THE POSTGRES PAPERS

by

Michael Stonebraker and Lawrence A. Rowe
(editors)

Memorandum No. UCB/ERL M86/85

25 June 1987
(Revised)

THE POSTGRES PAPERS

by

Michael Stonebraker and Lawrence A. Rowe
(editors)

Memorandum No. UCB/ERL M86/85

25 June 1987
(Revised)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE POSTGRES PAPERS

by

Michael Stonebraker and Lawrence A. Rowe
(editors)

Memorandum No. UCB/ERL M86/85

25 June 1987
(Revised)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

FORWARD

This collection of papers reports on the design of the POSTGRES data base management system. The first paper entitled "The Design of POSTGRES" gives an overview of the system and the various capabilities that were planned. It also describes the design as it was envisioned in early 1986. Then, three papers are included that describe aspects of the system in more detail. The first one indicates the data model and query language that is being implemented. This paper was written in March 1987, and the reader should note that the data model evolved somewhat during the intervening year to include the notion of inheritance and a set-oriented query language construct. The next paper describes the rules system that is actually being implemented. In spirit it is the same as presented in the original POSTGRES paper; however, the syntax has been changed to be somewhat cleaner and the lock manager that supports the implementation is considerably more complex. Lastly, a paper on the design of the storage manager is included. This paper elaborates on the sketchy design presented in the the 1986 paper, but contains no fundamental changes.

Lastly, this collection of papers concludes with a revised version of the POSTGRES programming environment that is being constructed by Lawrence Rowe and a collection of students. This system, Object-FADS, is a user-extendible object-oriented system that will allow the construction of user interface programs for POSTGRES. The paper presents the environment as well as the techniques that will support efficient execution of Object-FADS programs.

At the current time (July 1987) large pieces of POSTGRES are operational. The parser, optimizer and communication routines are complete. The run-time system is still primitive, but all POSTGRES function except support for rules and procedures is present. The storage manager for the magnetic disk system is working, while the archive system is in the early phase of coding. Lastly, the abstract data type system and extendible index capability is fully operational. We expect to have a complete prototype by January, 1988 that could be used by others for useful work.

The system is being constructed under the direction of Jeff Anton who is our full-time chief programmer by a team consisting of Philip Chang, Steven Grady, Serge Granik, Mike Hirohama, Spyros Potamianos and Yongdong Wang.

This research was sponsored by the National Science Foundation under grant DMC-8504633 and by the Navy Electronics Systems Command under contract NOOO39-84-C-0089.

Table of Contents

THE DESIGN OF POSTGRES

Michael Stonebraker and Lawrence A. Rowe

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

Abstract

This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system. The main design goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferencing including forward- and backward-chaining,

4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model.

The paper describes the query language, programming langauge interface, system architecture, query processing strategy, and storage system for the new system.

1. INTRODUCTION

The INGRES relational database management system (DBMS) was implemented during 1975-1977 at the Univerisity of California. Since 1978 various prototype extensions have been made to support distributed databases [STON83a], ordered relations [STON83b], abstract data types [STON83c], and QUEL as a data type [STON84a]. In addition, we proposed but never prototyped a new application program interface [STON84b]. The University of California version of INGRES has been "hacked up enough" to make the inclusion of substantial new function extremely difficult. Another problem with continuing to extend the existing system is that many of our proposed ideas would be difficult to integrate into that system because of earlier design decisions. Consequently, we are building a new database system, called POSTGRES (POST inGRES).

This paper describes the design rationale, the features of POSTGRES, and our proposed implementation for the system. The next section discusses the design goals for the system. Sections 3 and 4 presents the query language and programming language interface, respectively, to the system. Section 5 describes the system architecture including the process structure, query processing strategies, and storage system.

2. DISCUSSION OF DESIGN GOALS

The relational data model has proven very successful at solving most business data processing problems. Many commercial systems are being marketed that are based on the relational model and in time these systems will replace older technology DBMS's. However, there are many engineering applications (e.g., CAD systems, programming environments, geographic data, and graphics) for which a conventional relational system is not suitable. We have embarked on the design and implementation of a new generation of DBMS's, based on the relational model, that will provide the facilities required by these applications. This section describes the major design goals for this new system.

The first goal is to support complex objects [LORI83, STON83c]. Engineering data, in contrast to business data, is more complex and dynamic. Although the required data types can be simulated on a relational system, the performance of the applications is unacceptable. Consider the following simple example. The objective is to store a collection of geographic objects in a database (e.g., polygons, lines, and circles). In a conventional relational DBMS, a relation for each type of object with appropriate fields would be created:

```
POLYGON (id, other fields)
CIRCLE (id, other fields)
LINE (id, other fields)
```

To display these objects on the screen would require additional information that represented display characteristics for each object (e.g., color, position, scaling factor, etc.). Because this information is the same for all objects, it can be stored in a single relation:

```
DISPLAY( color, position, scaling, obj-type, object-id)
```

The "object-id" field is the identifier of a tuple in a relation identified by the "obj-type" field (i.e., POLYGON, CIRCLE, or LINE). Given this representation, the following commands would have to be executed to produce a display:

```
foreach OBJ in {POLYGON, CIRCLE, LINE} do
    range of O is OBJ
    range of D is DISPLAY
    retrieve (D.all, O.all)
    where D.object-id = O.id
    and D.obj-type = OBJ
```

Unfortunately, this collection of commands will not be executed fast enough by any relational system to "paint the screen" in real time (i.e., one or two seconds). The problem is that regardless of how fast your DBMS is there are too many queries that have to be executed to fetch the data for the object. The feature that is needed is the ability to store the object in a field in DISPLAY so that only one

query is required to fetch it. Consequently, our first goal is to correct this deficiency.

The second goal for POSTGRES is to make it easier to extend the DBMS so that it can be used in new application domains. A conventional DBMS has a small set of built-in data types and access methods. Many applications require specialized data types (e.g., geometic data types for CAD/CAM or a latitude and longitude position data type for mapping applications). While these data types can be simulated on the built-in data types, the resulting queries are verbose and confusing and the performance can be poor. A simple example using boxes is presented elsewhere [STON86]. Such applications would be best served by the ability to add new data types and new operators to a DBMS. Moreover, B-trees are only appropriate for certain kinds of data, and new access methods are often required for some data types. For example, K-D-B trees [ROBI81] and R-trees [GUTM84] are appropriate access methods for point and polygon data, respectively.

Consequently, our second goal is to allow new data types, new operators and new access methods to be included in the DBMS. Moreover, it is crucial that they be implementable by non-experts which means easy-to-use interfaces should be preserved for any code that will be written by a user. Other researchers are pursuing a similar goal [DEWI85].

The third goal for POSTGRES is to support active databases and rules. Many applications are most easily programmed using alerters and triggers. For example, form-flow applications such as a bug reporting system require active forms that are passed from one user to another [TSIC82, ROWE82]. In a bug report application, the manager of the program maintenance group should be notified if a high priority bug that has been assigned to a programmer has not been fixed by a specified date. A database alerter is needed that will send a message to the manager calling his attention to the problem. Triggers can be used to propagate updates in the database to maintain consistency. For example, deleting a department tuple in the DEPT relation might trigger an update to delete all employees in that department in the EMP relation.

In addition, many expert system applications operate on data that is more easily described as rules rather than as data values. For example, the teaching load of professors in the EECS department can be described by the following rules:

1) The normal load is 8 contact hours per year

2) The scheduling officer gets a 25 percent reduction

3) The chairman does not have to teach

4) Faculty on research leave receive a reduction proportional to their leave fraction

5) Courses with less than 10 students generate credit at 0.1 contact hours per student

6) Courses with more than 50 students generate EXTRA contact hours at a rate of 0.01 per student in excess of 50

7) Faculty can have a credit balance or a deficit of up to 2 contact hours

These rules are subject to frequent change. The leave status, course assignments, and administrative assignments (e.g., chairman and scheduling officer) all change frequently. It would be most natural to store the above rules in a DBMS and then infer the actual teaching load of individual faculty rather than storing teaching load as ordinary data and then attempting to enforce the above rules by a collection of complex integrity constraints. Consequently, our third goal is to support alerters, triggers, and general rule processing.

The fourth goal for POSTGRES is to reduce the amount of code in the DBMS written to support crash recovery. Most DBMS's have a large amount of crash recovery code that is tricky to write, full of special cases, and very difficult to test and debug. Because one of our goals is to allow user-defined access methods, it is imperative that the model for crash recovery be as simple as possible and easily extendible. Our proposed approach is to treat the log as normal data managed by the DBMS which will simplify the recovery code and simultaneously provide support for access to the historical data.

Our next goal is to make use of new technologies whenever possible. Optical disks (even writable optical disks) are becoming available in the commercial marketplace. Although they have slower access characteristics, their price-performance and reliability may prove attractive. A system design that includes optical disks in the storage hierarchy will have an advantage. Another technology that we foresee is workstation-sized processors with several CPU's. We want to design POSTGRES in such way as to take advantage of these CPU resources. Lastly, a design that could utilize special purpose hardware effectively might make a convincing case for designing and implementing custom designed VLSI chips. Our fifth goal, then, is to investigate a design that can effectively utilize an optical disk, several tightly coupled processors and custom designed VLSI chips.

The last goal for POSTGRES is to make as few changes to the relational model as possible. First, many users in the business data processing world will become familiar with relational concepts and this framework should be preserved if possible. Second, we believe the original "spartan simplicity" argument made by Codd [CODD70] is as true today as in 1970. Lastly, there are many semantic data models but there does not appear to be a small model that will solve everyone's problem. For example, a generalization hierarchy will not solve the problem of structuring CAD data and the design models developed by the CAD community will not handle generalization hierarchies. Rather than building a system that is based on a large, complex data model, we believe a new system should be built on a small, simple model that is extendible. We believe that we can accomplish our goals while preserving the relational model. Other researchers are striving for similar goals but they are using different approaches [AFSA85, ATKI84, COPE84, DERR85, LORI83, LUM85]

The remainder of the paper describes the design of POSTGRES and the basic system architecture we propose to use to implement the system.

