRES (by zeroing the disk or doing a wild branch into POSTGRES code). However, no performance penalty need be paid to call such procedures. On the other hand, unsafe procedures are called by spawning another process, loading the procedure into the created process and performing a remote procedure call. This protected version will incur considerably more overhead.

All registered procedures are initially unsafe and can be debugged without fear of crashing POSTGRES. The POSTGRES super-user (the person with the POSTGRES password) can update the safety flag to make a procedure trusted. Presumably, he does this only after inspecting the code or talking with the implementor of the procedure.

4.2. Procedural Data Types

4.2.1. POSTQUEL Fields

A column of a relation may be declared to be of type POSTQUEL procedure, e.g.:

```
create EMP (name = c10,
            age = i4,
            hobbies = POSTQUEL)
```

Each ADT has an associated external to internal conversion routine, and the one for POSTQUEL procedures will accept a quoted string containing the POSTQUEL code. With a registered procedure, file, which accepts the name of a file and returns the contents, we can express the following append command:

```
append to EMP (name = "Mike",
               age = 10,
               hobbies = file("/usr/myfile"))
```

The code in "/usr/myfile" is a collection of retrieve commands which access appropriate relations in the data base to get hobby tuples for Mike. An example collection of commands might be:

```
retrieve (windsurf.all) where windsurf.name = "Mike"
retrieve (softball.all) where softball.name = "Mike"
```

POSTQUEL procedures are automatically (and asynchronously) compiled and the answer is optionally precomputed and cached if the procedure is a retrieval. The cache is invalidated, if necessary, using the mechanisms in [STON86a]. Moreover, the "nested dot" notation can be used to address into the objects which are represented by POSTQUEL procedures as suggested in [STON84]. The following POSTQUEL command finds the batting average of Mike on the softball team.

```
retrieve (EMP.hobbies.batt-avg)
where EMP.name = "Mike"
```

Notice that any procedural object can access tuples which in turn contain procedures, so an object hierarchy can be constructed. Objects can be shared by being referenced in multiple procedural

fields. Next, the contents of a POSTQUEL field can be any query, so unpredictable composition of objects is readily supported. Finally, the nested dot notation allows the query language to be used to search inside of complex objects. Consequently, all objections to the the ADT paradigm can be overcome with POSTQUEL procedures.

Moreover, one can easily perform operations that are difficult with explicit data hierarchies, such as the ones in [HAMM81, SHIP81]. For example, the following POSTQUEL query will find all hobby data for Mike:

execute (EMP.hobbies) where EMP.name = "Mike"

To use a semantic data model, one can declare employees to be an object type and then declare a large collection of subtypes (e.g. softball-emp, windsurf-emp, etc). In order to find all the hobby information for Mike, one would have to iterate over all possible subtypes at great expense to answer the above query. Hence, POSTQUEL procedures can effectively simulate object hierarchies and also perform certain operations that are difficult with other approaches.

The remaining subsection suggests a variation of procedural types that is useful in a variety of circumstances.

4.2.2. Parameterized POSTQUEL Fields

In many instances one requires a column of a relation to be of type POSTQUEL procedure. However, all values for the column use the same procedure, differing only by the parameters used as arguments in the call. For example, suppose a second DEPT relation is added to the data base and a field "dept" is added to the EMP relation. The value of "dept" for each EMP tuple is the query:

retrieve (DEPT.all) where DEPT.dname = $1

The "$1" is simply a parameter to the query which changes from employee to employee and indicates his department. It is certainly possible to store the same query as the value for "dept" for each tuple in the EMP relation. However, space will be economized and integrity of the column will be enhanced if the procedure is "factored out" of the column and stored elsewhere.

