THE IMPLEMENTATION OF POSTGRES

Michael Stonebraker, Lawrence A. Rowe and Michael Hirohama
EECS Department
University of California, Berkeley

Abstract

Currently, POSTGRES is about 90,000 lines of code in C and is being used by assorted ‘‘bold and brave’’ early users. The system has been constructed by a team of 5 part time students led by a full time chief programmer over the last three years. During this period, we have made a large number of design and implementation choices. Moreover, in some areas we would do things quite differently if we were to start from scratch again. The purpose of this paper is to reflect on the design and implementation decisions we made and to offer advice to implementors who might follow some of our paths. In this paper we restrict our attention to the DBMS ‘‘backend’’ functions. In another paper some of us treat PICASSO, the application development environment that is being built on top of POSTGRES.

1. INTRODUCTION

Current relational DBMSs are oriented toward efficient support for business data processing applications where large numbers of instances of fixed format records must be stored and accessed. Traditional transaction management and query facilities for this application area will be termed **data management.**

To satisfy the broader application community outside of business applications, DBMSs will have to expand to offer services in two other dimensions, namely **object management** and **knowledge management.** Object management entails efficiently storing and manipulating non-traditional data types such as bitmaps, icons, text, and polygons. Object management problems abound in CAD and many other engineering applications. Object-oriented programming languages and data bases provide services in this area.

This research was sponsored by the Defense Advanced Research Projects Agency through NASA Grant NAG 2-530 and by the Army Research Office through Grant DAALO3-87-K-0083.

Knowledge management entails the ability to store and enforce a collection of **rules** that are part of the semantics of an application. Such rules describe integrity constraints about the application, as well as allowing the derivation of data that is not directly stored in the data base.

We now indicate a simple example which requires services in all three dimensions. Consider an application that stores and manipulates text and graphics to facilitate the layout of newspaper copy. Such a system will be naturally integrated with subscription and classified advertisement data. Billing customers for these services will require traditional data management services. In addition, this application must store non-traditional objects including text, bitmaps (pictures), and icons (the banner across the top of the paper). Hence, object management services are required. Lastly, there are many rules that control newspaper layout. For example, the ad copy for two major department stores can never be on facing pages. Support for such rules is desirable in this application.

We believe that **most** real world data management problems are **three dimensional.** Like the newspaper application, they will require a three dimensional solution. The fundamental goal of POSTGRES [STON86, WENS88] is to provide support for such three dimensional applications. To the best of our knowledge it is the first three dimensional data manager. However, we expect that most DBMSs will follow the lead of POSTGRES into these new dimensions.

To accomplish this objective, object and rule management capabilities were added to the services found in a traditional data manager. In the next two sections we describe the capabilities provided and comment on our implementation decisions. Then, in Section 4 we discuss the novel **no-overwrite** storage manager that we implemented in POSTGRES. Other papers have explained the major POSTGRES design decisions in these areas, and we assume that the reader is familiar with [ROWE87] on the data model, [STON88] on rule management, and [STON87] on storage management. Hence, in these three sections we stress considerations that led to our design, what we liked about the design, and the mistakes that we felt we made. Where appropriate we make suggestions for future implementors based on our experience.

Section 5 of the paper comments on specific issues in the implementation of POSTGRES and critiques the choices that we made. In this section we discuss how we interfaced to the operating system, our choice of programming languages and some of our implementation philosophy.

The final section concludes with some performance measurements of POSTGRES. Specifically, we report the results of some of the queries in the Wisconsin benchmark [BITT83].

2. THE POSTGRES DATA MODEL AND QUERY LANGUAGE

2.1. Introduction

Traditional relational DBMSs support a data model consisting of a collection of named relations, each attribute of which has a specific type. In current commercial systems possible types are floating point numbers, integers, character strings, and dates. It is commonly recognized that this data model is insufficient for non-business data processing applications. In designing a new data model and query language, we were guided by the following three design criteria.

1) orientation toward data base access from a query language

We expect POSTGRES users to interact with their data bases primarily by using the set-oriented query language, POSTQUEL. Hence, inclusion of a query language, an optimizer and the corresponding run-time system was a primary design goal.

It is also possible to interact with a POSTGRES data base by utilizing a navigational interface. Such interfaces were popularized by the CODASYL proposals of the 1970's and are enjoying a renaissance in recent object-oriented proposals such as ORION [BANE87] or O2 [VELE89]. Because POSTGRES gives each record a unique identifier (OID), it is possible to use the identifier for one record as a data item in a second record. Using optionally definable indexes on OIDs, it is then possible to navigate from one record to the next by running one query per navigation step. In addition, POSTGRES allows a user to define functions (methods) to the DBMS. Such functions can intersperce statements in a programming language, query language commands, and direct calls to internal POSTGRES interfaces. The ability to directly execute functions which we call **fast path** is provided in POSTGRES and allows a user to navigate the data base by executing a sequence of functions.

However, we do not expect this sort of mechanism to become popular. All navigational interfaces have the same disadvantages of CODASYL systems, namely the application programmer must construct a query plan for each task he wants to accomplish and substantial application maintenance is required whenever the schema changes.

2) Orientation toward multi-lingual access

We could have picked our favorite programming language and then tightly coupled POSTGRES to the compiler and run-time environment of that language. Such an approach would offer **persistence** for variables in this programming language, as well as a query language integrated with the control statements of the language. This approach has been followed in ODE [AGRA89] and many of the recent commercial start-ups doing object-oriented data bases.

Our point of view is that most data bases are accessed by programs written in several different languages, and we do not see any programming language Esperanto on the horizon. Therefore, most application development organizations are **multi-lingual** and require access to a data base from different languages. In addition, data base application packages that a user might acquire, for example to perform statistical or spreadsheet services, are often not coded in the language being used for developing applications. Again, this results in a multi-lingual environment.

Hence, POSTGRES is programming language **neutral,** that is, it can be called from many different languages. Tight integration of POSTGRES to a particular language requires compiler extensions and a run time system specific to that programming language. One of us has built an implementation of persistent CLOS (Common LISP Object System) on top of POSTGRES. Persistent CLOS (or persistent X for any programming language, X) is inevitably language specific. The run-time system must map the disk representation for language objects, including pointers, into the main memory representation expected by the language. Moreover, an object cache must be maintained in the program address space, or performance will suffer badly. Both tasks are inherently language specific.

We expect many language specific interfaces to be built for POSTGRES and believe that the query language plus the **fast path** interface available in POSTGRES offers a powerful, convenient abstraction against which to build these programming language interfaces.

3) small number of concepts

We tried to build a data model with as few concepts as possible. The relational model succeeded in replacing previous data models in part because of its simplicity. We wanted to have as few concepts as possible so that users would have minimum complexity to contend with. Hence, POSTGRES leverages the following three constructs:

types
functions
inheritance

In the next subsection we briefly review the POSTGRES data model. Then, we turn to a short description of POSTQUEL and fast path. We conclude the section with a discussion of whether POSTGRES is object-oriented followed by a critique of our data model and query language.

2.2. The POSTGRES Data Model

As mentioned in the previous section POSTGRES leverages **types** and **functions** as fundamental constructs. There are three kinds of types in POSTGRES and three kinds of functions and we discuss the six possibilities in this section.

Some researchers, e.g. [STON86b, OSBO86], have argued that one should be able to construct new **base types** such as bits, bitstrings, encoded character strings, bitmaps, compressed integers, packed decimal numbers, radix 50 decimal numbers, money, etc. Unlike most next generation DBMSs which have a hard-wired collection of base types (typically integers, floats and character strings), POSTGRES contains an **abstract data type** facility whereby any user can construct an arbitrary number of new base types. Such types can be added to the system while it is executing and require the defining user to specify functions to convert instances of the type to and from the character string data type. Details of the syntax appear in

[WENS88].

The second kind of type available in POSTGRES is a **constructed type.**[**] A user can create a new type by constructing a **record** of base types and instances of other constructed types. For example:

```
create DEPT (dname = c10, floor = integer, floorspace = polygon)
create EMP (name = c12, dept = DEPT, salary = float)
```

Here, DEPT is a type constructed from an instance of each of three base types, a character string, an integer and a polygon. EMP, on the other hand, is fabricated from base types and other constructed types.

A constructed type can optionally **inherit** data elements from other constructed types. For example, a SALESMAN type can be created as follows:

```
create SALESMAN (quota = float) inherits EMP
```

In this case, an instance of SALESMAN has a quota and inherits all data elements from EMP, namely name, dept and salary. We had the standard discussion about whether to include single or multiple inheritance and concluded that a single inheritance scheme would simply be too restrictive. As a result POSTGRES allows a constructed type to inherit from an arbitrary collection of other constructed types.

When ambiguities arise because an object has multiple parents with the same field name, we elected to refuse to create the new type. However, we isolated the resolution semantics in a single routine, which can be easily changed to track multiple inheritance semantics as they unfold over time in programming languages.