3. POSTQUEL

This section describes the query language supported by POSTGRES. The relational model as described in the original definition by Codd [CODD70] has been preserved. A database is composed of a collection of relations that contain

tuples with the same fields defined, and the values in a field have the same data type. The query language is based on the INGRES query language QUEL [HELD75]. Several extensions and changes have been made to QUEL so the new language is called POSTQUEL to distinguish it from the original language and other QUEL extensions described elsewhere [STON85a, KUNG84].

Most of QUEL is left intact. The following commands are included in POSTQUEL without any changes: Create Relation, Destroy Relation, Append, Delete, Replace, Retrieve, Retrieve into Result, Define View, Define Integrity, and Define Protection. The Modify command which specified the storage structure for a relation has been omitted because all relations are stored in a particular structure designed to support historical data. The Index command is retained so that other access paths to the data can be defined.

Although the basic structure of POSTQUEL is very similar to QUEL, numerous extensions have been made to support complex objects, user-defined data types and access methods, time varying data (i.e., versions, snapshots, and historical data), iteration queries, alerters, triggers, and rules. These changes are described in the subsections that follow.

3.1. Data Definition

The following built-in data types are provided;

1) integers,
2) floating point,
3) fixed length character strings,
4) unbounded varying length arrays of fixed types with an arbitrary number of dimensions,
5) POSTQUEL, and
6) procedure.

Scalar type fields (e.g., integer, floating point, and fixed length character strings) are referenced by the conventional dot notation (e.g., EMP.name).

Variable length arrays are provided for applications that need to store large homogenous sequences of data (e.g., signal processing data, image, or voice). Fields of this type are referenced in the standard way (e.g., EMP.picture[i] refers to the i-th element of the picture array). A special case of arrays is the text data type which is a one-dimensional array of characters. Note that arrays can be extended dynamically.

Fields of type POSTQUEL contain a sequence of data manipulation commands. They are referenced by the conventional dot notation. However, if a POSTQUEL field contains a retrieve command, the data specified by that command can be implicitly referenced by a multiple dot notation (e.g., EMP.hobbies.battingavg) as proposed elsewhere [STON84a] and first suggested by Zaniolo in GEM [ZANI83].

Fields of type procedure contain procedures written in a general purpose programming language with embedded data manipulation commands (e.g., EQUEL [ALLM76] or Rigel [ROWE79]). Fields of type procedure and POSTQUEL can be executed using the Execute command. Suppose we are given a relation with the following definition

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

in which the "hobbies" field is of type POSTQUEL. That is, "hobbies" contains queries that retrieve data about the employee's hobbies from other relations. The following command will execute the queries in that field:

```
execute (EMP.hobbies)
where EMP.name = "Smith"
```

The value returned by this command can be a sequence of tuples with varying types because the field can contain more than one retrieve command and different commands can return different types of records. Consequently, the programming language interface must provide facilities to determine the type of the returned records and to access the fields dynamically.

Fields of type POSTQUEL and procedure can be used to represent complex objects with shared subobjects and to support multiple representations of data. Examples are given in the next section on complex objects.

In addition to these built-in data types, user-defined data types can be defined using an interface similar to the one developed for ADT-INGRES [STON83c, STON86]. New data types and operators can be defined with the user-defined data type facility.

3.2. Complex Objects

This section describes how fields of type POSTQUEL and procedure can be used to represent shared complex objects and to support multiple representations of data.

Shared complex objects can be represented by a field of type POSTQUEL that contains a sequence of commands to retrieve data from other relations that represent the subobjects. For example, given the relations POLYGON, CIRCLE, and LINE defined above, an object relation can be defined that represents complex objects composed of polygons, circles, and lines. The definition of the object relation would be:

```
create OBJECT (name = char[10], obj = postquel)
```

The table in figure 1 shows sample values for this relation. The relation contains the description of two complex objects named "apple" and "orange." The object "apple" is composed of a polygon and a circle and the object "orange" is composed of a line and a polygon. Notice that both objects share the polygon with id equal to 10.

Multiple representations of data are useful for caching data in a data structure that is better suited to a particular use while still retaining the ease of access via a relational representation. Many examples of this use are found in database systems (e.g., main memory relation descriptors) and forms systems [ROWE85].

Name	OBJ
apple	retrieve (POLYGON.all) where POLYGON.id = 10 retrieve (CIRCLE.all) where CIRCLE.id = 40
orange	retrieve (LINE.all) where LINE.id = 17 retrieve (POLYGON.all) where POLYGON.id = 10

Figure 1. Example of an OBJECT relation.

Multiple representations can be supported by defining a procedure that translates one representation (e.g., a relational representation) to another representation (e.g., a display list suitable for a graphics display). The translation procedure is stored in the database. Continuing with our complex object example, the OBJECT relation would have an additional field, named "display," that would contain a procedure that creates a display list for an object stored in POLYGON, CIRCLE, and LINE:

```
create OBJECT(name=char[10], obj=postquel, display=cproc)
```

The value stored in the display field is a procedure written in C that queries the database to fetch the subobjects that make up the object and that creates the display list representation for the object.

This solution has two problems: the code is repeated in every OBJECT tuple and the C procedure replicates the queries stored in the object field to retrieve the subobjects. These problems can be solved by storing the procedure in a separate relation (i.e., normalizing the database design) and by passing the object to the procedure as an argument. The definition of the relation in which the procedures will be stored is:

```
create OBJPROC(name=char[12], proc=cproc)
append to OBJPROC(name="display-list", proc="...source code...")
```

Now, the entry in the display field for the "apple" object is

```
execute (OBJPROC.proc)
with ("apple")
where OBJPROC.name="display-list"
```

This command executes the procedure to create the alternative representation and passes to it the name of the object. Notice that the "display" field can be changed to a value of type POSTQUEL because we are not storing the procedure in OBJECT, only a command to execute the procedure. At this point, the procedure can execute a command to fetch the data. Because the procedure was passed the name of the object it can execute the following command to fetch its value:

execute (OBJECT.obj)
where OBJECT.name=*argument*

This solution is somewhat complex but it stores only one copy of the procedure's source code in the database and it stores only one copy of the commands to fetch the data that represents the object.

Fields of type POSTQUEL and procedure can be efficiently supported through a combination of compilation and precomputation described in sections 4 and 5.

3.3. Time Varying Data

POSTQUEL allows users to save and query historical data and versions [KATZ85, WOOD83]. By default, data in a relation is never deleted or updated. Conventional retrievals always access the current tuples in the relation. Historical data can be accessed by indicating the desired time when defining a tuple variable. For example, to access historical employee data a user writes

retrieve (E.all)
from E in EMP["7 January 1985"]

which retrieves all records for employees that worked for the company on 7 January 1985. The From-clause which is similar to the SQL mechanism to define tuple variables [ASTR76], replaces the QUEL Range command. The Range command was removed from the query language because it defined a tuple variable for the duration of the current user program. Because queries can be stored as the value of a field, the scope of tuple variable definitions must be constrained. The From-clause makes the scope of the definition the current query.

This bracket notation for accessing historical data implicitly defines a snapshot [ADIB80]. The implementation of queries that access this snapshot, described in detail in section 5, searches back through the history of the relation to find the appropriate tuples. The user can materialize the snapshot by executing a Retrieve-into command that will make a copy of the data in another relation.

Applications that do not want to save historical data can specify a cutoff point for a relation. Data that is older than the cutoff point is deleted from the database. Cutoff points are defined by the Discard command. The command

discard EMP before "1 week"

deletes data in the EMP relation that is more than 1 week old. The commands

discard EMP before "now"

and

discard EMP

retain only the current data in EMP.

It is also possible to write queries that reference data which is valid between two dates. The notation

relation-name[date1, date2]

specifies the relation containing all tuples that were in the relation at some time between date1 and date2. Either or both of these dates can be omitted to specify all data in the relation from the time it was created until a fixed date (i.e., relation-name[,date]), all data in the relation from a fixed date to the present (i.e.,

relation-name[date,]), or all data that was every in the relation (i.e., relation-name[ ]). For example, the query

```
retrieve (E.all)
from E in EMP[ ]
where E.name="Smith"
```

returns all information on employees named Smith who worked for the company at any time.

POSTQUEL has a three level memory hierarchy: 1) main memory, 2) secondary memory (magnetic disk), and 3) tertiary memory (optical disk). Current data is stored in secondary memory and historical data migrates to tertiary memory. However, users can query the data without having to know where the data is stored.

Finally, POSTGRES provides support for versions. A version can be created from a relation or a snapshot. Updates to a version do not modify the underlying relation and updates to the underlying relation will be visible through the version unless the value has been modified in the version. Versions are defined by the Newversion command. The command

```
newversion EMPTEST from EMP
```

creates a version named EMPTEST that is derived from the EMP relation. If the user wants to create a version that is not changed by subsequent updates to the underlying relation as in most source code control systems [TICH82], he can create a version off a snapshot.

A Merge command is provided that will merge the changes made in a version back into the underlying relation. An example of a Merge command is

```
merge EMPTEST into EMP
```

The Merge command will use a semi-automatic procedure to resolve updates to the underlying relation and the version that conflict [GARC84].

This section described POSTGRES support for time varying data. The strategy for implementing these features is described below in the section on system architecture.

3.4. Iteration Queries, Alerters, Triggers, and Rules

This section describes the POSTQUEL commands for specifying iterative execution of queries, alerters [BUNE79], triggers [ASTR76], and rules.