More exactly, if the above procedure is registered using the mechanism of the previous subsection, then the EMP relation can be specified by:

```
create EMP (name = c10,
            age = i4,
            dept = POSTQUEL[pname])
```

"Pname" is simply the registered name of the above POSTQUEL procedure. With EMP so defined, a new employee can be added to the data base by:

```
append to EMP (name = "Mike",
               age = 10,
               dept = "shoe")
```

The value specified by the user for the "dept" field is the parameter to the procedure. POSTGRES converts "shoe" to the correct type and stores the parameter in the actual field. Of course, registered procedures must be extended modestly to allow POSTQUEL commands with run-time parameters to support the above capability.

There are several advantages to parameterized POSTQUEL fields, as noted in [STON85b]. First, the user can specify queries with a nested dot notation rather than using a join. For example the query

retrieve (EMP.dept.floor) where EMP.name = “Mike”

finds the floor on which Mike works. Moreover, one obtains a particular kind of referential integrity by using a procedure because all employees who belong to a non-existent department have a query which returns nothing and thereby automatically have a null department. Lastly, the query optimizer can coalesce the user command with the definition of the procedure to "flatten out" the user command and then optimize the resulting composite query. Hence, one is not restricted to processing nested-dot commands in a particular order. The flattening algorithm is discussed in [STON85b].

Parameterized POSTQUEL fields and registered procedures bear some resemblance to Smalltalk methods. In Smalltalk, there are a collection of methods (procedures) defined for an object which are stored external to the object instances. In parameterized POSTQUEL, there is exactly one method associated with an object which is separately stored. A registered procedure is similar to a Smalltalk method; however, our registered procedures are “global” to the data base rather than bound to a specific object and inherited by other objects as in Smalltalk.

In the remainder of this section we indicate one further generalization of parameterized POSTQUEL procedures. Consider the case of procedures that cannot be expressed solely in POSTQUEL. This may result from the necessity to perform computations that are not expressible easily in POSTQUEL or to format output data in some peculiar way. A good example is the "progress" of employees noted earlier. This computation might be quite involved and perhaps require accessing other relations in the data base. In order to support precomputing of the value for “progress,” one one would like to define a field in EMP that was associated with a procedure written in a general purpose programming language.

The solution is to register a procedure in the data base for “progress” and then specify a second POSTQUEL procedure:

```
retrieve (result = progress(EMP.salary, EMP.age))
where EMP.name = $1
```

Then, the user can create the EMP relation as:

```
create EMP (name = c10, age = i4, progress = POSTQUEL[pname])
```

“Name” corresponds to the above registered POSTQUEL procedure. Hence, one can insert a new employee by:

```
append to EMP (name = “Mike”, age = 10, progress = “Mike”)
```

Clearly, it is undesirable to require the constant “Mike” to be specified twice in the append command. The following generalization of registered POSTQUEL procedures allows a more compact notation.

Suppose the parameters to a POSTQUEL command can be denoted “$i” to indicate the i-th parameter found in the POSTQUEL field itself or “$string” to indicate that the parameter is to come from the column in the same tuple with the name “string.” Hence, the above POSTQUEL retrieve command should be specified as:

```
retrieve (result = progress(EMP.salary, EMP.age))
where EMP.name = $name
```

With this specification, Mike can be added to EMP as follows:

```
append to EMP (name = “Mike”, age = 10)
```

The user can now find the progress of Mike in two different ways. First, he can use the registered procedure "progress" as follows:

retrieve (value = progress(EMP.salary, EMP.age))
where EMP.name = "Mike"

This will execute the registered procedure at the time that Mike's tuple is accessed. On the other hand, one can also access the field in EMP corresponding to "progress", i.e.:

retrieve (value = EMP.progress.result)
where EMP.name = "Mike"

This second form has one important advantage, namely the procedure for Mike may have been precomputed since all POSTQUEL fields are candidates for precomputation. If the registered procedure "progress" was flagged as precomputable, then the above POSTQUEL command may have cached answers for a variety of employees. Hence, if the progress of Mike is in the cache, the result is returned directly and no run-time computation need be performed. This is an important optimization if "progress" is a long computation.

