# THE POSTGRES NEXT GENERATION DBMS

*Michael Stonebraker and Greg Kemnitz*
*EECS Department*
*University of California, Berkeley*

## Abstract

The purpose of the POSTGRES project was to build a **next generation** DBMS to rectify the known deficiencies in current relational DBMSs. This system, constructed over a four year period by one full time programmer and 3-4 part time students is operational and consists of about 180,000 lines of C. POSTGRES is available free of charge and is being used by perhaps 125 sites around the world. This paper describes the major concepts of the system and details its current state. We restrict our attention to the DBMS "backend" functions, and make only passing mention of the front end tools available for POSTGRES.

## 1. INTRODUCTION

Commercial 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. The traditional transaction management and query facilities for this application area will be termed **data management,** and are addressed by relational systems.

To satisfy the needs of users outside of business applications, DBMSs must be expanded 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.

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.

---

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.

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.

A second example requiring all three services is indicated in [COMM90]. Hence, we believe that **most** real world data management problems that will arise in the 1990s are inherently **three dimensional,** and require **data, object,** and **knowledge** management services. The fundamental goal of POSTGRES [STON86, STON90, KEMN91B] is to provide support for such applications.

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 in these two areas. Then, in Section 4 we discuss the novel **no-overwrite** storage manager that we implemented in POSTGRES, and the notion of **time travel** that it supports. Section 5 continues with some of the implementation philosophy of POSTGRES. Section 6 indicates the current status of the system and indicates its current performance on a subset of the Wisconsin benchmark [BITT83] and on an engineering benchmark [CATT91]. Section 7 then ends the paper with a collection of conclusions.

The POSTGRES DBMS has been under construction since 1986. The initial concepts for the system were presented in [STON86] and the initial data model appeared in [ROWE87]. Our storage manager concepts are detailed in [STON87], and the first rule system that we implemented is discussed in [STON88]. Our first "demo-ware" was operational in 1987, and we released Version 1 of POSTGRES to a few external users in June 1989. A critique of Version 1 of POSTGRES appears in [STON90]. Version 2 followed in June 1990, and it included a new rules system documented in [STON90B]. We are now delivering Version 2.1, which is the subject of this paper. Further information on this system can be obtained from the reference manual [KEMN91B], the POSTGRES tutorial [KEMN91] and the release notes.

POSTGRES is now about 180,000 lines of code in C and has been written by a team consisting of a full time chief programmer and 3-4 part time students. It runs on Sun 3, Sun 4, DECstation, and Sequent Symmetry machines and can be obtained free of charge over the internet or on tape for a modest reproduction fee. For details on obtaining POSTGRES, please call or write:

Claire Mosher
521 Evans Hall
University of California

Berkeley, Ca. 94720
(415) 642-4662

# 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, money, and dates. It is commonly recognized that this data model is insufficient for future 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 used in some of the recent object-oriented systems. 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 intersperse statements in a programming language, query language commands, and direct calls to internal POSTGRES interfaces, such as the get_record routine in the access methods. Such functions are available to users in the query language or they can be directly executed. The latter capability is termed **fast path,** because it allows a programmer to package a collection of direct calls to POSTGRES internals into a user executable function. This will support highest possible performance by bypassing any unneeded portion of POSTGRES functionality.

As a result a POSTGRES application programmer is provided great flexibility in style of interaction, since he can intersperse queries, navigation, and direct function execution. This will allow him to use the query language and obtain data independence and automatic optimization or to selectively give up these benefits to obtain higher performance.

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 object-oriented DBMSs.

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 programming shops 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 in-house 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 any particular language requires compiler extensions and a run time system specific to that programming language. Another research group has built an implementation of persistent CLOS (Common LISP Object System) on top of POSTGRES [WANG88] and we are planning a version of persistent C++ in the future. 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. The reader is directed to [STON91] which discusses our approach to embedding POSTGRES capabilities in C++.

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 four constructs:

classes
inheritance
types
functions

In the next subsection we briefly review the POSTGRES data model. Then, we turn to a short description of POSTQUEL and fast path.

## 2.2. The POSTGRES Data Model

The fundamental notion in POSTGRES is that of a **class**,** which is a named collection of **instances** of objects. Each instance has the same collection of named **attributes** and each attribute is of a specific **type.** Moreover, each instance has a unique (never-changing) identifier (OID).

A user can create a new class by specifying the class name, along with all attribute names and their types, for example.

```
create EMP (name = c12, salary = float, age = int)
```

A class can optionally **inherit** data elements from other classes. For example, a SALESMAN class 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, salary and age. We had the standard discussion about whether to include single or multiple inheritance and concluded that a single inheritance scheme would be too restrictive. As a result POSTGRES allows a class to inherit from an arbitrary collection of other **parent** classes. When ambiguities arise because a class inherits the same attribute name from multiple parents, we elected to refuse to create the new class. 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.