We now turn to the POSTGRES notion of functions. There are three different classes of POSTGRES functions,

normal functions
operators
POSTQUEL functions

and we discuss each in turn.

A user can define an arbitrary collection of **normal functions** whose operands are base types or constructed types. For example, he can define a function, area, which maps an instance of a polygon into an instance of a floating point number. Such functions are automatically available in the query language as illustrated in the following example:

```
retrieve (DEPT.dname) where area (DEPT.floorspace) > 500
```

[**] In this section the reader can use the words **constructed type, relation,** and **class** interchangeably. Moreover, the words **record, instance,** and **tuple** are similarly interchangeable. This section has been purposely written with the chosen notation to illustrate a point about object-oriented data bases which is discussed in Section 2.5.

Normal functions can be defined to POSTGRES while the system is running and are dynamically loaded when required during query execution.

Functions are allowed on constructed types, e.g:

retrieve (EMP.name) where overpaid (EMP)

In this case overpaid has an operand of type EMP and returns a boolean. Functions whose operands are constructed types are inherited down the type hierarchy in the standard way.

Normal functions are arbitrary procedures written in a general purpose programming language (in our case C or LISP). Hence, they have arbitrary semantics and can run other POSTQUEL commands during execution. Therefore, queries with normal functions in the qualification cannot be optimized by the POSTGRES query optimizer. For example, the above query on overpaid employees will result in a sequential scan of all employees.

To utilize indexes in processing queries, POSTGRES supports a second class of functions, called **operators.** Operators are functions with one or two operands which use the standard operator notation in the query language. For example the following query looks for departments whose floor space has a greater area than that of a specific polygon:

retrieve (DEPT.dname) where DEPT.floorspace AGT polygon["(0,0), (1,1), (0,2)"]

The "area greater than" operator AGT is defined by indicating the token to use in the query language as well as the function to call to evaluate the operator. Moreover, several **hints** can also be included in the definition which assist the query optimizer. One of these hints is that ALE is the negator of this operator. Therefore, the query optimizer can transform the query:

retrieve (DEPT.dname) where not (DEPT.floorspace ALE polygon["(0,0), (1,1), (0,2)"])

which cannot be optimized into the one above which can be. In addition, the design of the POSTGRES access methods allows a B+-tree index to be constructed for the instances of floorspace appearing in DEPT records. This index can support efficient access for the **class** of operators {ALT, ALE, AE, AGT, AGE}. Information on the access paths available to the various operators is recorded in the POSTGRES system catalogs.

As pointed out in [STON87b] it is imperative that a user be able to construct new access methods to provide efficient access to instances of non-traditional base types. For example, suppose a user introduces a new operator "!!" defined on polygons that returns true if two polygons overlap. Then, he might ask a query such as:

retrieve (DEPT.dname) where DEPT.floorspace !! polygon["(0,0), (1,1), (0,2)"]

There is no B+-tree or hash access method that will allow this query to be rapidly executed. Rather, the query must be supported by some multidimensional access method such as R-trees, grid files, K-D-B trees, etc. Hence, POSTGRES was designed to allow new access methods to be written by POSTGRES users and then dynamically added to the system. Basically, an access method to POSTGRES is a collection of 13 normal functions which perform record level operations such as fetching the next record in a scan, inserting a new record, deleting a specific record, etc. All a user need do is define implementations for each of these functions and make a collection of entries in the system catalogs.

Operators are only available for operands which are base types because access methods traditionally support fast access to specific fields in records. It is unclear what an access method for a constructed type should do, and therefore POSTGRES does not include this capability.

The third kind of function available in POSTGRES is **POSTQUEL functions.** Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function. For example, the following function defines the overpaid employees:

```
define function high-pay as retrieve (EMP.all) where EMP.salary > 50000
```

POSTQUEL functions can also have parameters, for example:

```
define function ret-sal as retrieve (EMP.salary) where EMP.name = $1
```

Notice that ret-sal has one parameter in the body of the function, the name of the person involved. Such parameters must be provided at the time the function is called. A third example POSTQUEL function is:

```
define function set-of-DEPT as retrieve (DEPT.all) where DEPT.floor = $.floor
```

This function has a single parameter "$.floor". It is expected to appear in a record and receives the value of its parameter from the floor field defined elsewhere in the same record.

Each POSTQUEL function is automatically a constructed type. For example, one can define a FLOORS type as follows:

```
create FLOORS (floor = i2, depts = set-of-DEPT)
```

This constructed type uses the set-of-DEPT function as a constructed type. In this case, each instance of FLOORS has a value for depts which is the value of the function set-of-DEPT for that record.

In addition, POSTGRES allows a user to form a constructed type, one or more of whose fields has the special type POSTQUEL. For example, a user can construct the following type:

```
create PERSON (name = c12, hobbies = POSTQUEL)
```

In this case, each instance of hobbies contains a different POSTQUEL function, and therefore each person has a name and a POSTQUEL function that defines his particular hobbies. This support for POSTQUEL as a type allows the system to simulate non-normalized relations as found in NF**2 [DADA86].

POSTQUEL functions can appear in the query language in the same manner as normal functions. The following example ensures that Joe has the same salary as Sam:

```
replace EMP (salary = ret-sal("Joe")) where EMP.name = "Sam"
```

In addition, since POSTQUEL functions are a constructed type, queries can be executed against POSTQUEL functions just like other constructed types. For example, the following query can be run on the constructed type, high-pay:

```
retrieve (high-pay.salary) where high-pay.name = "george"
```

If a POSTQUEL function contains a single retrieve command, then it is very similar to a relational view definition, and this capability allows retrieval operations to be performed on objects which are essentially relational views.

Lastly, every time a user defines a constructed type, a POSTQUEL function is automatically defined with the same name. For example, when DEPT is constructed, the following function is automatically defined:

```
define function DEPT as retrieve (DEPT.all) where DEPT.OID = $1
```

When EMP was defined earlier in this section, it contained a field dept which was of type DEPT. In fact, DEPT was the above automatically defined POSTQUEL function. As a result, instance of a constructed type is available as a type because POSTGRES automatically defines a POSTQUEL function for each such type.

POSTQUEL functions are a very powerful notion because they allow arbitrary collections of instances of types to be returned as the value of the function. Since POSTQUEL functions can reference other POSTQUEL functions, arbitrary structures of complex objects can be assembled. Lastly, POSTQUEL functions allow collections of commands such as the 5 SQL commands that make up TP1 [ANON85] to be assembled into a single function and stored inside the DBMS. Then, one can execute TP1 by executing the single function. This approach is preferred to having to submit the 5 SQL commands in TP1 one by one from an application program. Using a POSTQUEL function, one replaces 5 round trips between the application and the DBMS with 1, which results in a 25% performance improvement in a typical OLTP application.

2.3. The POSTGRES Query Language

The previous section presented several examples of the POSTQUEL language. It is a set oriented query language that resembles a superset of a relational query language. Besides user defined functions and operators which were illustrated earlier, the features which have been added to a traditional relational language include:

path expressions
support for nested queries
transitive closure

support for inheritance

support for time travel

Path expressions are included because POSTQUEL allows constructed types which contain other constructed types to be hierarchically referenced. For example, the EMP type defined above contains a field which is an instance of the constructed type, DEPT. Hence, one can ask for the names of employees who work on the first floor as follows:

```
retrieve (EMP.name) where EMP.dept.floor = 1
```

rather than being forced to do a join, e.g:

```
retrieve (EMP.name) where EMP.dept = DEPT.OID and DEPT.floor = 1
```

POSTQUEL also allows queries to be nested and has operators that have sets of instances as operands. For example, to find the departments which occupy an entire floor, one would query:

```
retrieve (DEPT.dname)
where DEPT.floor NOTIN {D.floor from D in DEPT where D.dname != DEPT.dname}
```

In this case, the expression inside the curly braces represents a set of instances and NOTIN is an operator which takes a set of instances as its right operand.

The transitive closure operation allows one to explode a parts or ancestor hierarchy. Consider for example the constructed type:

```
parent (older, younger)
```

One can ask for all the ancestors of John as follows:

```
retrieve* into answer (parent.older)
using a in answer
where parent.younger = "John"
or parent.younger = a.older
```

In this case the * after retrieve indicates that the associated query should be run until answer fails to grow.

If one wishes to find the names of all employees over 40, one would write:

```
retrieve (E.name) using E in EMP
where E.age > 40
```

On the other hand, if one wanted the names of all salesmen or employees over 40, the notation is:

```
retrieve (E.name) using E in EMP*
where E.age > 40
```

Here the * after the constructed type EMP indicates that the query should be run over EMP and all constructed types under EMP in the inheritance hierarchy. This use of * allows a user to easily run queries over a constructed type and all its descendents.

Lastly, POSTGRES supports the notion of **time travel.** This feature allows a user to run historical queries. For example to find the salary of Sam at time T one would query:

```
retrieve (EMP.salary)
using EMP [T]
where EMP.name = "Sam"
```

POSTGRES will automatically find the version of Sam's record valid at the correct time and get the appropriate salary.