Iterative queries are requried to support transitive closure [GUTM84 KUNG84]. Iteration is specified by appending an asterisk ("*") to a command that should be repetitively executed. For example, to construct a relation that includes all people managed by someone either directly or indirectly a Retrieve*-into command is used. Suppose one is given an employee relation with a name and manager field:

```
create EMP(name=char[20],...,mgr=char[20],...)
```

The following query creates a relation that conatins all employees who work for Jones:

```
retrieve* into SUBORDINATES(E.name, E.mgr)
from E in EMP, S in SUBORDINATES
where E.name="Jones"
  or E.mgr=S.name
```

This command continues to execute the Retrieve-into command until there are no changes made to the SUBORDINATES relation.

The "*" modifier can be appended to any of the POSTQUEL data manipulation commands: Append, Delete, Execute, Replace, Retrieve, and Retrieve-into. Complex iterations, like the A-* heuristic search algorithm, can be specified using sequences of these iteration queries [STON85b].

Alerters and triggers are specified by adding the keyword "always" to a query. For example, an alerter is specified by a Retrieve command such as

```
retrieve always (EMP.all)
where EMP.name = "Bill"
```

This command returns data to the application program that issued it whenever Bill's employee record is changed.[1] A trigger is an update query (i.e., Append, Replace, or Delete command) with an "always" keyword. For example, the command

```
delete always DEPT
where count(EMP.name by DEPT.dname
          where EMP.dept = DEPT.dname) = 0
```

defines a trigger that will delete DEPT records for departments with no employees.

Iteration queries differ from alerters and triggers in that iteration queries run until they cease to have an effect while alerters and triggers run indefinitely. An efficient mechanism to awaken "always" commands is described in the system architecture section.

"Always" commands support a forward-chaining control structure in which an update wakes up a collection of alerters and triggers that can wake up other commands. This process terminates when no new commands are awakened. POSTGRES also provides support for a backward-chaining control structure.

The conventional approach to supporting inference is to extend the view mechanism (or something equivalent) with additional capabilities (e.g. [ULLM85, WONG84, JARK85]). The canonical example is the definition of the ANCESTOR relation based on a stored relation PARENT:

```
PARENT (parent-of, offspring)
```

Ancestor can then be defined by the following commands:

---

[1] Strictly speaking the data is returned to the program through a portal which is defined in section 4.

```
range of P is PARENT
range of A is ANCESTOR
define view  ANCESTOR (P.all)
define view* ANCESTOR (A.parent-of, P.offspring)
        where A.offspring = P.parent-of
```

Notice that the ANCESTOR view is defined by multiple commands that may involve recursion. A query such as:

```
retrieve (ANCESTOR. parent-of)
where ANCESTOR.offspring = "Bill"
```

is processed by extensions to a standard query modification algorithm [STON75] to generate a recursive command or a sequence of commands on stored relations. To support this mechanism, the query optimizer must be extended to handle these commands.

This approach works well when there are only a few commands which define a particular view and when the commands do not generate conflicting answers. This approach is less successful if either of these conditions is violated as in the following example:

```
define view DESK-EMP (EMP.all, desk = "steel") where EMP.age < 40
define view DESK-EMP (EMP.all, desk = "wood" where EMP.age >= 40
define view DESK-EMP (EMP.all, desk = "wood") where EMP.name = "hotshot"
define view DESK-EMP (EMP.all, desk = "steel") where EMP.name = "bigshot"
```

In this example, employees over 40 get a wood desk, those under 40 get a steel desk. However, "hotshot" and "bigshot" are exceptions to these rules. "Hotshot" is given a wood desk and "bigshot" is given a steel desk, regardless of their ages. In this case, the query:

```
retrieve (DESK-EMP.desk) where DESK-EMP.name = "bigshot"
```

will require 4 separate commands to be optimized and run. Moreover, both the second and the fourth definitions produce an answer to the query that is different. In the case that a larger number of view definitions is used in the specification of an object, then the important performance parameter will be isolating the view definitions which are actually useful. Moreover, when there are conflicting view definitions (e.g. the general rule and then exceptional cases), one requires a priority scheme to decide which of conflicting definitions to utilize. The scheme described below works well in such situations.

POSTGRES supports backward-chaining rules by virtual columns (i.e., columns for which no value is stored). Data in such columns is inferred on demand from rules and cannot be directly updated, except by adding or dropping rules. Rules are specified by adding the keyword "demand" to a query. Hence, for the DESK-EMP example, the EMP relation would have a virtual field, named "desk," that would be defined by four rules:

```
replace demand EMP (desk = "steel") where EMP.age < 40
replace demand EMP (desk = "wood" where EMP.age >= 40
replace demand EMP (desk = "wood") where EMP.name = "hotshot"
replace demand EMP (desk = "steel") where EMP.name = "bigshot"
```

The third and fourth commands would be defined at a higher priority than the first

and second. A query that accessed the desk field would cause the "demand" commands to be processed to determine the appropriate desk value for each EMP tuple retrieved.

This subsection has described a collection of facilities provided in POSTQUEL to support complex queries (e.g., iteration) and active databases (e.g., alerters, triggers, and rules). Efficient techniques for implementing these facilities are given in section 5.

4. PROGRAMMING LANGUAGE INTERFACE

This section describes the programming language interface (HITCHING POST) to POSTGRES. We had three objectives when designing the HITCHING POST and POSTGRES facilities. First, we wanted to design and implement a mechanism that would simplify the development of browsing style applications. Second, we wanted HITCHING POST to be powerful enough that all programs that need to access the database including the ad hoc terminal monitor and any preprocessors for embedded query languages could be written with the interface. And lastly, we wanted to provide facilities that would allow an application developer to tune the performance of his program (i.e., to trade flexibility and reliability for performance).

Any POSTQUEL command can be executed in a program. In addition, a mechanism, called a "portal," is provided that allows the program to retrieve data from the database. A portal is similar to a cursor [ASTR76], except that it allows random access to the data specified by the query and the program can fetch more than one record at a time. The portal mechanism described here is different than the one we previously designed [STON84b], but the goal is still the same. The following subsections describe the commands for defining portals and accessing data through them and the facilities for improving the performance of query execution (i.e., compilation and fast-path).

4.1. Portals

A portal is defined by a Retrieve-portal or Execute-portal command. For example, the following command defines a portal named P:

```
retrieve portal P(EMP.all)
where EMP.age < 40
```

This command is passed to the backend process which generates a query plan to fetch the data. The program can now issue commands to fetch data from the backend process to the frontend process or to change the "current position" of the portal. The portal can be thought of as a query plan in execution in the DBMS process and a buffer containing fetched data in the application process.

The program fetches data from the backend into the buffer by executing a Fetch command. For example, the command

```
fetch 20 into P
```

fetches the first twenty records in the portal into the frontend program. These records can be accessed by subscript and field references on P. For example, P[i] refers to the i-th record returned by the last Fetch command and P[i].name refers to the "name" field in the i-th record. Subsequent fetches replace the previously

fetched data in the frontend program buffer.

The concept of a portal is that the data in the buffer is the data currently being displayed by the browser. Commands entered by the user at the terminal are translated into database commands that change the data in the buffer which is then redisplayed. Suppose, for example, the user entered a command to scroll forward half a screen. This command would be translated by the frontend program (i.e., the browser) into a Move command followed by a Fetch command. The following two commands would fetch data into the buffer which when redisplayed would appear to scroll the data forward by one half screen:

```
move P forward 10
fetch 20 into P
```

The Move command repositions the "current position" to point to the 11-th tuple in the portal and the Fetch command fetches tuples 11 through 30 in the ordering established by executing the query plan. The "current position" of the portal is the first tuple returned by the last Fetch command. If Move commands have been executed since the last Fetch command, the "current position" is the first tuple that would be returned by a Fetch command if it were executed.

The Move command has other variations that simplify the implementation of other browsing commands. Variations exist that allow the portal postion to be moved forward or backward, to an absolute position, or to the first tuple that satisfies a predicate. For example, to scroll backwards one half screen, the following commands are issued:

```
move P backward 10
fetch 20 into P
```

In addition to keeping track of the "current position," the backend process also keeps track of the sequence number of the current tuple so that the program can move to an absolute position. For example, to scroll forward to the 63-rd tuple the program executes the command:

```
move P forward to 63
```

Lastly, a Move command is provided that will search forward or backward to the first tuple that satisfies a predicate as illustrated by the following command that moves forward to the first employee whose salary is greater than $25,000:

```
move P forward to salary > 25K
```

This command positions the portal on the first qualifying tuple. A Fetch command will fetch this tuple and the ones immediately following it which may not satisfy the predicate. To fetch only tuples that satisfy the predicate, the Fetch command is used as follows:

```
fetch 20 into P where salary > 25K
```

The backend process will continue to execute the query plan until 20 tuples have been found that satisfy the predicate or until the portal data is exhausted.

Portals differ significantly from cursors in the way data is updated. Once a cursor is positioned on a record, it can be modified or deleted (i.e., updated directly). Data in a portal cannot be updated directly. It is updated by Delete or Replace commands on the relations from which the portal data is taken. Suppose

the user entered commands to a browser that change Smith's salary. Assuming that Smith's record is already in the buffer, the browser would translate this request into the following sequence of commands:

```
replace EMP(salary=NewSalary)
where EMP.name = "Smith"
fetch 20 into P
```

The Replace command modifies Smith's tuple in the EMP relation and the Fetch command synchronizes the buffer in the browser with the data in the database. We chose this indirect approach to updating the data because it makes sense for the model of a portal as a query plan. In our previous formulation [STON84], a portal was treated as an ordered view and updates to the portal were treated as view updates. We believe both models are viable, although the query plan model requires less code to be written.