There are three kinds of classes. First a class can be a **real** (or base) class whose instances are stored in the data base. Alternately a class can be a **derived** class (or **view** or **virtual** class) whose instances are not physically stored but are materialized only when necessary. Definition and maintenance of views is considered in Section 3.5. Lastly, a class can be a **version** of another class, in which case it is stored as a **differential** relative to its **parent** class. Again Section 3.5 discusses in more detail how this mechanism works.

POSTGRES contains an extensive type system and a powerful notion of functions. There are three kinds of types in POSTGRES, base types, arrays of base types, and composite types, which we discuss in

---

** In this section the reader can use the words **class, constructed type,** and **relation** interchangeably. Moreover, the words **record, instance,** and **tuple** are similarly interchangeable. In fact, previous descriptions of the POSTGRES data model (i.e. [ROWE87, STON90]) used other terminology than this paper.

turn.

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 many next generation DBMSs which have a hard-wired collection of base types (typically integers, floats and character strings), POSTGRES contains an **abstract data type (ADT)** facility whereby any user can construct arbitrary 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 [KEMN91B]. Consequently, it is possible to construct a class, DEPT, as follows:

Create DEPT (dname = c10, manager = c12, floorspace = polygon, mailstop = point)

Here, a DEPT instance contains four attributes, the first two have familiar types while the third is a polygon indicating the space allocated to the department and the fourth is the geographic location of the mailstop.

A user can assign values to attributes of base types in POSTQUEL by either specifying a constant or a function which returns the correct type, e.g:

replace DEPT (mailstop = "(10,10)") where DEPT.dname = "shoe"
replace DEPT (mailstop = center (DEPT.polygon)) where DEPT.dname = "toy"

Arrays of base types are also supported as POSTGRES types. Therefore, if employees receive a different salary each month, we could redefine the EMP class as:

create EMP (name = c12, salary = float[12], age = int)

Arrays are supported in the POSTQUEL query language using the standard bracket notation, e.g:

retrieve (EMP.name) where EMP.salary[4] = 1000.
replace EMP (salary[6] = salary[5]) where EMP.name = "Jones"
replace EMP (salary = "12, 14, 16, 18, 20, 19, 17, 15, 13, 11, 9, 10") where EMP.name = "Fred"

Composite types allow an application designer to construct **complex objects,** i.e. attributes which contain other instances as part or all of their value. Hence, complex objects have a hierarchical internal structure, and POSTGRES supports two kinds of composite types. First, zero or more instances of any class is automatically a composite type. For example, the EMP class can be redefined to have attributes, manager and co-workers, each of which holds a collection of zero or more instances of the EMP class:

create EMP (name = c12, salary = float[12], age = int, manager = EMP, co-workers = EMP)

Consequently, each time a class is constructed, a type is automatically available to hold a collection of instances of the class.

In the above example manager and co-workers have the same structure for each instance of EMP. However, there are situations where the application designer requires a complex object which does not have this rigid structure. For example, consider extending the EMP class to keep track of the hobbies that each employee engages in. For example, Joe might engage in windsurfing and softball while Bill participates in bicycling, skiing, and skating. For each hobby, we must record hobby-specific information. For example, softball data includes the team the employee plays on, his position and batting average while windsurfing data includes the type of board owned and mean time to getting wet. It is clear that hobbies information for each employee is best modeled as a collection of zero or more instances of **various** classes. Moreover, each employee can have differently structured instances. To accomodate this diversity, POSTGRES supports a final constructed type, **set,** whose value is a collection of instances from all classes. Using this construct, hobbies information can be added to the EMP class as follows:

```
add to EMP (hobbies = set)
```

In summary, complex objects are supported in POSTGRES by two composite types. The first, indicated by a class name, contains zero or more instances of that class while the second, indicated by **set,** holds zero or more instances of any classes.

Composite types are supported in POSTQUEL by the concept of **path expressions.** Since manager in the EMP class is a composite type, its elements can be hierarchically addressed by a **nested dot** notation. For example to find the age of the manager of Joe, one would write:

```
retrieve (EMP.manager.age) where EMP.name = "Joe"
```

rather than being forced to perform some sort of a join. This nested dot notation is also found in IRIS [WILK90], ORION [KIM90], O2 [DEUX90], and EXTRA [CARE88].