Like relational systems, the result of a POSTQUEL command can be added to the data base as a new constructed type. In this case, POSTQUEL follows the lead of relational systems by removing duplicate records from the result. The user who is interested in retaining duplicates can do so by ensuring that the OID field of some instance is included in the target list being selected. For a full description of POSTQUEL the interested reader should consult [WENS88].

2.4. Fast Path

There are three reasons why we chose to implement a **fast path** feature. First, a user who wishes to interact with a data base by executing a sequence of functions to navigate to desired data can use fast path to accomplish his objective. Second, there are a variety of decision support applications in which the end user is given a specialized query language. In such environments, it is often easier for the application developer to construct a parse tree representation for a query rather than an ASCII one. Hence, it would be desirable for the application designer to be able to directly call the POSTGRES optimizer or executor. Most DBMSs do not allow direct access to internal system modules.

The third reason is a bit more complex. In the persistent CLOS layer of PICASSO, it is necessary for the run time system to assign a unique identifier (OID) to every constructed object that is persistent. It is undesirable for the system to synchronously insert each object directly into a POSTGRES data base and thereby assign a POSTGRES identifier to the object. This would result in poor performance in executing a persistent CLOS program. Rather, persistent CLOS maintains a cache of objects in the address space of the program and only inserts a persistent object into this cache synchronously. There are several options which control how the cache is written out to the data base at a later time. Unfortunately, it is essential that a persistent object be assigned a unique identifier at the time it enters the cache, because other objects may have to point to the newly created object and use its OID to do so.

If persistent CLOS assigns unique identifiers, then there will be a complex mapping that must be performed when objects are written out to the data base and real POSTGRES unique identifiers are assigned. Alternately, persistent CLOS must maintain its own system for unique identifiers, independent of the

POSTGRES one, an obvious duplication of effort. The solution chosen was to allow persistent CLOS to access the POSTGRES routine that assigns unique identifiers and allow it to preassign N POSTGRES object identifiers which it can subsequently assign to cached objects. At a later time, these objects can be written to a POSTGRES data base using the preassigned unique identifiers. When the supply of identifiers is exhausted, persistent CLOS can request another collection.

In all of these examples, an application program requires direct access to a user-defined or internal POSTGRES function, and therefore the POSTGRES query language has been extended with:

function-name (param-list)

In this case, besides running queries in POSTQUEL, a user can ask that any function known to POSTGRES be executed. This function can be one that a user has previously defined as a normal, operator, or POSTQUEL function or it can be one that is included in the POSTGRES implementation.

Hence, the user can directly call the parser, the optimizer, the executor, the access methods, the buffer manager or the utility routines. In addition he can define functions which in turn make calls on POSTGRES internals. In this way, he can have considerable control over the low level flow of control, much as is available through a DBMS toolkit such as Exodus [RICH87], but without all the effort involved in configuring a tailored DBMS from the toolkit. Moreover, should the user wish to interact with his data base by making a collection of function calls (method invocations), this facility allows the possibility. As noted in the introduction, we do not expect this interface to be especially popular.

The above capability is called **fast path** because it provided direct access to specific functions without checking the validity of parameters. As such, it is effectively a remote procedure call facility and allows a user program to call a function in another address space rather than in its own address space.

2.5. Is POSTGRES Object-oriented?

There have been many next generation data models proposed in the last few years. Some are characterized by the term "extended relational", others are considered "object-oriented" while yet others are termed "nested relational". POSTGRES could be accurately described as an object-oriented system because it includes unique identity for objects, abstract data types, classes (constructed types), methods (functions), and inheritance for both data and functions. Others (e.g. [ATKI89]) are suggesting definitions for the word "object-oriented", and POSTGRES satisfies virtually all of the proposed litmus tests.

On the other hand, POSTGRES could also be considered an extended relational system. As noted in a previous footnote, Section 2 could have been equally well written with the word "constructed type" and "instance" replaced by the words "relation" and "tuple". In fact, in previous descriptions of POSTGRES [STON86], this notation was employed. Hence, others, e.g. [MAIE89] have characterized POSTGRES as an extended relational system.

Lastly, POSTGRES supports the POSTQUEL type, which is exactly a nested relational structure. Consequently, POSTGRES could be classified as a nested relational system as well.

As a result POSTGRES could be described using any of the three adjectives above. In our opinion we can interchangeably use the words **relations, classes,** and **constructed types** in describing POSTGRES. Moreover, we can also interchangeably use the words **function** and **method.** Lastly, we can interchangeably use the words **instance, record,** and **tuple.** Hence, POSTGRES seems to be either object-oriented or not object-oriented, depending on the choice of a few tokens in the parser. As a result, we feel that most of the efforts to classify the extended data models in next generation data base systems are silly exercises in surface syntax.

In the remainder of this section, we comment briefly on the POSTGRES implementation of OIDs and inheritance. POSTGRES gives each record a unique identifier (OID), and then allows the application designer to decide for each constructed type whether he wishes to have an index on the OID field. This decision should be contrasted with most object-oriented systems which construct an OID index for all constructed types in the system automatically. The POSTGRES scheme allows the cost of the index to be paid only for those types of objects for which it is profitable. In our opinion, this flexibility has been an excellent decision.

Second, there are several possible ways to implement an inheritance hierarchy. Considering the SALESMEN and EMP example noted earlier, one can store instances of SALEMAN by storing them as EMP records and then only storing the extra quota information in a separate SALESMAN record. Alternately, one can store no information on each salesman in EMP and then store complete SALESMAN records elsewhere. Clearly, there are a variety of additional schemes.

POSTGRES chose one implementation, namely storing all SALESMAN fields in a single record. However, it is likely that applications designers will demand several other representations to give them the flexibility to optimize their particular data. Future implementations of inheritance will likely require several storage options.

2.6. A Critique of the POSTGRES Data Model

There are five areas where we feel we made mistakes in the POSTGRES data model:

union types
access method interface
functions
big objects
arrays

We discuss each in turn.

A desirable feature in any next-generation DBMS would be to support union types, i.e. an instance of a type can be an instance of one of several given types. A persuasive example (similar to one from [COPE84]) is that employees can be on loan to another plant or on loan to a customer. If two base types, customer and plant exist, one would like to change the EMP type to:

```
create EMP (name = c12, dept = DEPT, salary = float, on-loan-to = plant or customer)
```

Unfortunately including union types makes a query optimizer more complex. For example, to find all the employees on loan to the same organization one would state the query:

```
retrieve (EMP.name, E.name)
using E in EMP
where EMP.on-loan-to = E.on-loan-to
```

However, the optimizer must construct two different plans, one for employees on loan to a customer and one for employees on loan to a different plant. The reason for two plans is that the equality operator may be different for the two types. In addition, one must construct indexes on union fields, which entails substantial complexity in the access methods.

Union types are **highly** desirable in certain applications, and we considered three possible stances with respect to union types:

1) support only through abstract data types
2) support through POSTQUEL functions
3) full support

Union types can be easily constructed using the POSTGRES abstract data type facility. If a user wants a specific union type, he can construct it and then write appropriate operators and functions for the type. The implementation complexity of union types is thus forced into the routines for the operators and functions and onto the implementor of the type. Moreover, it is clear that there are a vast number of union types and an extensive type library must be constructed by the application designer. The PICASSO team stated that this approach placed an unacceptably difficult burden on them, and therefore position 1 was rejected.

Position 2 offers some support for union types but has problems. Consider the example of employees and their hobbies from [STON86]:

```
create EMP (name = c12, hobbies = POSTQUEL)
```

Here the hobbies field is a POSTQUEL function, one per employee, which retrieves all hobby information about that particular employee. Now consider the following POSTQUEL query:

```
retrieve (EMP.hobbies.average) where EMP.name = "Fred"
```

In this case the field average for each hobby record will be returned whenever it is defined. Suppose, however, that average is a float for the softball hobby and an integer for the cricket hobby. In this case, the application program must be prepared to accept values of different types.

The more difficult problem is the following legal POSTQUEL query:

```
retrieve into TEMP (result = EMP.hobbies.average) where EMP.name = "Fred"
```

In this case, a problem arises concerning the type of the result field, because it is a union type. Hence, adopting position 2 leaves one in an awkward position of not having a reasonable type for the result of the above query.

Of course, position 3 requires extending the indexing and query optimization routines to deal with union types. Our solution was to adopt position 2 and to add an abstract data type, ANY, which can hold an instance of any type. This solution which turns the type of the result of the above query from

```
one-of {integer, float}
```

into ANY is not very satisfying. Not only is information lost, but we are also forced to include with POSTGRES this universal type.

In our opinion, the only realistic alternative is to adopt position 3, swallow the complexity increase, and that is what we would do in any next system.

Another failure concerned the access method design and was the decision to support indexing only on the value of a field and not on a function of a value. The utility of indexes on functions of values is discussed in [LYNC88], and the capability was retrofitted, rather inelegantly, into one version of POSTGRES [AOKI89].