In addition to the Retrieve-portal command, portals can be defined by an Execute command. For example, suppose the EMP relation had a field of type POSTQUEL named "hobbies"

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

that contained commands to retrieve a person's hobbies from the following relations:

```
SOFTBALL (name, position, batting-avg)
COMPUTERS (name, isowner, brand, interest)
```

An application program can define a portal that will range over the tuples describing a person's hobbies as follows:

```
execute portal H(EMP.hobbies)
where EMP.name = "Smith"
```

This command defines a portal, named "H," that is bound to Smith's hobby records. Since a person can have several hobbies, represented by more than on Retrieve command in the "hobbies" field, the records in the buffer may have different types. Consequently, HITCHING POST must provide routines that allow the program to determine the number of fields, and the type, name, and value of each field in each record fetched into the buffer.

4.2. Compilation and Fast-Path

This subsection describes facilities to improve the performance of query execution. Two facilities are provided: query compilation and fast-path. Any POSTQUEL command, including portal commands, can take advantage of these facilities.

POSTGRES has a system catalog in which application programs can store queries that are to be compiled. The catalog is named "CODE" and has the following structure:

```
CODE(id, owner, command)
```

The "id" and "owner" fields form a unique identifier for each stored command. The "command" field holds the command that is to be compiled. Suppose the programmer of the relation browser described above wanted to compile the Replace

command that was used to update the employee's salary field. The program could append the command, with suitable parameters, to the CODE catalog as follows:

```
append to CODE(id=1, owner="browser",
        command="replace EMP(salary=$1) where EMP.name=$2")
```

"$1" and "$2" denote the arguments to the command. Now, to execute the Replace command that updates Smith's salary shown above, the program executes the following command:

```
execute (CODE.command)
with (NewSalary, "Smith")
where CODE.id=1 and CODE.owner="browser"
```

This command executes the Replace command after substituting the arguments.

Executing commands stored in the CODE catalog does not by itself make the command run any faster. However, a compilation demon is always executing that examines the entries in the CODE catalog in every database and compiles the queries. Assuming the compilation demon has compiled the Replace command in CODE, the query should run substantially faster because the time to parse and optimize the query is avoided. Section 5 describes a general purpose mechanism for invalidating compiled queries when the schema changes.

Compiled queries are faster than queries that are parsed and optimized at run-time but for some applications, even they are not fast enough. The problem is that the Execute command that invokes the compiled query still must be processed. Consequently, a fast-path facility is provided that avoids this overhead. In the Execute command above, the only variability is the argument list and the unique identifier that selects the query to be run. HITCHING POST has a run-time routine that allows this information to be passed to the backend in a binary format. For example, the following function call invokes the Replace command described above:

```
exec-fp(1, "browser", NewSalary, "Smith")
```

This function sends a message to the backend that includes only the information needed to determine where each value is located. The backend retrieves the compiled plan (possibly from the buffer pool), substitutes the parameters without type checking, and invokes the query plan. This path through the backend is hand-optimized to be very fast so the overhead to invoke a compiled query plan is minimal.

This subsection has described facilities that allow an application programmer to improve the performance of a program by compiling queries or by using a special fast-path facility.

5. SYSTEM ARCHITECTURE

This section describes how we propose to implement POSTGRES. The first subsection describes the process structure. The second subsection describes how query processing will be implemented, including fields of type POSTQUEL, procedure, and user-defined data type. The third subsection describes how alerters, triggers, and rules will be implemented. And finally, the fourth subsection describes the storage system for implementing time varying data.

5.1. Process Structure

DBMS code must run as a sparate process from the application programs that access the database in order to provide data protection. The process structure can use one DBMS process per application program (i.e., a process-per-user model [STON81]) or one DBMS process for all application programs (i.e., a server model). The server model has many performance benefits (e.g., sharing of open file descriptors and buffers and optimized task switching and message sending overhead) in a large machine environment in which high performance is critical. However, this approach requires that a fairly complete special-purpose operating system be built. In constrast, the process-per-user model is simpler to implement but will not perform as well on most conventional operating systems. We decided after much soul searching to implement POSTGRES using a process-per-user model architecture because of our limited programming resources. POSTGRES is an ambitious undertaking and we believe the additional complexity introduced by the server architecture was not worth the additional risk of not getting the system running. Our current plan then is to implement POSTGRES as a process-per-user model on Unix 4.3 BSD.

The process structure for POSTGRES is shown in figure 3. The POSTMASTER will contain the lock manager (since there are no shared segments in 4.3 BSD) and will control the demons that will perform various database services (such as asynchronously compiling user commands). There will be one POSTMASTER process per machine, and it will be started at "sysgen" time.

The POSTGRES run-time system executes commands on behalf of one application program. However, a program can have several commands executing at the same time. The message protocol between the program and backend will use a simple request-answer model. The request message will have a command designator and a sequence of bytes that contain the arguments. The answer message format will include a response code and any other data requested by the command.

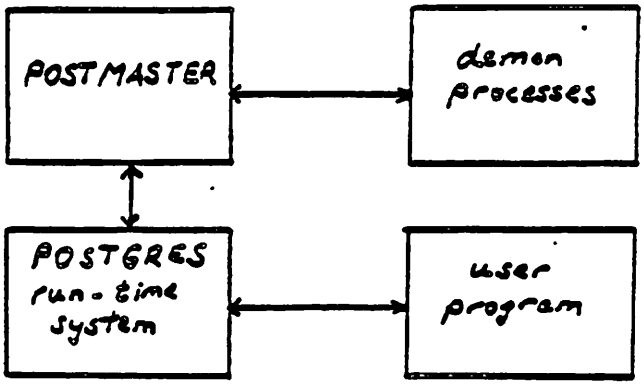

Figure 3. POSTGRES process structure.

Notice that in contrast to INGRES [STON76] the backend will not "load up" the communication channel with data. The frontend requests a bounded amount of data with each command.

5.2. Query Processing

This section describes the query processing strategies that will be implemented in POSTGRES. We plan to implement a conventional query optimizer. However, three extensions are required to support POSTQUEL. First, the query optimizer must be able to take advantage of user-defined access methods. Second, a general-purpose, efficient mechanism is needed to support fields of type POSTQUEL and procedure. And third, an efficient mechanism is required to support triggers and rules. This section describes our proposed implementation of these mechanisms.

5.2.1. Support for New Types

As noted elsewhere [STON86], existing access methods must be usable for new data types, new access methods must be definable, and query processing heuristics must be able to optimize plans for which new data types and new access methods are present. The basic idea is that an access method can support fast access for a specific collection of operators. In the case of B-trees, these operators are {<, =, >, >=, <=}. Moreover, these operators obey a collection of rules. Again for B-trees, the rules obeyed by the above set of operators is:

P1) key-1 < key-2 and key-2 < key-3 then key-1 < key-3
P2) key-1 < key-2 implies not key-2 < key-1
P3) key-1 < key-2 or key-2 < key-1 or key-1 = key-2
P4) key-1 <= key-2 if key-1 < key-2 or key-1 = key-2
P5) key-1 = key-2 implies key-2 = key-1
P6) key-1 > key-2 if key-2 < key-1
P7) key-1 >= key-2 if key-2 <= key-1

A B-tree access method will work for any collection of operators that obey the above rules. The protocol for defining new operators will be similar to the one described for ADT-INGRES [STON83c]. Then, a user need simply declare the collection of operators that are to be utilized when he builds an index, and a detailed syntax is presented in [STON86].

In addition, the query optimizer must be told the performance of the various access paths. Following [SELI79], the required information will be the number of pages touched and the number of tuples examined when processing a clause of the form:

relation.column OPR value

These two values can be included with the definition of each operator, OPR. The other information required is the join selectivity for each operator that can participate in a join, and what join processing strategies are feasible. In particular, nested iteration is always a feasible strategy, however both merge-join and hash-join work only in restrictive cases. For each operator, the optimizer must know whether merge-join is usable and, if so, what operator to use to sort each relation, and whether hash-join is usable. Our proposed protocol includes this information

with the definition of each operator.

Consequently, a table-driven query optimizer will be implemented. Whenever a user defines new operators, the necessary information for the optimizer will be placed in the system catalogs which can be accessed by the optimzier. For further details, the reader is refered elsewhere [STON86].

5.2.2. Support for Procedural Data

The main performance tactic which we will utilize is precomputing and caching the result of procedural data. This precomputation has two steps:

1) compiling an access plan for POSTQUEL commands
2) executing the access plan to produce the answer

When a collection of POSTQUEL commands is executed both of the above steps must be performed. Current systems drop the answer on the floor after obtaining it, and have special code to invalidate and recompute access plans (e.g. [ASTR76]). On the other hand, we expect to cache both the plan and the answer. For small answers, we expect to place the cached value in the field itself. For larger answers, we expect to put the answer in a relation created for the purpose and then put the name of the relation in the field itself where it will serve the role of a pointer.

Moreover, we expect to have a demon which will run in background mode and compile plans utilizing otherwise idle time or idle processors. Whenever a value of type procedure is inserted into the database, the run-time system will also insert the identity of the user submitting the command. Compilation entails checking the protection status of the command, and this will be done on behalf of the submitting user. Whenever, a procedural field is executed, the run-time system will ensure that the user is authorized to do so. In the case of "fast-path," the run-time system will require that the executing user and defining user are the same, so no run-time access to the system catalogs is required. This same demon will also precompute answers. In the most fortunate of cases, access to procedural data is instantaneous because the value of the procedure is cached. In most cases, a previous access plan should be valid sparing the overhead of this step.