Composite types can have a value which is a function which returns the correct type, e.g:

```
replace EMP (hobbies = compute-hobbies("Jones")) where EMP.name = "Jones"
```

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

C functions
operators
POSTQUEL functions

A user can define an arbitrary number of **C functions** whose arguments are base types or composite 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 query which finds the names of departments for which area returns a result greater than 500:

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

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

C functions can also have an argument which is a class name, e.g:

retrieve (EMP.name) where overpaid (EMP)

In this case overpaid has an operand of type EMP and returns a boolean, and the query finds the names of all employees for which overpaid returns true. A function whose argument is a class name is inherited down the class hierarchy in the standard way. Hence, overpaid is automatically available for the SALESMAN class. In some circles such functions are called **methods.** Moreover, overpaid can either be considered as a function using the above syntax or as a new attribute for EMP whose type is the return type of the function. Using the latter interpretation, the user can restate the above query as:

retrieve (EMP.name) where EMP.overpaid

Hence, overpaid is interchangeably a function defined for each instance of EMP or a new attribute for EMP. The same interpretation of such functions appears in IRIS [WILK90].

C functions are arbitrary C procedures. Hence, they have arbitrary semantics and can run arbitrary POSTQUEL commands during execution. Therefore, queries with C 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 instances of the class.

To utilize indexes in processing queries, POSTGRES supports a second kind of function, 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 "(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 "(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 any base type. Consequently, a B-tree index for floorspace in DEPT supports efficient access for the **collection** of operators {ALT, ALE, AE, AGT, AGE}. Information on the access paths available for 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 "!!" that returns true if two polygons overlap. Then, he might ask a query such as:

retrieve (DEPT.dname) where DEPT.floorspace !! "(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 C 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 high-paid employees:

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

POSTQUEL functions can also have parameters, for example:

define function sal-lookup (c12) returns float as
retrieve (EMP.salary) where EMP.name = $1

Notice that sal-lookup has one argument in the body of the function, the name of the person involved. This argument must be provided at the time the function is called.

Such functions may be placed in a query, e.g:

retrieve (EMP.name) where EMP.salary = sal-lookup("Joe")

or they can be directly executed using the fast path facility to be described in Section 2.4:

sal-lookup ("Joe")

Moreover, attributes of a composite type automatically have values which are functions that return the correct type. For example, consider the function:

```
define function mgr-lookup (c12) returns EMP as
retrieve (EMP.all) where EMP.name = DEPT.manager and DEPT.name = $1
```

This function can be used to assign values to the manager attribute in the EMP class, for example:

```
append to EMP (name = "Sam", salary = 1000, age = 40, manager = mgr-lookup ("shoe"))
```

Like C functions, POSTQUEL functions can have a specific class as an argument:

```
define function neighbors (DEPT) returns DEPT as
retrieve (DEPT.all) where DEPT.floor = $.floor
```

This function is defined for each instance of DEPT and its value is the result of the query with the appropriate value substituted for $.floor. Like C functions that have a class as an argument, such POSTQUEL functions can either be thought of as functions and queried as follows:

```
retrieve (DEPT.name) where neighbors(DEPT).name = "shoe"
```

or they can be thought of as new attributes using the following query syntax:

```
retrieve (DEPT.name) where DEPT.neighbors.name = "shoe"
```

## 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, array support, and path expressions which were illustrated earlier, the features which have been added to a traditional relational language include:

support for nested queries
transitive closure
support for inheritance
support for time travel

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 NOT-IN {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 NOT-IN 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 class:

parent (older, younger)

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

retrieve* into answer (parent.older) from 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. As noted in this example, the result of a POSTQUEL command can be added to the data base as a new class. 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.

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

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

Here the * after EMP indicates that the query should be run over EMP and all classes under EMP in the inheritance hierarchy. This use of * allows a user to easily run queries over a class 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) from 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. Section 4 discusses support for this feature in more detail.

## 2.4. Fast Path

There are two reasons why we chose to implement a **fast path** feature. First, 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 interface to the POSTGRES optimizer or executor. Most DBMSs do not allow direct access to internal system modules.

The second reason is a bit more complex. In the Berkeley implementation of persistent CLOS, it is necessary for the run time system to assign a unique identifier (OID) to every persistent object it constructs. 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 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, a user can ask that any function known to POSTGRES be executed. This function can be one that a user has previously defined or it can be one that is included in the POSTGRES implementation. Hence, a 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.

The above capability is called **fast path** because it provides 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.

## 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. However, there are a large number of more general rules which an application designer would want to support. For example, one might want to insist that a specific employee, Joe, has the same salary as another employee, Fred.

This rule is very difficult to enforce in application logic because it would require the application to see all updates to the salary field, in order to **fire** application logic to enforce the rule at the correct time. A better solution is to enforce the rule inside the data manager.

In addition, most current systems have special purpose rules systems to support relational views, and protection. In building the POSTGRES rules system we were motivated by the desire to construct **one** general purpose rules system that could perform **all** of the following functions:

view management
triggers
integrity constraints
referential integrity
protection
version control

This should be contrasted with other approaches e.g. [ESWA76, MCCA89, WIDO90] which have different goals.