Another comment on the access method design concerns extendibility. Because a user can add new base types dynamically, it is essential that he also be able to add new access methods to POSTGRES if the system does not come with an access method that supports efficient access to his types. The standard example of this capability is the use of R-trees [GUTM84] to speed access to geometric objects. We have now designed and/or coded three access methods for POSTGRES in addition to B+-trees. Our experience has consistently been that adding an access method is **VERY HARD.** There are four problems that complicate the situation. First, the access method must include explicit calls to the POSTGRES locking subsystem to set and release locks on access method objects. Hence, the designer of a new access method must understand locking and how to use the particular POSTGRES facilities. Second, the designer must understand how to interface to the buffer manager and be able to get, put, pin and unpin pages. Next, the POSTGRES execution engine contains the ‘‘state’’ of the execution of any query and the access methods must understand portions of this state and the data structures involved. Last but not least, the designer must write 13 non-trivial routines. Our experience so far is that novice programmers can add new types to POSTGRES; however, it requires a highly skilled programmer to add a new access method. Put differently, the manual on how to add new data types to POSTGRES is 2 pages long, the one for access methods is 50 pages.

We failed to realize the difficulty of access method construction. Hence, we designed a system that allows end users to add access methods dynamically to a running system. However, access methods will be built by sophisticated system programmers who could have used a simpler to build interface.

A third area where our design is flawed concerns POSTGRES support for POSTQUEL functions. Currently, such functions in POSTGRES are collections of commands in the query language POSTQUEL. If one defined budget in DEPT as a POSTQUEL function, then the value for the shoe department budget might be the following command:

```
retrieve (DEPT.budget) where DEPT.dname = "candy"
```

In this case, the shoe department will automatically be assigned the same budget as the candy department. However, it is impossible for the budget of the shoe department to be specified as:

```
if floor = 1  then
   retrieve (DEPT.budget) where DEPT.dname = "candy"
else
   retrieve (DEPT.budget) where DEPT.dname = "toy"
```

This specification defines the budget of the shoe department to be the candy department budget if it is on the first floor. Otherwise, it is the same as the toy department. This query is not possible because POSTQUEL has no conditional expressions. We had **extensive** discussions about this and other extensions to POSTQUEL. Each such extension was rejected because it seemed to turn POSTQUEL into a programming language and not a query language.

A better solution would be be to allow a POSTQUEL function to be expressible in a general purpose programming language enhanced with POSTQUEL queries. Hence, there would be no distinction between normal functions and POSTQUEL functions. Put differently, normal functions would be able to be constructed types and would support path expressions.

There are three problems with this approach. First, path expressions for normal functions cannot be optimized by the POSTGRES query optimizer because they have arbitrary semantics. Hence, most of the optimizations planned for POSTQUEL functions would have to be discarded. Second, POSTQUEL functions are much easier to define than normal functions because a user need not know a general purpose programming language. Also, he need not specify the types of the function arguments or the return type because POSTGRES can figure these out from the query specification. Hence, we would have to give up ease of definition in moving from POSTQUEL functions to normal functions. Lastly,, normal functions have a protection problem because they can do arbitrary things, such as zeroing the data base. POSTGRES deals with this problem by calling normal functions in two ways:

trusted -- loaded into the POSTGRES address space
untrusted -- loaded into a separate address space

Hence, normal functions are either called quickly with no security or slowly in a protected fashion. No such security problem arises with POSTQUEL functions.

An better approach might have been to support POSTQUEL functions written in the 4th generation language (4GL) being designed for PICASSO [ROWE89]. This programming system leaves type information in the system catalogs. Consequently, there would be no need for a separate registrations step to indicate type information to POSTGRES. Moreover, a processor for the language is available for integration in POSTGRES. It is also easy to make a 4GL "safe", i.e. unable to perform wild branches or malicious actions Hence, there would be no security problem. Also, it seems possible that path expressions could be optimized for 4GL functions.

Current commercial relational products seem to be moving in this direction by allowing data base procedures to be coded in their proprietary 4th generation languages (4GLs). In retrospect we probably should have looked seriously at designing POSTGRES to support functions written in a 4GL.

Next, POSTGRES allows types to be constructed that are of arbitrary size. Hence, large bitmaps are a perfectly acceptable POSTGRES data type. However, the current POSTGRES user interface (portals) allows a user to fetch one or more instances of a constructed type. It is currently impossible to fetch only a portion of an instance. This presents an application program with a severe buffering problem; it must be capable of accepting an entire instance, no matter how large it is. We should extend the portal syntax in a straightforward way to allow an application to position a portal on a specific field of an instance of a constructed type and then specify a byte-count that he would like to retrieve. These changes would make it much easier to insert and retrieve big fields.

Lastly, we included arrays in the POSTGRES data model. Hence, one could have specified the SALESMAN type as:

```
create SALESMAN (name = c12, dept = DEPT, salary = float, quota = float[12])
```

Here, the SALESMAN has all the fields of EMP plus a quota which is an array of 12 floats, one for each month of the year. In fact, character strings are really an array of characters, and the correct notation for the above type is:

```
create SALESMAN (name = c[12], dept = DEPT, salary = float, quota = float[12])
```

In POSTGRES we support fixed and variable length arrays of base types, along with an array notation in POSTQUEL. For example to request all salesmen who have an April quota over 1000, one would write:

```
retrieve (SALESMAN.name) where SALESMAN.quota[4] > 1000
```

However, we do not support arrays of constructed types; hence it is not possible to have an array of instances of a constructed type. We omitted this capability only because it would have made the query optimizer and executor somewhat harder. In addition, there is no built-in search mechanism for the elements of an array. For example, it is not possible to find the names of all salesmen who have a quota over 1000 during any month of the year. In retrospect, we should included general support for arrays or no support at all.

3. THE RULES SYSTEM

3.1. Introduction

It is clear to us that all DBMSs need a rules system. Current commercial systems are required to support referential integrity [DATE81], which is merely a simple-minded collection of rules. In addition, most current systems have special purpose rules systems to support relational views, protection, and integrity constraints. Lastly, a rules system allows users to do event-driven programming as well as enforce integrity constraints that cannot be performed in other ways. There are three high level decisions that the POSTGRES team had to make concerning the philosophy of rule systems.

First, a decision was required concerning how many rule syntaxes there would be. Some approaches, e.g. [ESWA76, WIDO89] propose rule systems oriented toward application designers that would augment other rule systems present for DBMS internal purposes. Hence, such systems would contain several independently functioning rules systems. On the other hand, [STON82] proposed a rule system that tried to support user functionality as well as needed DBMS internal functions in a single syntax.

From the beginning, a goal of the POSTGRES rules system was to have only one syntax. It was felt that this would simplify the user interface, since application designers need learn only one construct. Also, they would not have to deal with deciding which system to use in the cases where a function could be performed by more than one rules system. It was also felt that a single rules system would ease the implementation difficulties that would be faced.

Second, there are two implementation philosophies by which one could support a rule system. The first is a **query rewrite** implementation. Here, a rule would be applied by converting a user query to an alternate form prior to execution. This transformation is performed between the query language parser and the optimizer. Support for views [STON75] is done this way along with many of the proposals for recursive query support [BANC86, ULLM85]. Such an implementation will be very efficient when there are a small number of rules on any given constructed type and most rules cover the whole constructed type. For example, a rule such as:

EMP [dept] contained-in DEPT[dname]

expresses the referential integrity condition that employees cannot be in a non-existent department and applies to all EMP instances. However, a query rewrite implementation will not work well if there are a large number of rules on each constructed type, each of them covering only a few instances. Consider, for example, the following three rules:

employees in the shoe department have a steel desk

employees over 40 have a wood desk

employees in the candy department do not have a desk

To retrieve the kind of a desk that Sam has, one must run the following three queries:

```
retrieve (desk = ‘‘steel’’) where EMP.name = ‘‘Sam’’ and EMP.dept = ‘‘shoe’’
retrieve (desk = ‘‘wood’’) where EMP.name= ‘‘Sam’’ and EMP.age > 40
retrieve (desk = null) where EMP.name = ‘‘Sam’’ and EMP.dept = ‘‘candy’’
```

Hence, a user query must be rewritten for each rule, resulting in a serious degradation of performance unless all queries are processed as a group using multiple query optimization techniques [SELL86].

Moreover, a query rewrite system has great difficulty with **exceptions** [BORG85]. For example consider the rule ‘‘all employees have a steel desk’’ together with the exception ‘‘Jones is an employee who has a wood desk’’. If one ask for the kind of desk and age for all employees over 35, then the query must be rewritten as the following 2 queries:

```
retrieve (desk = "steel", EMP.age) where EMP.age > 35 and EMP.name != "Jones"
retrieve (desk = "wood", EMP.age) where EMP.age > 35 and EMP.name = "Jones"
```

In general, the number of queries as well as the complexity of their qualifications increases linearly with the number of rules. Again, this will result in bad performance unless multiple query optimization techniques are applied.