The following example suggests another situation in which precomputation of POSTQUEL procedures containing registered procedures in a general purpose programming language is a crucial optimization. Consider a forms management application whereby an individual form is composed of various trim features and fields, each with a collection of attributes. It is desirable that forms be stored in the data base so they can be easily shared by multiple applications. However, it is also important that forms be compiled into an efficient main-memory representation appropriate to the run-time forms management code. Currently, users of INGRES [RTI86] must explicitly compile a form after they are through constructing it. If the form is changed, they must explicitly recompile it anew.

With POSTGRES, one can register a procedure "compile" which accepts as its single argument, the identifier of a form. Then one can register the following POSTQUEL command:

retrieve (result = compile(FORMS.identifier)
where FORMS.identifier = $id

Lastly, one need only declare a FORMS relation as follows:

create FORMS (id = i4, compiled = POSTQUEL[pname])

The compiled version of a form will be created asynchronously by caching the value of the POSTQUEL command. Since the definition of forms changes slowly, the cache will be only infrequently invalidated. Moreover, the user is spared from the difficulty of remembering to compile form definitions. In all cases he simply executes the following retrieve:

retrieve (computation = FORMS.compiled.result)
where FORMS.id = xxx

We close this section by discussing the reasons for not extending the POSTQUEL procedural data types of this section to ones written in a general purpose programming language. First, if a column of a relation was of data type "arbitrary procedure," then there would be no way of knowing the data type of each argument expected by the procedure or the data type of the result. Hence, it is necessary to register all procedures in a general purpose programming language to obtain this information. For POSTQUEL procedures, registration is not necessary because POSTGRES can ascertain the data types of all arguments and the composition of all the result relations.

There are two difficulties with extending parameterized procedures to allow any registered procedure instead of only those written in POSTQUEL. First, the "multiple dot" notation allows

fields to be selected from the output of a procedure by name, and the registration step does not contain a mechanism to name fields in procedural output. Second, the $string notation discussed above cannot easily be extended to registered procedures. Both difficulties do not arise in POSTQUEL procedures.

5. DISCUSSION

This section briefly reviews the power available in the procedural fields described in the previous section.

First, note that a variety of data hierarchies can be effectively modeled. One approach is discussed in a companion paper which uses a single relation to store the form of the type hierarchy and a second relation to store the operators that can be applied to any given object in the hierarchy [ROWE86]. However several others approaches can also be utilized. For example, one can use one or more procedural fields in the relation that corresponds to any given object to assemble the objects which are "inherited" by any given object. This inheritance can be of arbitrary composition, and is not limited to "is-a" hierarchies.

Registered operators must have unique names, so it is not possible to have several operators of the same name and then inherit the one which is "closest" to a given object in some object hierarchy. We considered allowing operators to be multiply defined; however, that would have given us all the messy problems that come with multiple inheritance (i.e. determining which operator to actually use in a specific instance).

Lastly, notice that procedures can be used for many different purposes (e.g. storage of user commands, triggers, rules, data base procedures, the code for operators, etc.). Hence, we feel that utilizing a single powerful construct is a better approach than extending the data model with more anemic capabilities.

REFERENCES

[ADIB80] Adiba, M.E. and Lindsay, B.G., "Database Snapshots," IBM San Jose Res. Tech. Rep. RJ-2772, March 1980.

[BACH77] Bachman, C. and Daya, M., "The Role Concept in Database Models," Proc. 1977 VLDB Conference, Tokyo, Japan, October 1977.

[BATO85] Batory, D. and Kim, W., "Modeling Concepts for VLSI CAD Objects," ACM-TODS, Sept. 1985.

[CHEN76] Chen, P., "The Entity-Relationship Model -- Toward a Unified View of Data," ACM-TODS, June 1976.

[CODD70] Codd, E., "A Relational Model of Data for Large Shared Data Bases," CACM, June 1970.