Both the compiled plan and the answer must be invalidated if necessary. The plan must be invalidated if the schema changes inappropriately, while the answer must be invalidated if data that it accesses has been changed. We now show that this invalidation can be efficiently supported by an extended form of locks. In a recent paper [STON85c] we have analyzed other alternate implementations which can support needed capabilities, and the one we will now present was found to be attractive in many situations.

We propose to support a new kind of lock, called an I lock. The compatibility matrix for I locks is shown in figure 4. When a command is compiled or the answer precomputed, POSTGRES will set I locks on all database objects accessed during compilation or execution. These I locks must be persistent (i.e. survive crashes), of fine granularity (i.e. on tuples or even fields), escalatable to coarser granularity, and correctly detect "phantoms" [ESWA75]. In [STON85a], it is suggested that the best way to satisfy these goals is to place I locks in data records themselves.

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

Figure 4. Compatibility modes for I locks.

The * in the table in figure 4 indicates that a write lock placed on an object containing one or more I locks will simply cause the precomputed objects holding the I locks to be invalidated. Consequently, they are called "invalidate-me" locks. A user can issue a command:

retrieve (relation.I) where qualification

which will return the identifiers of commands having I locks on tuples in question. In this way a user can see the consequences of a proposed update.

Fields of type POSTQUEL can be compiled and POSTQUEL fields with no update statements can be precomputed. Fields of type procedure can be compiled and procedures that do not do input/output and do not update the database can be precomputed.

5.2.3. Alerters, Triggers, and Inference

This section describes the tactic we will use to implement alerters, triggers, and inference.

Alerters and triggers are specified by including the keyword "always" on the command. The proposed implementation of "always" commands is to run the command until it ceases to have an effect. Then, it should be run once more and another special kind of lock set on all objects which the commands will read or write. These T locks have the compatibility matrix shown in figure 5. Whenever a transaction writes a data object on which a T-lock has been set, the lock manager simply wakes-up the corresponding "always" command. Dormant "always" commands are stored in a system relation in a field of type POSTQUEL. As with I locks, T locks must be persistent, of fine granularity and escalatable. Moreover, the identity of commands holding T locks can be obtained through the special field, T added to all relations.

Recall that inferencing will be support by virtual fields (i.e., "demand" commands). "Demand" commands will be implemented similar to the way "always" commands are implemented. Each "demand" command would be run until the collection of objects which it proposes to write are isolated. Then a D lock is set on each such object and the command placed in a POSTQUEL field in the system catalogs. The compatibility matrix for D locks is shown in figure 6. The "&"

	R	W	I	T
R	ok	no	ok	ok
W	no	no	*	#
I	ok	no	ok	ok
T	ok	no	ok	ok

Figure 5. Compatibility modes for T locks.

	R	W	I	T	D
R	ok	no	ok	ok	&
W	no	no	*	#	no
I	ok	no	ok	ok	ok
T	ok	no	ok	ok	ok
D	ok	no	*	#	ok

Figure 6. Compatibility modes for D locks.

indicates that when a command attempts to read an object on which a D lock has been set, the "demand" command must be substituted into the command being executed using an algorithm similar to query modification to produce a new command to execute. This new command represents a subgoal which the POSTGRES system attempts to satisfy. If another D lock is encountered, a new subgoal will result, and the process will only terminate when a subgoal runs to completion and generates an answer. Moreover, this answer can be cached in the field and invalidated when necessary, if the intermediate goal commands set I locks as they run. This process is a database version of PROLOG style unification [CLOC81], and supports a backward chaining control flow. The algorithm details appear in [STON85b] along with a proposal for a priority scheme.

5.3. Storage System

The database will be partly stored on a magnetic disk and partly on an archival medium such as an optical disk. Data on magnetic disk includes all secondary indexes and recent database tuples. The optical disk is reserved as an archival store containing historical tuples. There will be a demon which "vacuums" tuples from magnetic disk to optical disk as a background process. Data on magnetic disk will be stored using the normal UNIX file system with one relation

per file. The optical disk will be organized as one large repository with tuples from various relations intermixed.

All relations will be stored as heaps (as in [ASTR76]) with an optional collection of secondary indexes. In addition relations can be declared "nearly ordered," and POSTGRES will attempt to keep tuples close to sort sequence on some column. Lastly, secondary indexes can be defined, which consist of two separate physical indexes one for the magnetic disk tuples and one for the optical disk tuples, each in a separate UNIX file on magnetic disk. Moreover, a secondary index on will automatically be provided for all relations on a unique identifier field which is described in the next subsection. This index will allow any relation to be sequentially scanned.

5.3.1. Data Format

Every tuple has an immutable unique identifier (IID) that is assigned at tuple creation time and never changes. This is a 64 bit quantity assigned internally by POSTGRES. Moreover, each transaction has a unique 64 bit transaction identifier (XACTID) assigned by POSTGRES. Lastly, there is a call to a system clock which can return timestamps on demand. Loosely, these are the current time-of-day.

Tuples will have all non-null fields stored adjacently in a physical record. Moreover, there will be a tuple prefix containing the following extra fields:

IID	: immutable id of this tuple
tmin	: the timestamp at which this tuple becomes valid
BXID	: the transaction identifier that assigned tmin
tmax	: the timestamp at which this tuple ceases to be valid
EXID	: the transaction identifier that assigned tmax
v-IID	: the immutable id of a tuple in this or some other version
descriptor	: descriptor on the front of a tuple

The descriptor contains the offset at which each non-null field starts, and is similar to the data structure attached to System R tuples [ASTR76]. The first transaction identifier and timestamp correspond to the timestamp and identifier of the creator of this tuple. When the tuple is updated, it is not overwritten; rather the identifier and timestamp of the updating transaction are recorded in the second (timestamp, transaction identifier) slot and a new tuple is constructed in the database. The update rules are described in the following subsection while the details of version management are deferred to later in the section.

5.3.2. Update and Access Rules

On an insert of a new tuple into a relation, tmin is marked with the timestamp of the inserting transaction and its identity is recorded in BXID. When a tuple is deleted, tmax is marked with the timestamp of the deleting transaction and its identity is recorded in EXID. An update to a tuple is modelled as an insert followed by a delete.

To find all the record which have the qualification, QUAL at time T the run time system must find all magnetic disk records such that:

1) tmin < T < tmax and BXID and EXID are committed and QUAL
2) tmin < T and tmax = null and BXID is committed and QUAL
3) tmin < T and BXID = committed and EXID = not-committed and QUAL

Then it must find all optical disk records satisfying 1). A special transaction log is described below that allows the DBMS to determine quickly whether a particular transaction has committed.

5.3.3. The POSTGRES Log and Accelerator

A new XACTID is assigned sequentially to each new transaction. When a transaction wishes to commit, all data pages which it has written must be forced out of memory (or at least onto stable storage). Then a single bit is written into the POSTGRES log and an optional transaction accelerator.

Consider three transaction identifiers; T1 which is the "youngest" transaction identifier which has been assigned, T2 which is a "young" transaction but guaranteed to be older than the oldest active transaction, and T3 which is a "young" transaction that is older than the oldest committed transaction which wrote data which is still on magnetic disk. Assume that T1-T3 are recorded in "secure main memory" to be presently described.

For any transaction with an identifier between T1 and T2, we need to know which of three states it is in:

0 = aborted
1 = committed
2 = in-progress

For any transaction with an identifier between T2 and T3, a "2" is impossible and the log can be compressed to 1 bit per transaction. For any transaction older than T3, the vacuum process has written all records to archival storage. During this vacuuming, the updates to all aborted transactions can be discarded, and hence all archival records correspond to committed transactions. No log need be kept for transactions older than T3.

The proposed log structure is an ordered relation, LOG as follows:

line-id: the access method supplied ordering field
bit-1[1000]: a bit vector
bit-2[1000]: a second bit vector

The status of xact number i is recorded in bit (remainder of i divided by 1000) of line-id number i/1000.

We assume that several thousand bits (say 1K-10K bytes) of "secure main memory" are available for 10-100 blocks comprising the "tail" of the log. Such main memory is duplexed or triplexed and supported by an uninterruptable power supply. The assumed hardware structure for this memory is the following. Assume a circular "block pool" of n blocks each of size 2000 bits. When more space is needed, the oldest block is reused. The hardware maintains a pointer which indicates the current largest xact identifier (T1 - the high water mark) and which bit it will use. it also has a second pointer which is the current oldest transaction in the buffer (the low water mark) and which bit it points to. When high-water approaches low-water, a block of the log must be "reliably" pushed to disk and

joins previously pushed blocks. Then low-water is advanced by 1000. High-water is advanced every time a new transaction is started. The operations available on the hardware structure are:

advance the high-water (i.e. begin a xact)
push a block and update low-water
abort a transaction
commit a transaction

Hopefully, the block pool is big enough to allow all transactions in the block to be committed or aborted before the block is "pushed." In this case, the block will never be updated on disk. If there are long running transactions, then blocks may be forced to disk before all transactions are committed or aborted. In this case, the subsequent commits or aborts will require an update to a disk-based block and will be much slower. Such disk operations on the LOG relation must be done by a special transaction (transaction zero) and will follow the normal update rules described above.

A trigger will be used to periodically advance T2 and replace bit-2 with nulls (which don't consume space) for any log records that correspond to transactions now older than T2.

At 5 transactions per second, the LOG relation will require about 20 Mbytes per year. Although we expect a substantial amount of buffer space to be available, it is clear that high transaction rate systems will not be able to keep all relevant portions of the XACT relation in main memory. In this case, the run-time cost to check whether individual transactions have been committed will be prohibitive. Hence, an optional transaction accelerator which we now describe will be a advantageous addition to POSTGRES.