Lastly, a query rewrite system does not offer any help in resolving situations when the rules are violated. For example, the above referential integrity rule is silent on what to do if a user tries to insert an employee into a non-existent department.

On the other hand, one could adopt a **trigger** implementation based on individual record accesses and updates to the data base. Whenever a record is accessed, inserted, deleted or modified, the low level execution code has both the old record and the new record readily available. Hence, assorted actions can easily be taken by the low level code. Such an implementation requires the rule firing code to be placed deep in the query execution routines. It will work well if there are many rules each affecting only a few instances, and it is easy to deal successfully with conflict resolution at this level. However, rule firing is deep in the executor, and it is thereby impossible for the query optimizer to construct an efficient execution plan for a chain of rules that are awakened.

Hence, this implementation complements a query rewrite scheme in that it excels where a rewrite scheme is weak and vica-versa. Since we wanted to have a single rule system, it was clear that we needed to provide both styles of implementation.

A third issue that we faced was the paradigm for the rules system. A conventional production system consisting of collections of if-then rules has been explored in the past [ESWA76, STON82], and is a readily available alternative. However, such a scheme lacks expressive power. For example, suppose one wants to enforce a rule that Joe makes the same salary as Fred. In this case, one must specify two different if-then rules. The first one indicates the action to take if Fred receives a raise, namely to propagate the change on to Joe. The second rule specifies that any update to Joe's salary must be refused. Hence, many user rules require two or more if-then specifications to achieve the desired effect.

The intent in POSTGRES was to explore a more powerful paradigm. Basically, any POSTGRES command can be turned into a rule by changing the semantics of the command so that it is logically either **always** running or **never** running. For example, Joe may be specified to have the same salary as Fred by the rule:

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

This single specification will propagate Joe's salary on to Fred as well as refuse direct updates to Fred's salary. In this way a single ‘‘always’’ rule replaces the two statements needed in a production rule syntax.

Moreover, to efficiently support the triggering implementation where there are a large number of rules present for a single constructed type, each of which applies to only a few instances, the POSTGRES team designed a sophisticated marking scheme whereby rule wake-up information is placed on individual instances. Consequently, regardless of the number of rules present for a single constructed type, only those which actually must fire will be awakened. This should be contrasted to proposals without such data structures, which will be hopelessly inefficient whenever a large number of rules are present for a single constructed type.

Lastly, the decision was made to support the query rewrite scheme by escalating markers to the constructed type level. For example, consider the rule:

```
always replace EMP (age = 40) where name != "Bill"
```

This rule applies to all employees except Bill and it would be a waste of space to mark each individual employee. Rather, one would prefer to set a single marker in the system catalogs to cover the whole constructed type implicitly. In this case, any query, e.g:

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

will be altered prior to execution by the query rewrite implementation to:

```
retrieve (age = 40) where EMP.name = "Sam" and EMP.name != "Bill"
```

At the current time much of the POSTGRES Rules System (PRS) as described in [STON88] is operational, and there are three aspects of the design which we wish to discuss in the next three subsections, namely:

complexity
absence of needed function and
efficiency

Then, we close with the second version of the POSTGRES Rules system (PRS II) which we are currently designing. This rules system is described in more detail in [STON89, STON89b].

3.2. Complexity

The first problem with PRS is that the implementation is exceedingly complex. It is difficult to explain the marking mechanisms that cause rule wake-up even to a sophisticated person. Moreover, some of us have an uneasy feeling that the implementation may not be quite correct. The fundamental problem can be illustrated using the Joe-Fred example above. First, the rule must be awakened and run whenever Fred's salary changes. This requires that one kind of marker be placed on the salary of Fred. However, if Fred is given a new name, say Bill, then the rule must be deleted and reinstalled. This requires a second kind of marker on the name of Fred. Additionally, it is inappropriate to allow any update to Joe's salary; hence a third kind of marker is required on that field. Furthermore, if Fred has not yet been hired, then the rule must take effect on the insertion of his record. This requires a marker to be placed in the index for employee names. To support rules that deal with ranges of values, for example:

```
always replace EMP (age = 40)
where EMP.salary > 50000 and EMP.salary < 60000
```

we require that two ‘‘stub’’ markers be placed in the index to denote the ends of the scan. In addition, each intervening index record must also be marked. Ensuring that all markers are correctly installed and appropriate actions taken when record accesses and updates occur has been a challenge.

Another source of substantial complexity is the necessity to deal with priorities. For example, consider a second rule:

```
always replace EMP (age = 50) where EMP.dept = "shoe"
```

In this case a highly paid shoe department employee would be given two different ages. To alleviate this situation, the second rule could be given a higher priority, e.g:

```
always replace EMP (age = 50) where EMP.dept = "shoe"
priority = 1
```

The default priority for rules is 0; hence the first rule would set the age of highly paid employees to 40 unless they were in the shoe department, in which case their age would be set to 50 by the second rule.

Priorities, of course, add complications to the rules system. For example, if the second rule above is deleted, then the first rule must be awakened to correct the ages of employees in the shoe department.

Another aspect of complexity is our decision to support both early and late evaluation of rules. Consider the example rule that Joe makes the same salary as Fred. This rule can be awakened when Fred gets a salary adjustment, or activation can be delayed until a user requests the salary of Joe. Activation can be delayed as long as possible in the second case, and we term this **late** evaluation while the former case is termed **early** evaluation. This flexibility also results in substantial extra complexity. For example, certain rules cannot be activated late. If salaries of employees are indexed, then the rule that sets Joe's salary to that of Fred must be activated early because the index must be kept correct. Moreover, it is impossible for an early rule to read data that is written by a late rule. Hence, additional restrictions must be imposed.

Getting PRS correct has entailed uncounted hours of discussion and considerable implementation complexity. The bottom line is that the implementation of a rule system that is clean and simple to the user is, in fact, extremely complex and tricky. Our personal feeling is that we should have embarked on a more modest rules system.

3.3. Absence of Needed Function

The definition of a **useful** rules system is one that can handle at least all of the following problems in one integrated system:

support for views
protection
referential integrity
other integrity constraints

We focus in this section on support for views. The query rewrite implementation of a rules system should be able to translate queries on views into queries on real objects. In addition, updates to views should be similarly mapped to updates on real objects.

There are various special cases of view support that can be performed by PRS, for example materialized views. Consider the following view definition:

```
define view SHOE-EMP (name = EMP.name, age = EMP.age, salary = EMP.salary)
where EMP.dept = ‘‘shoe’’
```

The following two PRS rules specify a materialization of this view:

```
always append to SHOE-EMP (name = EMP.name, salary = EMP.salary) where EMP.dept = ‘‘shoe’’
always delete SHOE-EMP where SHOE-EMP.name not-in {EMP.name where EMP.dept = ‘‘shoe’’}
```

In this case, SHOE-EMP will always contain a correct materialization of the shoe department employees, and queries can be directed to this materialization.

However, there seemed to be no way to support updates on views that are not materialized. One of us has spent countless hours attempting to support this function through PRS and failed. Hence, inability to support operations conventional views is a major weakness of PRS.

3.4. Implementation Efficiency

The current POSTGRES implementation uses markers on individual fields to support rule activation. The only escalation supported is to convert a collection of field level markers to a single marker on the entire constructed type. Consequently, if a rule covers a single instance, e.g:

```
always replace EMP (salary = 1000) where EMP.name = "Sam"
```

then a total of 3 markers will be set, one in the index, one on the salary field and one on the name field. Each marker is composed of:

rule-id	-- 6 bytes
priority	-- 1 byte
marker-type	-- 1 byte

Consequently, the marker overhead for the rule is 24 bytes, Now consider a more complex rule:

```
always replace EMP (salary = 1000) where EMP.dept = "shoe"
```

If 1000 employees work in the shoe department, then 24,000 bytes of overhead will be consumed in markers. The only other option is to escalate to a marker on the entire constructed type, in which case the rule will be activated if any salary is read or written and not just for employees in the shoe department. This will be an overhead intensive option. Hence, for rules which cover many instances but not a significant fraction of all instances, the POSTGRES implementation will not be very space efficient.

We are considering several solutions to this problem. First, we have generalized B+-trees to efficiently store interval data as well as point data. Such ‘‘segmented B+-trees’’ are the subject of a separate paper [KOLE89]. This will remove the space overhead in the index for the dominant form of access method. Second, to lower the overhead on data records, we will probably implement markers at the physical block level as well as at the instance and constructed type levels. The appropriate extra granularities are currently under investigation.

3.5. The Second POSTGRES Rules System

Because of the inability of the current rules paradigm to support views and to a lesser extent the fundamental complexity of the implementation, we are converting to a second POSTGRES rules system (PRS II). This rules system has much in common with the first implementation, but returns to the traditional production rule paradigm to obtain sufficient control to perform view updates correctly. This section outlines our thinking, and a complete proposal appears in [STON89b].