## 3.2. POSTGRES Rules

The rules we are using have a familiar production rule syntax of the form:

ON event (TO) object WHERE POSTQUEL-qualification
THEN DO [instead] 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 class or class.column. POSTQUEL-qualification is a normal qualification, with no additions or changes. The optional keyword **instead** indicates that the action indicated by POSTQUEL-command(s) is to be performed instead of the action which caused the rule to activate. If **instead** is missing, then the action is done in addition to the user event. Lastly, POSTQUEL-commands is a set of POSTQUEL commands with the following two changes:

new or current can appear instead of the name of a class in front of any attribute.
refuse (target-list) is added as a new POSTQUEL command

In this notation we could specify that Fred's salary adjustments get propagated on to Joe as follows:

on new EMP.salary where EMP.name = "Fred"
then do replace E (salary = new.salary) from E in EMP where E.name = "Joe"

In general, rules specify additional actions to be taken as a result of user updates. These additional actions may activate other rules, and a **forward chaining** control flow results, as was popularized in OPS5 [FORG81].

POSTGRES allows events to be retrieves as well as updates. Moreover, the action can be one or more queries. Consequently, the rule that Joe must have the same salary as Fred can also be expressed as:

on retrieve to EMP.salary where EMP.name = "Joe"
then do instead retrieve (EMP.salary) where EMP.name = "Fred"

In this case, Joe's salary is not explicitly stored; rather it is **derived** by activating the above rule. In this case the two data items are kept in synchronization by storing one and deriving the other. Moreover, if Fred's salary is not explicitly stored, then further rules would be awakened to find the ultimate answer, and a **backward chaining** control flow results. This control structure was popularized in Prolog [CLOC81].

If Fred receives frequent raises and Joe's salary is rarely queried, then the backward chaining representation will be more efficient. On the other hand, if many queries are directed to Joe's salary and Fred is rarely updated, then the forward chaining alternative is preferred. In POSTGRES, the application designer must decide whether he desires a forward chaining or backward chaining control flow and specify his rules accordingly.

## 3.3. Implementation of Rules

There are two implementations for POSTGRES rules. The first is through **record level** processing deep in the run-time system. This rules system is called when individual records are accessed, deleted, inserted or modified. The second implementation is through a **query rewrite** module. This code exists between the parser and the query optimizer and converts a user command to an alternate form prior to optimization. In the rest of this section we briefly discuss each implementation by explaining how each system processes the rule which progagates Fred's salary on to Joe, i.e:

on new EMP.salary where EMP.name = "Fred"
then do replace E (salary = new.salary) from E in EMP where E.name = "Joe"

The record-level rule system causes a marker to be placed on the salary attribute of Fred's instance. This marker contain the identifier of the corresponding rule and the types of events to which it is sensitive. If the executor touches a marked attribute, then it calls the rule system before proceeding. The rule system is passed the current instance and the proposed new one. It discovers that the event of the rule actually applies, substitutes new values and current values in the action part of the rule and then executes the action. When the action is complete, it returns control to the executor which installs the proposed update and continues.

If Fred's name is changed, then the marker on his salary must be dropped. In addition, if Joe is hired before Fred, then the markers must be added at the time Fred's record is inserted into the DBMS. To perform these tasks POSTGRES requires other markers which are discussed in [STON90B]. Also, if a rule sets a sufficient number of markers in a class, then POSTGRES can perform marker **escalation** and place an enclosing marker on the entire class. Again details appear in [STON90B].

The record-level rules system is especially efficient if there are a large number of rules, and each covers only a few instances. In this case, no extra overhead will be required unless a marked instance is actually touched. Hence, the rule systems requires no "tax", unless a rule actually applies. In this case, the overhead is that required to ensure the event is true and then to execute the action.

On the other hand, consider the following rule:

```
on replace to EMP.salary
then do
    append to AUDIT (name = current.name, salary = current.salary, new = new.salary, user = user())
```

and an incoming query:

```
replace EMP (salary = 1.1 * EMP.salary) where EMP.age < 50
```

Clearly, utilizing the record-level rules system will entail firing this rule once per elderly employee, a large overhead. It is much more efficient to **rewrite** the user command to:

```
append to AUDIT (name = EMP.name, salary = EMP.salary, new = 1.1 * EMP.salary, user = user())
   where EMP.age < 50
replace EMP (salary = 1.1 * EMP.salary) where EMP.age < 50
```

In this case, the auditing operation is done in bulk as a single command. In [STON90B] we present a general algorithm which can rewrite any POSTGRES command to enforce any rule. In general, if there are N rules for a given class, then each user command will turn into a total of N + 1 resulting commands. Therefore, this rules system will perform poorly if there are a large number of small scope rules but admirably if there are a small number of large scope rules.

As a result the two implementations are complementary, and we are exploring a **rule chooser** which could suggest the best implementation for any given rule. Unfortunately the two implementations have different semantics in certain cases, and we now turn to this topic.