We expect that virtually all of the transaction between T2 and T3 will be committed transactions. Consequently, we will use a second XACT relation as a bloom filter [SEVR76] to detect aborted transactions as follows. XACT will have tuples of the form:

line-id : the access method supplied ordering field
bitmap[M] : a bit map of size M

For any aborted transaction with a XACTID between T2 and T3, the following update must be performed. Let N be the number of transactions allocated to each XACT record and let LOW be T3 - remainder (T3/N).

replace XACT (bitmap[i] = 1)
where XACT.line-id = (XACTID - LOW)modulo N
and i = hash (remainder ((XACTID - LOW) / N))

The vacuum process advances T3 periodically and deletes tuples from XACT that correspond to transactions now older than T3. A second trigger will run periodically and advance T2 performing the above update for all aborted transactions now older than T2.

Consequently, whenever the run-time system wishes to check whether a candidate transaction, C-XACTID between T2 and T3 committed or aborted, it examines

bitmap[hash (reaminder((C-XACTID - LOW) / N))]

If a zero is observed, then C-XACTID must have committed, otherwise C-XACTID may have committed or aborted, and LOG must be examined to discover the true outcome.

The following analysis explores the performance of the transaction accelerator.

5.3.4. Analysis of the Accelerator

Suppose B bits of main memory buffer space are available and that M = 1000. These B bits can either hold some (or all) of LOG or they can hold some (or all) of XACT. Moreover, suppose transactions have a failure probability of F, and N is chosen so that X bits in bitmap are set on the average. Hence, N = X / F. In this case, a collection of Q transactions will require Q bits in LOG and

Q* F * 1000 / X

bits in the accelerator. If this quantity is greater than Q, the accelerator is useless because it takes up more space than LOG. Hence, assume that F * 1000 / X << 1. In this case, checking the disposition of a transaction in LOG will cause a page fault with probability:

FAULT (LOG) = 1 - [B / Q]

On the other hand, checking the disposition of a transaction in the accelerator will cause a page fault with probability:

P(XACT) = 1 - (B * X) / (Q * F * 1000)

With probability

X / 1000

a "1" will be observed in the accelerator data structure. If

B < Q * F * 1000 / X

then all available buffer space is consumed by the accelerator and a page fault will be assuredly generated to check in LOG if the transaction committed or aborted. Hence:

FAULT (XACT) = P(XACT) + X / 1000

If B is a larger value, then part of the buffer space can be used for LOG, and FAULT decreases.

The difference in fault probability between the log and the accelerator

delta = FAULT (LOG) - FAULT (XACT)

is maximized by choosing:

X = 1000 * square-root (F)

Figure 7 plots the expected number of faults in both systems for various buffer sizes with this value for X. As can be seen, the accelerator loses only when there is a miniscule amount of buffer space or when there is nearly enough to hold the whole log. Moreover

size (XACT) = square-root (F) * size (LOG)

and if

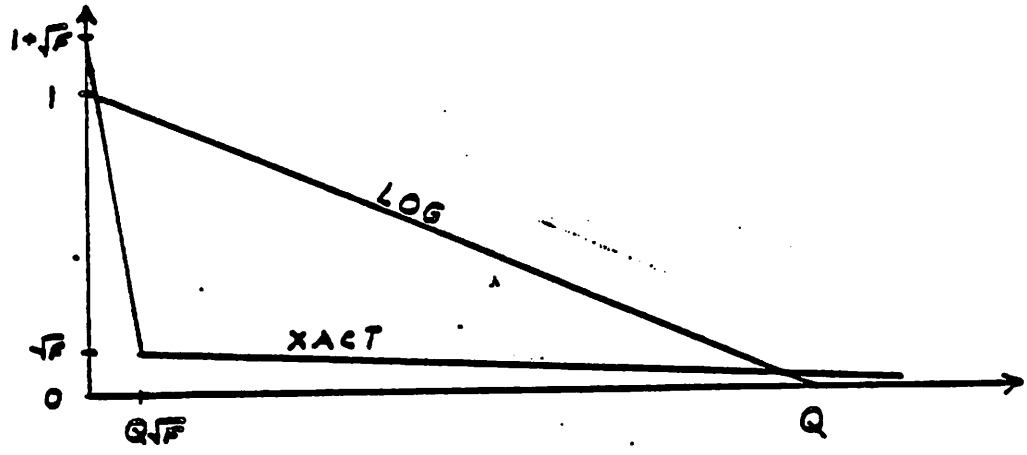

Figure 7. Expected number of faults versus buffer size.

B = size (XACT)

then the fault probability is lowered from

FAULT (LOG) = 1 - square-root (F)

to

FAULT (XACT) = square-root (F)

If F = .01, then buffer requirements are reduced by a factor of 10 and FAULT from .9 to .1. Even when F = .1, XACT requires only one-third the buffer space, and cuts the fault probability in half.

5.3.5. Transaction Management

If a crash is observed for which the disk-based database is intact, then all the recovery system must do is advance T2 to be equal to T1 marking all transactions in progress at the time of the crash "aborted." After this step, normal processing can commence. It is expected that recovery from "soft" crashes will be essentially instantaneous.

Protection from the perils of "hard" crashes, i.e. ones for which the disk is not intact will be provided by mirroring database files on magnetic disk either on a volume by volume basis in hardware or on a file by file basis in software.

We envison a conventional two phase lock manager handling read and write locks along with I, T and D locks. It is expected that R and W locks will be placed in a conventional main memory lock table, while other locks will reside in data records. The only extension which we expect to implement is "object locking." In this situation, a user can declare that his stored procedures are to be executed with

no locking at all. Of course, if two uses attempt to execute a stored procedure at the same time, one will be blocked because the first executor will place a write lock on the executed tuple. In this way, if a collection of users is willing to guarantee that there are no "blind" accesses to the pieces of objects (by someone directly accessing relations containing them), then they can be guaranteed consistency by the placement of normal read and write locks on procedural objects and no locks at all on the component objects.

5.3.6. Access Methods

We expect to implement both B-tree and OB-tree [STON83b] secondary indexes. Moreover, our ADT facility supports an arbitrary collection of user defined indexes. Each such index is, in reality, a pair of indexes one for magnetic disk records and one for archival records. The first index is of the form

index-relation (user-key-or-keys, pointer-to-tuple)

and uses the same structure as current INGRES secondary indexes. The second index will have pointers to archival tuples and will add "tmin" and "tmax" to whatever user keys are declared. With this structure, records satisfying the qualification:

where relation.key = value

will be interpreted to mean:

where (relation["now"].key = value)

and will require searching only the magnetic disk index. General queries of the form:

where relation[T].key = value

will require searching both the magnetic disk and the archival index. Both indexes need only search for records with qualifying keys; moreover the archival index can further restrict the search using tmax and tmin.

Any POSTQUEL replace command will insert a new data record with an appropriate BXID and tmin, and then insert a record into all key indexes which are defined, and lastly change tmax on the record to be updated. A POSTQUEL append will only perform the first and third steps while a delete only performs the second step. Providing a pointer from the old tuple to the new tuple would allow POSTGRES to insert records only into indexes for keys that are modified. This optimization saves many disk writes at some expense in run-time complexity. We plan to implement this optimization.

The implementor of a new access method structure need only keep in mind that the new data record must be forced from main memory before any index records (or the index record will point to garbage) and that multiple index updates (e.g. page splits) must be forced in the correct order (i.e. from leaf to root). This is easily accomplished with a single low level command to the buffer manager:

order page1, page2

Inopportune crashes may leave an access method which consists of a multi-level tree with dangling index pages (i.e. pages that are not pointed two from anywhere else in the tree). Such crashes may also leave the heap with uncommitted data

records that cannot be reached from some indexes. Such dangling tuples will be garbage collected by the vacuum process because they will have EXID equal to not committed. Unfortunately if dangling data records are not recorded in any index, then a sweep of memory will be periodicaly required to find them. Dangling index pages must be garbage collected by conventional techniques.

Ordered relations pose a special problem in our environment, and we propose to change OB trees slightly to cope with the situation. In particular, each place there is a counter in the original proposal [STON83b] indicating the number of descendent tuple-identifiers, the counter must be replaced by the following:

counter-1 : same as counter
flag : the danger bit

Any inserter or deleter in an OB tree will set the danger flag whenever he updates counter-1. Any OB tree accessor who reads a data item with the danger flag set must interrupt the algorithm and recompute counter-1 (by descending the tree). Then he reascends updating counter-1 and resetting the flag. After this interlude, he continues with his computation. In this way the next transaction "fixes up" the structure left dangling by the previous inserter or deleter, and OB-trees now work correctly.

5.3.7. Vacuuming the Disk

Any record with BXID and EXID of committed can be written to an optical disk or other long term repository. Moreover, any records with an BXID or EXID corresponding to an aborted transaction can be discarded. The job of a "vacuum" demon is to perform these two tasks. Consequently, the number of magnetic disk records is nearly equal to the number with EXID equal to null (i.e. the magnetic disk holds the current "state" of the database). The archival store holds historical records, and the vacuum demon can ensure that ALL archival records are valid. Hence, the run-time POSTGRES system need never check for the validity of archived records.

The vacuum process will first write a historical record to the archival store, then insert a record in the IID archival index, then insert a record in any archival key indexes, then delete the record from magnetic disk storage, and finaly delete the record from any magnetic disk indexes. If a crash occurs, the vacuum process can simply begin at the start of the sequence again.

If the vacuum process promptly archives historical records, then one requires disk space for the currently valid records plus a small portion of the historical records (perhaps about 1.2 times the size of the currently valid database) Additionally, one should be able to maintain good physical clustering on the attribute for which ordering is being attempted on the magnetic disk data set because there is constant turnover of records.

Some users may wish recently updated records to remain on magnetic disk To accomplish this tuning, we propose to allow a user to instruct the vacuum as follows:

vacuum rel-name where QUAL

A reasonable qualification might be:

vacuum rel-name where rel-name.tmax < now - 20 days

In this case, the vacuum demon would not remove records from the magnetic disk representation of rel-name until the qualification became true.