The production rule syntax we are using in PRS II has the form:

```
ON event TO object WHERE POSTQUEL-qualification
THEN DO POSTQUEL-command(s)
```

Here, event is retrieve, replace, delete, append, new (i. e. replace or append) or old (i.e. delete or replace). Moreover, object is either the name of a constructed type or constructed-type.column. POSTQUEL-qualification is a normal qualification, with no additions or changes. Lastly, POSTQUEL-commands is a set of POSTQUEL commands with the following two changes:

NEW, OLD or CURRENT can appear instead of the name of a constructed type in front of any attribute.

refuse (target-list) is added as a new POSTQUEL command

In this notation we would specify the "Fred-Joe" rule as:

```
on NEW EMP.salary where EMP.name = "Fred"
then do
  replace E (salary = CURRENT.salary)
  using E in EMP
  where E.name = "Joe"

on NEW EMP.salary where EMP.name = "Joe"
then do
  refuse
```

Notice, that PRS II is less powerful than the "always" system because the Fred-Joe rule require two specifications instead of one.

PRS II has both a query rewrite implementation and a trigger implementation, and it is an optimization decision which one to use as noted in [STON89b]. For example, consider the rule:

```
on retrieve to SHOE-EMP
then do
retrieve (EMP.name, EMP.age, EMP.salary) where EMP.dept = "shoe"
```

Any query utilizing such a rule, e.g:

```
retrieve (SHOE-EMP.name) where SHOE-EMP.age < 40
```

would be processed by the rewrite implementation to:

```
retrieve (EMP.name) where EMP.age < 40 and EMP.dept = ‘‘shoe’’
```

As can be seen, this is identical to the query modification performed in relational view processing techniques [STON75]. This rule could also be processed by the triggering system, in which case the rule would materialize the records in SHOE-EMP iteratively.

Moreover, it is straightforward to support additional functionality, such as allowing multiple queries in the definition of a view. Supporting materialized views can be efficiently done by **caching** the action part of the above rule, i.e. executing the command before a user requests evaluation. This corresponds to moving the rule to early evaluation. Lastly, supporting views that are partly materialized and partly specified as procedures as well as views that involve recursion appears fairly simple. In [STON89b] we present details on these extensions.

Consider the following collection of rules that support updates to SHOE-EMP:

```
on NEW SHOE-EMP
then do
  append to EMP (name = NEW.name, salary = NEW.salary)

on OLD SHOE-EMP
then do
  delete EMP where EMP.name = OLD.name and EMP.salary = OLD.salary

on UPDATE to SHOE-EMP
then do
  replace EMP (name = NEW.name, salary = NEW.salary)
  where EMP.name = NEW.name
```

If these rules are processed by the trigger implementation, then an update to SHOE-EMP, e.g:

```
replace SHOE-EMP (salary = 1000) where SHOE-EMP.name = ‘‘Mike’’
```

will be processed normally until it generates a collection of

```
[new-record, old-record]
```

pairs. At this point the triggering system can be activated to make appropriate updates to underlying constructed types. Moreover, if a user wishes non-standard view update semantics, he can perform any particular actions he desires by changing the action part of the above rules.

PRS II thereby allows a user to use the rules system to define semantics for retrievals and updates to views. In fact, we expect to build a compiler that will convert a higher level view notration into the needed collection of PRS II rules. In addition, PRS II retains all functionality of the first rules system, so protection, alerters integrity constraints, and arbitrary triggers are readily expressed. The only disadvantage is that PRS II requires two rules to perform many tasks expressible as a single PRS rule. To overcome this disadvantage, we will likely continue to support the PRS syntax in addition to the PRS II syntax and compile

PRS into PRS II. support

PRS II can be supported by the same implementation that we proposed for the query rewrite implementation of PRS, namely marking instances in the system catalogs. Moreover, the query rewrite algorithm is nearly the same as in the first implementation. The triggering system can be supported by the same instance markers as in PRS. In fact, the implementation is bit simpler because a couple of the types of markers are not required. Because the implementation of PRS II is so similar to our initial rules system, we expect to have the conversion completed in the near future.

4. STORAGE SYSTEM

4.1. Introduction

When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different. All current commercial systems use a storage manager with a write-ahead log (WAL), and we felt that this technology was well understood. Moreover, the original INGRES prototype from the 1970s used a similar storage manager, and we had no desire to do another implementation.

Hence, we seized on the idea of implementing a ‘‘no-overwrite’’ storage manager. Using this technique the old record remains in the data base whenever an update occurs, and serves the purpose normally performed by a write-ahead log. Consequently, POSTGRES has no log in the conventional sense of the term. Instead the POSTGRES log is simply 2 bits per transaction indicating whether each transaction committed, aborted, or is in progress.

Two very nice features can be exploited in a no-overwrite system. First, aborting a transaction can be instantaneous because one does not need to process the log undoing the effects of updates; the previous records are readily available in the data base. More generally, to recover from a crash, one must abort all the transactions in progress at the time of the crash. This process can be effectively instantaneous in POSTGRES.

The second benefit of a no-overwrite storage manager is the possibility of **time travel.** As noted earlier, a user can ask a historical query and POSTGRES will automatically return information from the record valid at the correct time.

This storage manager should be contrasted with a conventional one where the previous record is overwritten with a new one. In this case a write-ahead log is required to maintain the previous version of each record. There is no possibility of time travel because the log cannot be queried since it is in a different format. Moreover, the data base must be restored to a consistent state when a crash occurs by processing the log to undo any partially completed transactions. Hence, there is no possibility of instantaneous crash recovery.

Clearly a no-overwrite storage manager is superior to a conventional one if it can be implemented at comparable performance. There is a brief hand-wave of an argument in [STON87] that alleges this might be the case. In our opinion, the argument hinges around the existence of **stable** main memory. In the absence of stable memory, a no-overwrite storage manager must force to disk at commit time all pages written by a transaction. This is required because the effects of a committed transaction must be durable in case a crash occurs and main memory is lost. A conventional data manager on the other hand, need only force to disk at commit time the log pages for the transaction's updates. Even if there are as many log pages as data pages (a highly unlikely occurence), the conventional storage manager is doing sequential I/O to the log while a no-overwrite storage manager is doing random I/O. Since sequential I/O is substantially faster than random I/O, the no-overwrite solution is guaranteed to offer worse performance.

However, if stable main memory is present then neither solution must force pages to disk. In this environment, performance should be comparable. Hence, with stable main memory it appears that a no-overwrite solution is competitive. As computer manufacturers offer some form of stable main memory, a no-overwrite solution may become a viable storage option.

In designing the POSTGRES storage system, we were guided by two philosophical premises. First, we decided to make a clear distinction between current data and historical data. We expected access patterns to be highly skewed toward current records. In addition, queries to the archive might look very different from those accessing current data. For both reasons, POSTGRES maintains two different physical collections of records, one for the current data and one for historical data, each with its own indexes.

Second, our design assumes the existence of a randomly addressable archive device on which historical records are placed. Our intuitive model for this archive is an optical disk. Our design was purposely made consistent with an archive that has a write-once-read-many (WORM) orientation. This characterizes many of the optical disks on the market today.

In the next subsection we indicate two problems with the POSTGRES design. Then, in Section 5.3 we make additional comments on the storage manager.

4.2. Problems in the POSTGRES Design

There are at least two problems with our design. First, it is unstable under heavy load. An asynchronous demon, known as vacuum cleaner, is responsible for moving historical records from the magnetic disk structure holding the current records to the archive where historical records remain. Under normal circumstances, the magnetic disk portion of each constructed type is (say) only 1.1 times the minimum possible size of the constructed type. Of course, the vacuum cleaner consumes CPU and I/O resources running in background achieving this goal. However, if the load on a POSTGRES data base increases, then the vacuum cleaner may not get to run. In this case the magnetic disk portion of a constructed type will increase, and performance will suffer because the execution engine must read historical records on the magnetic disk during the (presumably frequent) processing of queries to the current data base. As a result, performance will degrade proportionally to the excess size of the magnetic disk portion of the data base. As load increases, the vacuum cleaner gets less resources, and performance degrades as the size of the magnetic disk data base increases. This will ultimately result in a POSTGRES data base going into melt-down.

Obviously, the vacuum cleaner should be run in background if possible so that it can consume resources at 2:00 A.M. when there is little other activity. However, if there is consistent heavy load on a system, then the vacuum cleaner must be scheduled at the same priority as other tasks, so the above instability does not occur. The bottom line is that scheduling the vacuum cleaner is a tricky optimization problem.

The second comment which we wish to make is that future archive systems are likely to be read/write, and rewritable optical disks have already appeared on the market. Consequently, there is no reason for us to have restricted ourselves to WORM technology. Certain POSTGRES assumptions were therefore unnecessary, such as requiring the current portion of any constructed type to be on magnetic disk.