[CODD79] Codd, E., "Extending Database Relations to Capture More Meaning," ACM-TODS, Dec 1979.

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

[DATE81] Date, C., "Referential Integrity," Proc. Seventh International VLDB Conference, Cannes, France, Sept. 1981.

[GOLD83] Goldberg, A and Robson, D., "Smalltalk-80: The Language and Its Implementation," Addison-Wesley, Reading Mass., 1983.

[HAMM81] Hammer, M. amd McLeod, D., "Database Description with SDM: A Semantic Database Model," ACM-TODS, Sept 1981.

[KATZ85] Katz, R.H., Information Management for Engineering Design, Springer-Verlag, 1985.

[KATZ86] Katz, R. et. al., "Version Modeling Concepts for Computer-Aided Design Databases," Proc. 1986 ACM-SIGMOD International Conference on Management of Data, Washington, D.C., May 1986.

[KENT83] Kent, W., (private communication)

[KORT84] Korth, H. et. al., "System/U: A Database System Based on the Universal Relation Assumption," ACM-TODS, Sept, 1984.

[LOHM83] Lohman, G. et. al., "Remotely Senses Geophysical Databases: Experience and Implications for Generalized DBMS," Proc. 1983 ACM-SIGMOD International Conference on Management of Data, San Jose, Ca., May 1983.

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

[LUM85] Lum, V., et. al., "Design of an Integrated DBMS to Support Advanced Applications," Proc. Int. Conf. on Foundations of Data Org., Kyoto Univ., Japan, May 1985.

[OZSO85] Ozsoyoglu, G. et. al., "A Language and a Physical Organization Technique for Summary Tables," Proc. 1985 ACM-SIGMOD International

Conference on Management of Data, Austin, Tx., June 1985.

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

[RTI85] Relational Technology, Inc., "INGRES Reference Manual, Version 4.0" Alameda, Ca., November 1985.

[ROWE83] Rowe, N., "Top-Down Statistical Estimation on a Database," Proc. 1983 ACM-SIGMOD INternational Conference on Management of Data, San Jose, Ca., May 1983.

[ROWE86] Rowe, L., "A Shared Object Hierarchy," submitted to this conference

[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, July 1977.

[SNOD85] Snodgrass, R and Ahn, I., "A Taxonomy of Time in Databases," Proc 1985 ACM-SIGMOD International Conference on Management of Data, Austin, Tx., June 1985.

[SPON84] Spooner, D., "Database Support for Interactive Computer Graphics," Proc. 1984 ACM-SIGMOD International Conference on Management of data, Boston, Mass., June 1984.

[STON83a] Stonebraker, M., "Document Processing in a Relational Database System," ACM TOOIS, April 1983.

[STON83b] Stonebraker, M., et. al., "Application of Abstract Data Types and Abstract Indexes to CAD Data," Proc. Engineering Applications Stream of 1983 Data Base Week, San Jose, Ca., May 1983.

[STON84] Stonebraker, M. et. al., "QUEL as a Data Type,"Proc. Conference on Management of Data, Boston, Mass., June 1984.

[STON85a] Stonebraker, M., "Triggers and Inference in Data Base Systems," Proc. Islamoora Conference on Expert Data Bases, Islamoora, Fla., Feb 1985, to appear as a Springer-Verlag book.

[STON85b] Stonebraker, M. et. al., "Extending a Relational Data Base System with Procedures," (submitted for publication).

[STON86a] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86b] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[ULLM85] Ullman, J., "Implementation of Logical Query Languages for Databases," ACM-TODS, Sept. 1985.

[WONG80] Wong, E. and Katz, R., "Logical Design and Schema Conversion for Relational and DBTG Databases," Proc. 1980 E-R Conference.

[WOOD83] Woodfill, J. and Stonebraker, M., "An Implementation of Hypothetical Relations," Proc. 9th VLDB Conference, Florence, Italy, Dec. 1983.

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