5.3.8. Version Management

Versions will be implemented by allocating a differential file [SEVR76] for each separate version. The differential file will contain the tuples added to or subtracted from the base relation. Secondary indexes will be built on versions to correspond to those on the base relation from which the version is constructed.

The algorithm to process POSTQUEL commands on versions is to begin with the differential relation corresponding to the version itself. For any tuple which satisfies the qualification, the v-IID of the inspected tuple must be remembered on a list of "seen IID's" [WOOD83]. If a tuple with an IID on the "seen-id" list is encountered, then it is discarded. As long as tuples can be inspected in reverse chronological order, one will always notice the latest version of a tuple first, and then know to discard earlier tuples. If the version is built on top of another version, then continue processing in the differential file of the next version. Ultimately, a base relation will be reached and the process will stop.

If a tuple in a version is modified in the current version, then it is treated as a normal update. If an update to the current version modifies a tuple in a previous version or the base relation, then the IID of the replaced tuple will be placed in the v-IID field and an appropriate tuple inserted into the differential file for the version. Deletes are handled in a similar fashion.

To merge a version into a parent version then one must perform the following steps for each record in the new version valid at time T:

1) if it is an insert, then insert record into older version
2) if it is a delete, then delete the record in the older version
3) if it is a replace, then do an insert and a delete

There is a conflict if one attempts to delete an already deleted record. Such cases must be handled external to the algorithm. The tactics in [GARC84] may be helpful in reconciling these conflicts.

An older version can be rolled forward into a newer version by performing the above operations and then renaming the older version.

6. SUMMARY

POSTGRES proposes to support complex objects by supporting an extendible type system for defining new columns for relations, new operators on these columns, and new access methods. This facility is appropriate for fairly "simple" complex objects. More complex objects, especially those with shared subobjects or multiple levels of nesting, should use POSTGRES procedures as their definition mechanism. Procedures will be optimized by caching compiled plans and even answers for retrieval commands.

Triggers and rules are supported as commands with "always" and "demand" modifiers. They are efficiently supported by extensions to the locking system. Both

forward chaining and backward chaining control structures are provided within the data manager using these mechanisms. Our rules system should prove attractive when there are multiple rules which might apply in any given situation.

Crash recovery is simplified by not overwriting data and then vacuuming tuples to an archive store. The new storage system is greatly simplified from current technology and supports time-oriented access and versions with little difficulty. The major cost of the storage system is the requirement to push dirty pages of data to stable storage at commit time.

An optical disk is used effectively as an archival medium, and POSTGRES has a collection of demons running in the background. These can effectively utilize otherwise idle processors. Custom hardware could effectively provide stable main memory, support for the LOG relation, and support for run-time checking of tuple validity.

Lastly, these goals are accomplished with no changes to the relational model at all. At the current time coding of POSTGRES is just beginning. We hope to have a prototype running in about a year.

REFERENCES

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

[AFSA85] Afasarmanesh, H., et. al., "An Extensible Object-Oriented Approach to Database for VLSI/CAD," Proc. 1985 Very Large Data Base Conference, Stockholm, Sweden, August 1985.

[ALLM76] Allman, E., et. al., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc 1976 ACM-SIGPLAN-SIGMOD Conference on Data, Salt Lake City, Utah, March 1976.

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

[ATKI84] Atkinson, M.P. et. al., "Progress with Persistent Programming," in Database, Role and Structure (ed. P. Stocker), Cambridge Univeristy of Press, 1984.

[BUNE79] Bunemann, P. and Clemons, E., "Efficiently Monitoring Relational Data Bases," ACM-TODS, Sept. 1979.

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

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

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

[DERR85] Derritt, N., Personal Communication, HP Laboratories, October 1985.

[DEWI85] DeWitt, D.J. and Carey, M.J., "Extensible Database Systems," Proc. 1st International Workshop on Expert Data Bases, Kiowah, S.C., Oct 1984.

[ESWA75] Eswaren, K., "A General Purpose Trigger Subsystem and Its Inclusion in a Relational Data Base System," IBM Research, San Jose, Ca., RJ 1833, July 1976.

[GARC84] Garcia-Molina, H., et. al., "Data-Patch: Integrating Inconsistent copies of a Database after a Partition," Tech. Rep. TR# 304, Dept. Elec. Eng. and Comp. Sci., Princeton Univ., 1984.

[HELD75] Held, G. et. al., "INGRES: A Relational Data Base System," Proc 1975 National Computer Conference, Anaheim, Ca., June 1975.

[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.

[JARK85] Jarke, M. et. al., "Data Constructors: On the Integration of Rules and Relations," Proc. 1985 Very Large Data Base Conference, Stockholm, Sweden, August 1985.

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

[KUNG84] Kung, R. et. al., "Heuristic Search in Database Systems," Proc. 1st International Workshop on Expert Data Bases, Kiowah, S.C., Oct 1984.

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

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

[ROBI81] Robinson, J., "The K-D-B Tree: A Search Structure for Large Multidimensional Indexes," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.

[ROWE79] Rowe, L.A. and Shoens, K., "Data Abstraction, Views, and Updates in Rigel," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, MA, May 1979.

[ROWE82] Rowe, L.A. and Shoens, K. "FADS - A Forms Application Development System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FL, June 1982.

[ROWE85] Rowe, L., "Fill-in-the-Form Programming," Proc. 1985 Very Large Data Base Conference, Stockholm, Sweden, August 1985.

[SELI79] Selinger, P. et. al., "Access Path Selection in a Relational Data Base System," Proc 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[SEVR76] Severence, D., and Lohman, G., "Differential Files: Their Application to the Maintenance of large Databases," ACM-TODS, June 1976.

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

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

[STON81] Stonebraker, M., "Operating System Support for Database Management," CACM, July 1981.

[STON83a] Stonebraker, M., et. al., "Performance Analysis of a Distributed Data Base System," Proc. 3th Symposium on Reliability in Distributed Software and Data Base Systems, Clearwater, Fla, Oct. 1983

[STON83b] Stonebraker, M., "Document Processing in a Relational Database System," ACM TOOIS, April 1983.

[STON83c] Stonebraker, M., et. al., "Application of Abstract Data Types and Abstract Indexes to CAD Data," Proc. Engineering Applications Stream of 1983 Data Base Week, San Jose, Ca., May 1983.

[STON84a] Stonebraker, M. et. al., "QUEL as a Data Type," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[STON84b] Stonebraker, M. and Rowe, L.A., "PORTALS: A New Application Program Interface," Proc. 1984 VLDB Conference, Singapore, Sept 1984.

[STON85a] Stonebraker, M., "Extending a Data Base System with Procedures," (submitted for publication).

[STON85b] Stonebraker, M., "Triggers and Inference in Data Base Systems," Proc. Islamoora Conference on Expert Data Bases, Islamoora, Fla., Feb 1985, to appear as a Springer-Verlag book.

[STON85c] Stonebraker, M. et. al., "An Analysis of Rule Indexing Implementations in Data Base Systems," (submitted for publication)

[STON86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[TICH82] Tichy, W.F., "Design, Implementation, and Evaluation of a Revision Control System, Proc. 6th Int. Conf. on Soft. Eng., Sept 1982.

[TSIC82] Tsichritzis, D.C. "Form Management," CACM 25, July 1982.

[ULLM85] Ullman, J., "Implementation of Logical Query Languages for Data Bases," Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data, Austin, TX, May 1985.

[WONG84] Wong, E., et al., "Enhancing INGRES with Deductive Power," Proceedings of the 1st International Workshop on Expert Data Base Systems, Kiowah SC, October 1984.

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

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

The POSTGRES Data Model†

Lawrence A. Rowe
Michael R. Stonebraker

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Abstract

The design of the POSTGRES data model is described. The data model is a relational model that has been extended with abstract data types including user-defined operators and procedures, relation attributes of type procedure, and attribute and procedure inheritance. These mechanism can be used to simulate a wide variety of semantic and object-oriented data modeling constructs including aggregation and generalization, complex objects with shared subobjects, and attributes that reference tuples in other relations.

1. Introduction

This paper describes the data model for POSTGRES, a next-generation extensible database management system being developed at the University of California [23]. The data model is based on the idea of extending the relational model developed by Codd [5] with general mechanisms that can be used to simulate a variety of semantic data modeling constructs. The mechanisms include: 1) abstract data types (ADT's), 2) data of type procedure, and 3) rules. These mechanisms can be used to support complex objects or to implement a shared object hierarchy for an object-oriented programming language [17]. Most of these ideas have appeared elsewhere [21,22,24,25].

We have discovered that some semantic constructs that were not directly supported can be easily added to the system. Consequently, we have made several changes to the data model and the syntax of the query language that are documented here. These changes include providing support for primary keys, inheritance of data and procedures, and attributes that reference tuples in other relations.

The major contribution of this paper is to show that inheritance can be added to a relational data model with only a modest number of changes to the model and the implementation of the system. The conclusion that we draw from this result is that the major concepts provided in an object-oriented data model (e.g., structured attribute types, inheritance, union type attributes, and support for shared subobjects) can be cleanly and efficiently supported in an extensible relational database management system. The features used to support these mechanisms are abstract data types and attributes of type procedure.

The remainder of the paper describes the POSTGRES data model and is organized as follows. Section 2 presents the data model. Section 3 describes the attribute type system. Section 4 describes how the query language can be extended with user-defined procedures. Section 5 compares the model with other data models and section 6 summarizes the paper.