4.3. Other Comments

Historical indexes will usually be on a combined key consisting of a time range together with one or more keys from the record itself. Such two-dimensional indexes can be stored using the technology of R-trees [GUTM84], R+-trees [FALO87] or perhaps in some new way. We are not particularly comfortable that good ways to index time ranges have been found, and we encourage additional work in this area. A possible approach is segmented R-trees which we are studying [KOLE89].

Another comment concerns POSTGRES support for time travel. There are many tasks that are very difficult to express with our mechanisms. For example, the query to find the time at which Sam's salary increased from $5000 to $6000 is very tricky in POSTQUEL.

A last comment is that time travel can be implemented with a conventional transaction system using a write ahead log. For example, one need only have an ‘‘archive’’ constructed type for each physical constructed type for which time travel is desired. When a record is updated, its previous value is written in the archive with the appropriate timestamps. If the transaction fails to commit, this archive insert and the corresponding record update is unwound using a conventional log. Such an implementation may well have substantial benefits, and we should have probably considered such a possibility. In making storage system decisions we were guided by a missionary zeal to do something different than a conventional write ahead log scheme. Hence, we may have overlooked other intriguing options.

5. THE POSTGRES IMPLEMENTATION

5.1. Introduction

POSTGRES contains a fairly conventional parser, query optimizer and execution engine. Two aspects of the implementation deserve special mention,

dynamic loading and the process structure
choice of implementation language

and we discuss each in turn.

5.2. Dynamic Loading and Process Structure

POSTGRES assumes that data types, operators and functions can be added and subtracted dynamically, i.e. while the system is executing. Moreover, we have designed the system so that it can accommodate a potentially very large number of types and operators. Consequently, the user functions that support the implementation of a type must be dynamically loaded and unloaded. Hence, POSTGRES maintains a cache of currently loaded functions and dynamically moves functions into the cache and then ages them out of the cache. Moreover, the parser and optimizer run off of a main memory cache of information about types and operators. Again this cache must be maintained by POSTGRES software. It would have been much easier to assume that all types and operators were linked into the system at POSTGRES initialization time and have required a user to reinstall POSTGRES when he wished to add or drop types. Moreover, users of prototype software are not running systems which cannot go down for rebooting. Hence, the function is not essential.

Second, the rules system forces significant complexity on the design. A user can add a rule such as:

```
always retrieve (EMP.salary)
where EMP.name = "Joe"
```

In this case his application process wishes to be notified of any salary adjustment to Joe. Consider a second user who gives Joe a raise. The POSTGRES process that actually does the adjustment will notice that a marker has been placed on the salary field. However, in order to alert the first user, one of four things must happen:

a) POSTGRES could be designed as a single server process. In this case within the current process the first user’s query could simply be activated. However, such a design is incompatible with running on a shared memory multiprocessor, where a so-called multi-server is required. Hence, this design was discarded.

b) The POSTGRES process for the second user could run the first user’s query and then connect to his application process to deliver results. This requires that an application process be coded to expect communication from random other processes. We felt this was too difficult to be a reasonable solution.

c) The POSTGRES process for the second user could connect to the input socket for the first user's POSTGRES and deliver the query to be run. The first POSTGRES would run the query and then send results to the user. This would require careful synchronization of the input socket among multiple independent command streams. Moreover, it would require the second POSTGRES to know the portal name on which the first user's rule was running.

d) The POSTGRES process for the second user could alert a special process called the **POSTMASTER.** This process would in turn alert the process for the first user where the query would be run and the results delivered to the application process.

We have adopted the fourth design as the only one we thought was practical. However, we have thereby constructed a process through which everybody must channel communications. If the POSTMASTER crashes, then the whole POSTGRES environment must be restarted. This is a handicap, but we could think of no better solution. Moreover, there are a collection of system demons, including the vacuum cleaner mentioned above, which need a place to run. In POSTGRES they are run as subprocesses managed by the POSTMASTER.

A last aspect of our design concerns the operating system process structure. Currently, POSTGRES runs as one process for each active user. This was done as an expedient to get a system operational as quickly as possible. We plan on converting POSTGRES to use lightweight processes available in the operating systems we are using. These include PRESTO for the Sequent Symmetry and threads in Version 4 of Sun/OS.

5.3. Programming Language Used

At the beginning of the project, we were forced to make a commitment to a programming language and machine environment. The machine was an easy one, since SUN workstations were nearly omnipresent at Berkeley, and any other choice would have been non-standard. However, we were free to choose any language in which to program. We considered the following:

C
C++
MODULA 2+
LISP
ADA
SMALLTALK

We dismissed SMALLTALK quickly because we felt it was too slow and compilers were not readily available for a wide variety of platforms. We felt it desirable to keep open the option of distributing our software widely. We felt ADA and MODULA 2+ offered limited advantages over C++ and were not widely used in the Berkeley environment. Hence, obtaining pretrained programmers would have been a problem. Lastly, we were not thrilled to use C, since INGRES had been coded in C and we were anxious to choose a different language, if only for the sake of doing something different. At the time we started (10/85), there was not a stable C++ compiler, so we did not seriously consider this option.

By a process of elimination, we decided to try writing POSTGRES in LISP. We expected that it would be especially easy to write the optimizer and inference engine in LISP, since both are mostly tree processing modules. Moreover, we were seduced by AI claims of high programmer productivity for applications written in LISP.

We soon realized that parts of the system were more easily coded in C, for example the buffer manager which moves 8K pages back and forth to the disk and uses a modified LRU algorithm to control what pages are resident. Hence, we adopted the policy that we would use both C and LISP and code modules of POSTGRES in whichever language was most appropriate. By the time Version 1 was operational, it contained about 17000 lines of LISP and about 63000 lines of C.

Our feeling is that the use of LISP has been a terrible mistake for several reasons. First, current LISP environments are very large. To run a ‘‘nothing’’ program in LISP requires about 3 mbytes of address space. Hence, POSTGRES exceeds 4 mbytes in size, all but 1 mbyte is the LISP compiler, editor and assorted other non required (or even desired) functions. Hence, we suffer from a gigantic footprint. Second, a DBMS never wants to stop when garbage collection happens. Any response time sensitive program must therefore allocate and deallocate space manually, so that garbage collection never happens during normal processing. Consequently, we spent extra effort ensuring that LISP garbage collection is not used by POSTGRES. Hence, this aspect of LISP, which improves programmer productivity, was not available to us. Third, LISP execution is slow. As noted in the performance figures in the next section our LISP code is more than twice as slow as the comparable C code. Of course, it is possible that we are not skilled LISP programmers or do not know how to optimize the language; hence our experience should be suitably discounted.

However, none of these irritants was the real disaster. We have found that debugging a two language system is extremely difficult. The C debugger, of course, knows nothing about LISP while the LISP debugger knows nothing about C. As a result, we have found debugging POSTGRES to be a painful and frustrating task. Memory allocation bugs were among the most painful since LISP and C have very different models of dynamic memory. Of course, it is true that the optimizer and inference engine were easier to code in LISP. Hence, we saved some time there. However, this was more than compensated by the requirement of writing a lot of utility code that would convert LISP data structures into C and vica versa.

In fact, our assessment is that the primary productivity increases in LISP come from the nice programming environment (e.g. interactive debugger, nice workstation tools, etc.) and not from the language itself. Hence, we would encourage the implementors of other programming languages to study the LISP environment carefully and implement the better ideas.

As a result we have just finished moving our 17000 lines of LISP to C to avoid the debugging hassle and secondarily to avoid the performance and footprint problems in LISP. Our experience with LISP and two language systems has not been positive, and we would caution others not to follow in our footsteps.

6. STATUS AND PERFORMANCE

At the current time (October 1989) the LISP-less Version 1 of POSTGRES has been in the hands of users for a short time, and we are shaking the last bugs out of the C port. In addition, we have designed all of the additional functionality to appear in Version 2. The characteristics of Version 1 are:

a) The query language POSTQUEL runs except for aggregates, functions and set operators.

b) All object management capabilities are operational except POSTQUEL types.

c) Some support for rules exists. Specifically, replace always commands are operational; however the implementation currently only supports early evaluation and only with markers on whole columns.

d) The storage system is complete. However, we are taking delivery shortly on an optical disk jukebox, and so the archive is currently not implemented on a real optical disk. Moreover, R-trees to support time travel are not yet implemented.

e) Transaction management runs.