## 3.4. Semantics of Rules

Consider the rule

```
on retrieve to EMP.salary where EMP.name = "Joe"
then do instead retrieve (EMP.salary) where EMP.name = "Fred"
```

and the following user query:

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

If query rewrite is used to support the above rule, then the user query will be rewritten to:

retrieve (EMP.name, E.salary) from E in EMP where EMP.name = "Joe" and E.name = "Fred"

Consider the possible answers to the user query for various numbers of instances of Fred. If there is no Fred in the data base, then the query rewritten by the rules system will return no instances. If there is one Fred, then one instance will be returned, while N Freds will cause N instances to be returned. Therefore, query rewrite implements the union semantics indicated in column 1 of Table 1. On the other hand, the record-level implementation can return Joe with a null salary if Fred doesn't exist. If there are multiple Freds, it can return any one of them, all of them, or an error. Therefore, it can implement union, random or error semantics.

Two conclusions are evident from this discussion. First, the desired semantics for this example are debatable. Moreover, a case can probably be made for each of the semantics, depending on the attribute whose value is provided by the rule. Hence, one should probably include all three, so that an informed user can choose which one fits his application. Second, it is infeasible for the query rewrite system to produce anything other than union semantics. Therefore, a user who desires different semantics must choose the record-level system. As a result, the selection of which rule system to use has semantic as well as performance implications.

A separate semantic matter concerns the time that rules are activated. There are certain rules which must be activated **immediately** upon occurence of the event in the rule, and others which should be

| | Union Semantics | Random Semantics | Error Semantics |
|---|---|---|---|
| no Fred | 0 instances | 1 instance with a null salary | 1 instance with a null salary |
| 1 Fred | 1 instance | 1 instance | 1 instance |
| N Freds | N instances | 1 instance | error |

Semantics of Joe's salary

Table 1

**deferred** to the end of the user's transaction. Also, some rules should be run as **part of** the user's transaction, while others should run in a **separate** transaction. For example, the following rule must run immediately in the same transaction:

on retrieve to EMP.salary where EMP.name = "Joe"
then do instead retrieve (EMP.salary) where EMP.name = "Fred"

while the one below must be activated **immediately** in a **different** transaction,

on retrieve to EMP.salary
then do append to AUDIT (name = current.name, salary = current.salary, user = user())

In this last example, the user can abort after he has retrieved salary data of interest. If the action is run in the user's transaction, then aborting will subvert the desired auditing. In addition, the action must be performed immediately for the same reason.

As a result there are at least four reasonable rule activation policies:

immediate -- same transaction
immediate -- different transaction
deferred -- same transaction
deferred -- different transaction

At the moment, POSTGRES only implements the first option. In time, we may support all four.

## 3.5. Rule System Applications

In this section we discuss the implementation of POSTGRES views and versions. In both cases, required functionality is supported by **compiling** user level syntax into one or more rules for subsequent activation inside POSTGRES.

Views (or virtual classes) are an important DBMS concept because they allow previously implemented classes to be supported even when the schema changes. For example, the view, TOY-EMP, can be defined as follows:

define view TOY-EMP (EMP.all) where EMP.dept = "toy"

This view is complied into the following POSTGRES rule:

on retrieve to TOY-EMP
then do instead retrieve (EMP.all) where EMP.dept = "toy"

Any query ranging over TOY-EMP will be processed correctly by either implementation of the POSTGRES rules system. However, a key problem is supporting updates on views. Current commercial relational systems support only a subset of SQL update commands, namely those which can be unambiguously processed against the underlying base tables. POSTGRES takes a much more general approach. If the application designer specifies a default view, i.e:

define default view LOW-PAY (EMP.OID, EMP.name, EMP.age) where EMP.salary < 5000

then, a collection of default update rules will be compiled for the view. For example, the replace rule for LOW-PAY is:

on replace to LOW-PAY.age
then do instead replace EMP (age = new.age) where EMP.OID = current.OID

These default rules will give the correct view updating semantics as long as the view has no ambiguous updates. However, the application designer is free to specify his own update semantics by indicating other update rules. For example, he could define the following replace rule for TOY-EMP

on replace to TOY-EMP.dept
then do instead delete EMP where EMP.name = current.name and new.dept != "toy"

Therefore, default views are supported by compiling the view syntax into a collection of rules. Other update semantics can be readily specified by user-written updating rules.

A second area where compilation to rules can support desired functionality is that of **versions.** The goal is to create a **hypothetical** version of a class with the following properties:

1) Initially the hypothetical class has all instances of the base class
2) The hypothetical class can then be freely updated to diverge from the base class
3) Updates to the hypothetical class do not cause physical modifications to the base class
3) Updates to the base class are visible in the hypothetical class, unless the instance updated has been deleted or modified in the hypothetical class.