The focus has been on getting the function in POSTGRES to run. So far, only minimal attention has been paid to performance. Figure 1 shows assorted queries in the Wisconsin benchmark and gives results for three systems running on a SUN 3/280. All numbers are run on a non-quiescent system so there may be significant fluctuations. The first two are the C and LISP versions of POSTGRES. These are functionally identical systems with the same algorithms embodied in the code. The footprint of the LISP system is about 4.5 Mbytes while the C system is about 1 Mbyte. For comparison purposes we also include the performance numbers for the commercial version of INGRES in the third column. As can be seen, the LISP system is several times slower than the C system. In various other benchmarks we have never seen the C

	POSTGRES C-based	POSTGRES LISP-based	INGRES RTI 5.0
nullqry	0.4	0.3	0.2
scan 10Ktups	36.	180.	5.2
retrieve into query 1% selectivity	38.	n/a	9.9
append to 10Ktup	4.7	180.	0.4
delete from 10Ktup	37.	n/a	5.7
replace in 10Ktup	42.	280.	5.7

A Comparison of INGRES and POSTGRES
(Times are listed in seconds per query.)
Figure 1

system less than twice as fast as the LISP system. Moreover, the C system is several times slower than a commercial system. The Public domain version of INGRES that we worked on in the mid 1970's is about a factor of two slower than commercial INGRES. Hence, it appears that POSTGRES s about one-half the speed of the original INGRES. There are substantial inefficiencies in POSTGRES, especially in the code which checks that a retrieved record is valid. We expect that subsequent tuning will get us somewhere in between the performance of Public domain INGRES and RTI INGRES.

7. CONCLUSIONS

In this section we summarize our opinions about certain aspects of the design of POSTGRES. First, we are uneasy about the complexity of the POSTGRES data model. The comments in Section 2 all contain suggestions to make it more complex. Moreover, other research teams have tended to construct even more complex data models, e.g. EXTRA [CARE88]. Consequently, a simple concept such as referential integrity, which can be done in only one way in existing commercial systems, can be done in several different ways in POSTGRES. For example, the user can implement an abstract data type and then do the required checking in the input conversion routine. Alternately, he can use a rule in the POSTGRES rules system. Lastly, he can use a POSTQUEL function for the field that corresponds to the foreign key in a current relational system. There are complex performance tradeoffs between these three solutions, and a decision must be made by a sophisticated application designer. We fear that real users, who have a hard time with data base design for existing relational systems, will find the next-generation data models, such as the one in POSTGRES, impossibly complex. The problem is that applications exist where each representation is the only acceptable one. The demand for wider application of data base technology ensures that vendors will produce systems with these more complex data models.

Another source of uneasiness is the fact that rules and POSTQUEL functions have substantial overlap in function. For example, a POSTQUEL function can be simulated by one rule per record, albeit at some performance penalty. On the other hand, all rules, except retrieve always commands, can be alternately implemented using POSTQUEL functions. We expect to merge the two concepts in Version 2, and our proposal appears in [STON89b].

In the areas of rules and storage management, we are basically satisfied with POSTGRES capabilities. The syntax of the rule system should be changed as noted in Section 3; however this is not a significant issue and it should be available easily in Version 2. The storage manager has been quite simple to implement. Crash recovery code has been easy to write because the only routine which must be carefully written is the vacuum cleaner. Moreover, access to past history seems to be a highly desirable capability.

Furthermore, the POSTGRES implementation certainly erred in the direction of excessive sophistication. For example, new types and functions can be added on the fly without recompiling POSTGRES. It would have been much simpler to construct a system that required recompilation to add a new type. Second, we have implemented a complete transaction system in Version 1. Other prototypes tend to assume a single user environment. In these and many other ways, we strove for substantial generality; however the net effect has been to slow down the implementation effort and make the POSTGRES internals much more complex. As a result, POSTGRES has taken us considerably longer to build than the original version of INGRES. One could call this the ‘‘second system’’ effect. It was essential that POSTGRES be more usable than the original INGRES prototype in order for us to feel like we were making a contribution.

A last comment concerns technology transfer to commercial systems. It appears that the process is substantially accelerating. For example, the relational model was constructed in 1970, first prototypes of implementations appeared around 1976-77, commercial versions first surfaced around 1981 and popularity of relational systems in the marketplace occurred around 1985. Hence, there was a 15 year period during which the ideas were transferred to commercial systems. Most of the ideas in POSTGRES and in other next generation systems date from 1984 or later. Commercial systems embodying some of these ideas have already appeared and major vendors are expected to have advanced systems within the next year or two. Hence, the 15 year period appears to have shrunk to less than half that amount. This acceleration is impressive, but it will lead to rather short lifetimes for the current collection of prototypes.

REFERENCES

[AGRA89] Agrawal, R. and Gehani, N., "ODE: The Language and the Data Model," Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Or., May 1989.

[ATKI89] Atkinson, M. et. al., ‘‘The Object-oriented Database System Manifesto,’’ Altair Technical Report 30-89, Rocquencourt, France, August 1989.

[ANON85] Anon et. al., ‘‘A Measure of Transaction Processing Power,’’ Tandem Computers, Cupertino, CA, Technical Report 85.1, 1985.

[AOKI89] Aoki, P., ‘‘Implementation of Extended Indexes in POSTGRES,’’ Electronics Research Laboratory, University of California, Technical Report 89-62, July 1989.

[BANC86] Bancilhon, F. and Ramakrishnan, R., ‘‘An Amateur’s Introduction to Recursive Query Processing,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[BANE87] Banerjee, J. et. al., ‘‘Semantics and Implementation of Schema Evolution in Object-oriented Databases,’’ Proc. 1987 ACM SIGMOD Conference on Management of Data, San Francisco, Ca., May 1987.

[BITT83] Bitton, D. et. al., ‘‘Benchmarking Database Systems: A Systematic Approach,’’ Proc. 1983 VLDB Conference, Cannes, France, Sept. 1983.

[BORG85] Borgida, A., ``Language Features for Flexible Handling of Exceptions in Information Systems,´´ ACM-TODS, Dec. 1985.

[CARE88] Carey, M. et. al., ‘‘A Data Model and Query Language for EXODUS,’’ Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

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

[DADA86] Dadam, P. et. al., ‘‘A DBMS Prototype to Support NF2 Relations,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

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

[ESWA76] Eswaren, K., ‘‘Specification, Implementation and Interactions of a Rule Subsystem in an Integrated Database System,’’ IBM Research, San Jose, Ca., Research Report RJ1820, August 1976.

[FALO87] Faloutsos, C. et. al., ‘‘Analysis of Object Oriented Spatial Access Methods,’’ Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, Ca., May 1987.

[GUTM84] Gutman, A., ‘‘R-trees: A Dynamic Index Structure for Spatial Searching,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[KOLE89] Kolovson, C. and Stonebraker, M., ‘‘Segmented Search Trees and their Application to Data Bases,’’ (in preparation).

[LYNC88] Lynch, C. and Stonebraker, M., ‘‘Extended User-Defined Indexing with Application to Textual Databases,’’ Proc. 1988 VLDB Conference, Los Angeles, Ca., Sept. 1988.

[MAIE89] Maier, D., ‘‘Why Isn’t There an Object-oriented Data Model?’’ Proc. 11th IFIP World Congress, San Francisco, Ca., August 1989.

[OSBO86] Osborne, S. and Heaven, T., ‘‘The Design of a Relational System with Abstract Data Types as Domains,’’ ACM TODS, Sept. 1986.

[RICH87] Richardson, J. and Carey, M., ‘‘Programming Constructs for Database System Implementation in EXODUS,’’ Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, Ca., May 1987.

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," Proc. 1987 VLDB Conference, Brighton, England, Sept 1987.

[ROWE89] Rowe, L. et. al., ‘‘The Design and Implementation of Picasso,’’ (in preparation).

[SELL86] Sellis, T., ‘‘Global Query Optimization,’’ Proc 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1986.

[STON75] Stonebraker, M., ‘‘Implementation of Integrity Constraints and Views by Query Modification,’’ Proc. 1975 ACM-SIGMOD Conference, San Jose, Ca., May 1975.

[STON82] Stonebraker, M. et. al., ‘‘A Rules System for a Relational Data Base Management System,’’ Proc. 2nd International Conference on Databases,’’ Jerusalem, Israel, June 1982 (available from Academic press).

[STON86] Stonebraker, M. and Rowe, L., ‘‘The Design of POSTGRES,’’ Proc. 1986 ACM-SIGMOD Conference, Washington, D.C., June 1986.

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

[STON87] Stonebraker, M., "The POSTGRES Storage System," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON87b] Stonebraker, M. et. al., ‘‘Extensibility in POSTGRES,’’ IEEE Database Engineering, Sept. 1987.

[STON88] Stonebraker, M. et. al., "The POSTGRES Rules System," IEEE Transactions on Software Engineering, July 1988.

[STON89] Stonebraker, M. et. al., ‘‘Commentary on the POSTGRES Rules System,’’ SIGMOD RECORD, Sept. 1989.

[STON89b] Stonebraker, M. et. al., ‘‘Rules, Procedures and Views,’’ (in preparation).

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

[VELE89] Velez, F. et. al., ‘‘The O2 Object manager: An Overview,’’ GIP ALTAIR, Le Chesnay, France, Technical Report 27-89, February 1989.

[WENS88] Wensel, S. (ed.), ‘‘The POSTGRES Reference Manual,’’ Electronics Research Laboratory, University of California, Berkeley, CA, Report M88/20, March 1988.

[WIDO89] Widom, J. and Finkelstein, S., ‘‘A Syntax and Semantics for Set-oriented Production Rules in Relational Data Bases, IBM Research, San Jose, Ca., June 1989.