Of course, it is possible to support versions by making a complete copy of the class for the version and then making subsequent updates in the copy. More efficient algorithms which make use of **differential** files are presented in [KATZ82, WOOD83].

In POSTGRES any user can create a version of a class as follows:

create version my-EMP from EMP

This command is supported by creating two **differential** classes for EMP:

EMP-MINUS (deleted-OID)
EMP-PLUS (all-fields-in EMP, replaced-OID)

and installing a collection of rules. EMP-MINUS holds the OID for any instance in EMP which is to be deleted from the version, and is the negative differential. On the other hand, EMP-PLUS holds any new instances added to the version as well as the new record for any modification to an instance of EMP. In the latter case, the OID of the record replaced in EMP is also recorded.

The retrieve rule installed at the time the version is created is:

```
on retrieve to my-EMP
then do instead
retrieve (EMP-PLUS.all)
retrieve (EMP.all) where EMP.OID NOT-IN {EMP-PLUS.replaced-OID}
        and EMP.OID NOT-IN {EMP-MINUS.deleted-OID}
```

The delete rule for the version is similarly:

```
on delete to my-EMP
then do instead
append to EMP-MINUS (deleted-OID = current.OID) where EMP.OID = current.OID
delete EMP-PLUS where EMP-PLUS.OID = current.OID
```

The interested reader can derive the replace and append rules or consult [ONG90] for a complete explanation. Also, there is a performance comparison in [ONG90] which shows that a rule system implementation of versions has comparable performance to an algorithmic implementation with hard-wired code deep in the executor.

Both of the examples in this section have shown important DBMS function that can be supported with very little code by compiling higher level syntax into a collection of rules. In addition, both examples are only possible with a rule system like POSTGRES that supports both forward and backward chaining rules.

## 4. STORAGE SYSTEM

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, instantaneous crash recovery and time travel. 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. Of course, the tradeoff is that a POSTGRES data base at any given time will have committed instances intermixed with instances which were written by aborted transactions. The run time system must distinguish these two kinds of instances and ignore the latter ones. The techniques used are discussed in [STON87].

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.

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. To support time travel, POSTGRES maintains two different physical collections of records, one for the current data and one for historical data, each with its own indexes. As noted in [STON87] there is an asynchronous demon, which we call the **vacuum cleaner,** running in the background which moves records which are no longer valid from the current data base to the historical data base. The historical data base is formatted to perform well on an archival device such as an optical disk jukebox. Further details can be obtained from [STON87].

## 5. THE POSTGRES IMPLEMENTATION

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

the process structure
extendability
dynamic loading
rule wake-up

and we discuss each in turn.

The first aspect of our design concerns the operating system process structure. Currently, POSTGRES runs as one process for each active user. Therefore, N active users will get N POSTGRES processes which share the POSTGRES code, buffer pool and lock table but have private data segments. This was done as an expedient to get a system operational as quickly as possible. Hence, we deliberately ducked the complexity associated with building POSTGRES as a single server process to which the N users can connect or as a collection of J, J < N, servers to which users connect. Either option would have required process management and scheduling to be built inside of POSTGRES, and we wanted to avoid these difficulties.

Second, POSTGRES extendability has been accomplished by making the parser, optimizer and execution engine entirely table-driven. For example, if the parser sees a token, ||, it checks in the operator class in the system catalogs to see if the operator is defined. If not, it generates an error. Information for frequently used operators is cached in a main memory data structure for augmented performance. When the optimizer evaluates a qualification, such as:

where EMP.location || "(0,0)"

it checks to see if there is an index on location and if so whether the operator || is supported for the index and what the selectivity of the clause is. With this information it can compute the expected cost of an indexed scan and compare it with a sequential scan. The general algorithm is sketched in [STON86B]. Basically, the optimizer is table-driven off the system catalogs, which describe the present storage configuration.

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. The downside of this design decision is that a dynamic loader is required for each hardware platform on which POSTGRES operates.

Lastly, the record-oriented implementation for rules system forces significant complexity on our design. A user can add a rule such as:

```
on new EMP.salary where EMP.name = "Joe"
then do retrieve (new.salary)
```

In this case his application process wishes to be notified of any salary adjustment for 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 and alerts a special process called the **POSTMASTER.** This process in turn alerts the process for the first user where the query would be run and the results delivered to the application process.

## 6. POSTGRES PERFORMANCE

At the current time (June 1991) POSTGRES Version 2.1 has been distributed for nearly three months and has been installed by at least 125 sites. In this section we indicate POSTGRES, Version 2.1 performance on both the Wisconsin benchmark [BITT83] and on an engineering benchmark [CATT91]. For the Wisconsin benchmark, we compare POSTGRES with the University of California version of INGRES which we worked on from 1974-78. Figure 1 shows the performance of the two systems for a subset of the Wisconsin benchmark executing on a Sun SPARCstation. As can be seen, POSTGRES is around twice the speed of UCB-INGRES.

We have also compared the performance of POSTGRES with that of INGRES, Version 5.0, a commercial DBMS from the INGRES products division of ASK Computer Systems. On a SUN 3/280 POSTGRES is about 3/5 of the performance of ASK-INGRES for the Wisconsin benchmark. There are still substantial inefficiencies in POSTGRES, especially in the code which checks that a retrieved record is valid. We expect that subsequent tuning planned for Version 3 will get us somewhere closer to ASK-INGRES.

As a second benchmark, we report the performance of POSTGRES on the benchmark in [CATT91]. In this benchmark, we compare POSTGRES with the systems reported by Cattell, namely his in-house system, an OODB from one of the commercial vendors, and a commercial RDBMS. In Figure 2 we report results for three configurations of the small data base version of the benchmark, using POSTGRES configured with 7.0 Mbytes of buffer space. The first two describe a remote data base configuration in which the

| Query | Meaning | POSTGRES | UCB-INGRES |
|---|---|---|---|
| 2 | select 10% into temp, no index | 9.8 | 10.2 |
| 3 | select 1% into temp, clust. index | 0.7 | 5.2 |
| 5 | select 1% into temp, non-clust. index | 1.2 | 5.3 |
| 6 | select 10% into temp, non-clust. index | 4.0 | 8.9 |
| 7 | select 1 to screen, clust. index | 0.3 | 0.9 |
| 9 | joinAselB, no index | 12.6 | 35.3 |
| 10 | joinABprime, no index | 17.0 | 35.3 |
| 11 | joinCselAselB, no index | 25.9 | 53.7 |
| 14 | joinCselAselB, clust. index | 24.1 | 56.7 |
| 17 | joinCselAselB, non-clust. index | 35.2 | 68.7 |
| 18 | project 1% into temp | 18.5 | 36.7 |

A Comparison of UCB-INGRES and POSTGRES

(Times are listed in seconds per query.)

Figure 1

data base resides on a SUN 3/280 and the application program executes on a separate SUN 3/60, and we indicate respectively "cold" (1st execution of the command) and "warm" (after cache stabilizes) numbers. The third set of results describes a "local" configuration for which both the application program and the data base reside on the same SUN 3/280. "Warm-local" numbers are omitted because they are essentially idential to the "warm-remote" results.

The numbers for the other systems were reported in [CATT91] running on a different Sun 3/280. Because the disk on the Cattell system is dramatically faster than the the disk on the POSTGRES system, the comparison is not "apples to apples". As a result, we also report "cooked" POSTGRES numbers, obtained by multiplying the POSTGRES I/O time by the ratio of the average seek times of the two disks and making the appropriate adjustment. The cooked numbers are our best guess for POSTGRES performance on the Cattell hardware.

To make POSTGRES perform as well as possible, we wrote all three benchmark routines as a C functions which are executed using the Fast Path feature of POSTGRES described in Section 2.4. These functions make appropriate calls directly on the POSTGRES access methods to manipulate the data base. This is a high performance way of using POSTGRES, but of course, provides no data independence whatsoever.

As can be seen, POSTGRES beats the relational system by a substantial factor. Relative to the in-house system POSTGRES loses by about a factor of 2. Since the two systems are executing similar algorithms; the difference is mostly accounted for by tuning considerations. For example, on the warm-lookup both systems are measuring the CPU time to perform 1000 access method calls. Since the POSTGRES access methods are slower than the in-house system, we obviously has some tuning left to do. The second consideration concerns data base size. POSTGRES puts a large header on the front of each record, and incurs a substantial space penalty because record size is so small on this benchmark. Hence, on the cold-lookup benchmark, POSTGRES will bring into the cache about twice as many pages as the in-house system. Obviously, we must optimize the size of the headers to be competitive on small record benchmarks. We expect that subsequent tuning of this sort will improve POSTGRES performance to approximate that of the in-house system.

The OODB system is faster than both the in-house system and POSTGRES on the insert operation because it clusters different record types on the same disk page. This allows it to do less I/O for the insert than the other two systems. It also outperforms the other systems on "warm" operations because it caches records in main memory format rather than disk format.

Two comments should be made at this point. First, POSTGRES allows an application designer to tradeoff performance for data independence and other DBMS services. He can code the benchmark for maximum performance and no data independence as we did above. Alternately, he can use the query language and obtain lower performance with full DBMS services. Hence, POSTGRES allows the application designer to choose the right mix of performance and data base services appropriate for his application.

A second comment is that the in-house and OODB systems run the data base in the same address space as the user program. Consequently, a malicious or careless user can obliterate the data base and compromise DBMS security. On the other hand, POSTGRES imports only specific user functions into its address space. Although such functions can be malicious or careless and cause data loss, POSTGRES is trusting only indicated functions and not whole user programs. Moreover, POSTGRES provides a registration facility for functions, at which point they can be scrutinized for security. Therefore, POSTGRES provides a higher degree of data security than available from the other systems. Of course, POSTGRES must import all routines that the indicated collection of functions makes calls on, which could be the entire

| Query | in-house | OODB | RDBMS | POSTGRES | POSTGRES-cooked |
|---|---|---|---|---|---|
| cold-remote-lookup | 7.6 | 20 | 29 | 24.2 | 17.5 |
| cold-remote-traversal | 17 | 17 | 90 | 44.1 | 36.8 |
| cold-remote-insert | 8.2 | 3.6 | 20 | 9.5 | 7.3 |
| warm-remote-lookup | 2.4 | 1.0 | 19 | 8.4 | 8.4 |
| warm-remote-traversal | 8.4 | 1.2 | 84 | 26.8 | 26.8 |
| warm-remote-insert | 7.5 | 2.9 | 20 | 5.4 | 4.5 |
| cold-local-lookup | 5.4 | 13 | 27 | 24.1 | 17.4 |
| cold-local-traversal | 13 | 9.8 | 90 | 44.0 | 36.7 |
| cold-local-insert | 7.4 | 1.5 | 22 | 9.5 | 7.3 |

A Comparison of Several Systems on the Cattell Benchmark

Figure 2

application in the worst case.

## 7. CONCLUSIONS

This paper has presented the design, implementation and some of the philosophy of POSTGRES. We feel that it meets most of the "litmus test" presented in [COMM90]; hence, POSTGRES capabilities may serve as a beacon for future evolution of commercial systems.

We expect to produce Version 3 of POSTGRES which should be available in the third quarter of 1991. It will be as fast and bug-free as possible, and contain the complete implementation of aggregates and complex objects. At that time, we will have implemented the entire proposed system with the exception of:

1) Union, intersection and other set functions have not been constructed. The only set functions available are IN and NOT-IN.
2) A where clause cannot appear inside the {...} notation

We are starting to design the successor to POSTGRES, temporarily designated POSTGRES II, which will attempt to manage main-memory data, disk-based data, and archive-based data in an elegant, unified manner. A first look at our ideas appears in [STON91].

**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.

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

[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.

[CATT91] Cattell, R., "An Engineering Database Benchmark," in J. Gray, Ed, The Benchmark Handbook for Database and Transaction Processing Systems," Morgan-Kaufmann, San Mateo, Ca., 1991.

[CLOC81] Clocksin, W. and Mellish, C., “Programming in Prolog,” Springer-Verlag, Berlin, Germany, 1981.

[COMM90] Committee for Advanced DBMS Function, "Third Generation Database System Manifesto, SIGMOD Record, Sept. 1990.

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

[DEUX90] Deux, O. et. al., "The Story of O2," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[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.

[FORG81] Forgy, C., "The OPS5 User’s Manual," Carneigie Mellon Univ., Technical Report, 1981.

[KATZ82] Katz, R. and Lehman, T., "Storage Structures for Versions and Alternatives," Computer Science Dept., University of Wisconsin, Madison, Wisc., Report 479, July 1982.

[KEMN91] Kemnitz, G. and Stonebraker, M., "The POSTGRES Tutorial," Electronics Research Laboratory, Memorandum M91/82, February 1991.

[KEMN91B] Kemnitz, G. (ed.), “The POSTGRES Reference Manual, Version 2.1” Electronics Research Laboratory, University of California, Berkeley, CA, Report M91/10, February 1991.

[KIM90] Kim, W. et. al., "Architecture of the ORION Next-Generation Database System," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[MCCA89] McCarthy, D. and Dayal, U., "Architecture of an Active Database System," Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[ONG90] Ong, L. and Goh, J., "A Unified Framework for Version Modeling Using Production Rules in a Database System", University of California, Electronics Research Laboratory, Memo UCB/ERL M90/33, April 1990.

[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.

[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.

[STON90] Stonebraker, M. et. al., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[STON90B] Stonebraker, M. et. al., ‘‘On Rules, Procedures Caching and Views,’’ Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[STON91] Stonebraker, M., "Managing Persistent Objects in a Multi-level Store," Proc. 1991 ACM-SIGMOD Conference on Management of Data, Denver, Co., May 1991. Research Laboratory, Memorandum M91/72, February 1991.

[WANG88] Wang, Y., "The PICASSO Shared Object Hierarchy," MS Report, University of California, Berkeley, June 1988.

[WIDO90] Widom, J. and Finkelstein, S., "Set-Oriented Production Rules in Relational Database Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[WILK90] Wilkinson, K. et. al., "The IRIS Architecture and Implementation," IEEE Transactions on Knowledge and Data Engineering, March 1990.

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