2. Data Model

A database is composed of a collection of *relations* that contain tuples which represent real-world entities (e.g., documents and people) or relationships (e.g., authorship). A relation has attributes of fixed types that represent properties of the entities and relationships (e.g.,

† This research was supported by the National Science Foundation under Grant DCR-8507256 and the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

the title of a document) and a primary key. Attribute types can be atomic (e.g., integer, floating point, or boolean) or structured (e.g., array or procedure). The primary key is a sequence of attributes of the relation, when taken together, uniquely identify each tuple.

A simple university database will be used to illustrate the model. The following command defines a relation that represents people:

```
create PERSON ( Name = char[25],
    Birthdate = date, Height = int4,
    Weight = int4, StreetAddress = char[25],
    City = char[25], State = char[2])
```

This command defines a relation and creates a structure for storing the tuples.

The definition of a relation may optionally specify a primary key and other relations from which to inherit attributes. A primary key is a combination of attributes that uniquely identify each tuple. The key is specified with a key-clause as follows:

```
create PERSON ( . . .)
key (Name)
```

Tuples must have a value for all key attributes. The specification of a key may optionally include the name of an operator that is to be used when comparing two tuples. For example, suppose a relation had a key whose type was a user-defined ADT. If an attribute of type *box* was part of the primary key, the comparison operator must be specified since different *box* operators could be used to distinguish the entries (e.g., area equals or box equality). The following example shows the definition of a relation with a key attribute of type *box* that uses the area equals operator (*AE*) to determine key value equality:

```
create PICTURE(Title = char[25], Item = box)
key (Item using AE)
```

Data inheritance is specified with an inherits-clause. Suppose, for example, that people in the university database are employees and/or students and that different attributes are to be defined for each category. The relation for each category includes the *PERSON* attributes and the attributes that are specific to the category. These relations can be defined by replicating the *PERSON* attributes in each relation definition or by inheriting them for the definition of *PERSON*. Figure 1 shows the relations and an inheritance hierarchy that could be used to share the definition of the attributes. The commands that define the relations other than the *PERSON* relation defined above are:

```
create EMPLOYEE (Dept = char[25],
    Status = int2, Mgr = char[25],
    JobTitle = char[25], Salary = money)
inherits (PERSON)

create STUDENT (Sno = char[12],
    Status = int2, Level = char[20])
inherits (PERSON)

create STUDEMP (IsWorkStudy = bool)
inherits (STUDENT, EMPLOYEE)
```

A relation inherits all attributes from its parent(s) unless an attribute is overriden in the definition. For example, the *EMPLOYEE* relation inherits the *PERSON* attributes *Name*, *Birthdate*, *Height*, *Weight*, *StreetAddress*, *City*, and *State*. Key specifications are also inherited so *Name* is also the key for EMPLOYEE.

Relations may inherit attributes from more than one parent. For example, *STUDEMP* inherits attributes from *STUDENT* and *EMPLOYEE*. An inheritance conflict occurs when the same attribute name is inherited from more than one parent (e.g., *STUDEMP* inherits *Status* from *EMPLOYEE* and *STUDENT*). If the inherited attributes have the same type, an attribute with the type is

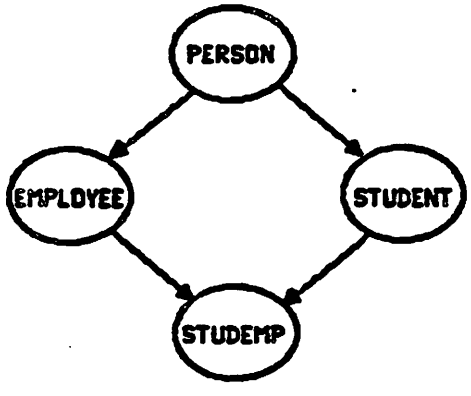

Figure 1: Relation hierarchy.

included in the relation that is being defined. Otherwise, the declaration is disallowed.[1]

The POSTGRES query language is a generalized version of QUEL [13], called *POSTQUEL*. QUEL was extended in several directions. First, POSTQUEL has a from-clause to define tuple-variables rather than a range command. Second, arbitrary relation-valued expressions may appear any place that a relation name could appear in QUEL. Third, transitive closure and execute commands have been added to the language [14]. And lastly, POSTGRES maintains historical data so POSTQUEL allows queries to be run on past database states or on any data that was in the database at any time. These extensions are described in the remainder of this section.

The from-clause was added to the language so that tuple-variable definitions for a query could be easily determined at compile-time. This capability was needed because POSTGRES will, at the user's request, compile queries and save them in the system catalogs. The from-clause is illustrated in the following query that lists all work-study students who are sophomores:

```
retrieve (SE.name)
from SE in STUDEMP
where SE.IsWorkStudy
   and SE.Status = "sophomore"
```

The from-clause specifies the set of tuples over which a tuple-variable will range. In this example, the tuple-variable *SE* ranges over the set of student employees.

A default tuple-variable with the same name is defined for each relation referenced in the target-list or where-clause of a query. For example, the query above could have been written:

```
retrieve (STUDEMP.name)
where STUDEMP.IsWorkStudy
   and STUDEMP.Status = "sophomore"
```

Notice that the attribute *IsWorkStudy* is a boolean-valued attribute so it does not require an explicit value test (e.g., *STUDEMP.IsWorkStudy = "true"*).

The set of tuples that a tuple-variable may range over can be a named relation or a relation-expression. For example, suppose the user wanted to retrieve all students in the database who live in Berkeley regardless of whether they are students or student employees. This query can be written as follows:

```
retrieve (S.name)
from S in STUDENT*
where S.city = "Berkeley"
```

The "*" operator specifies the relation formed by taking the union of the named relation (i.e., *STUDENT*) and all relations that inherit attributes from it (i.e., *STUDEMP*). If the "*" operator was not used, the query retrieves only tuples in the student relation (i.e., students who are not student employees). In most data models that support inheritance the relation name defaults to the union of relations over the inheritance hierarchy (i.e., the data described by *STUDENT** above). We chose a different default because queries that involve unions will be slower than queries on a single relation. By forcing the user to request the union explicitly with the "*" operator, he will be aware of this cost.

Relation expressions may include other set operators:

```
delim $$
```

union ($union$), intersection ($inter$), and difference (−). For example, the following query retrieves the names of people who are students or employees but not student employees:

```
retrieve (S.name)
from S in (STUDENT $union$ EMPLOYEE)
```

Suppose a tuple does not have an attribute referenced elsewhere in the query. If the reference is in the target-list, the return tuple will

[1] Most attribute inheritance models have a conflict resolution rule that selects one of the conflicting attributes. We chose to disallow inheritance because we could not discover an example where it made sense, except when the types were identical. On the other hand, procedure inheritance (discussed below) does use a conflict resolution rule because many examples exist in which one procedure is prefered.

not contain the attribute.[2] If the reference is in the qualification, the clause containing the qualification is "false".

POSTQUEL also provides set comparison operators and a relation-constructor that can be used to specify some difficult queries more easily than in a conventional query language.

delim off

For example, suppose that students could have several majors. The natural representation for this data is to define a separate relation:

```
create MAJORS(Sname = char[25],
    Mname = char[25])
```

where *Sname* is the student's name and *Mname* is the major. With this representation, the following query retrieves the names of students with the same majors as Smith:

```
retrieve (M1.Sname)
from M1 in MAJORS
where {(x.Mname) from x in MAJORS
          where x.Sname = M1.Sname}
  ⊂ {(x.Mname) from x in MAJORS
       where x.Sname="Smith"}
```

The expressions enclosed in set symbols ("{...}") are relation-constructors.

The general form of a relation-constructor[3] is

```
{(target-list) from from-clause
          where where-clause}
```

which specifies the same relation as the query

```
retrieve (target-list)
from from-clause
where where-clause
```

Note that a tuple-variable defined in the outer query (e.g., M1 in the query above) can be used within a relation-constructor but that a tuple-variable defined in the relation-constructor cannot be used in the outer query. Redefinition of a tuple-variable in a relation constructor creates a distinct variable as in a block-structured programming language (e.g., PASCAL). Relation-valued expressions (including attributes of type procedure described in the next section) can be used any place in a query that a named relation can be used.

Database updates are specified with conventional update commands as shown in the following examples:

```
/* Add a new employee to the database. */
append to EMPLOYEE(name = value,
    age = value, ...)

/* Change state codes using
    MAP(OldCode, NewCode). */
replace P(State = MAP.NewCode)
from P in PERSON*
where P.State = MAP.OldCode

/* Delete students born before today. */
delete STUDENT
where STUDENT.Birthdate < "today"
```

Deferred update semantics are used for all updates commands.

POSTQUEL supports the transitive closure commands developed in QUEL* [14]. A "*" command continues to execute until no tuples are retrieved (e.g., **retrieve***) or updated (e.g., **append***, **delete***, or **replace***). For example, the following query creates a relation that contains all employees who work for Smith:

```
retrieve* into SUBORD(E.Name, E.Mgr)
from E in EMPLOYEE, S in SUBORD
where E.Name = "Smith"
    or E.Mgr = S.Name
```

This command continues to execute the **retrieve-into** command until there are no changes made to the *SUBORD* relation.

Lastly, POSTGRES saves data deleted from or modified in a relation so that queries can be executed on historical data. For example, the following query looks for students who lived in Berkeley on August 1, 1980:

```
retrieve (S.Name)
from S in STUDENT["August 1, 1980"]
where S.City = "Berkeley"
```

The date specified in the brackets following the relation name specifies the relation at the designated time. The date can be specified in many different formats and optionally may

---

[2] The application program interface to POSTGRES allows the stream of tuples passed back to the program to have dynamically varying columns and types.

[3] Relation constructors are really aggregate functions. We have designed a mechanism to support extensible aggregate functions, but have not yet worked out the query language syntax and semantics.

include a time of day. The query above only examines students who are not student employees. To search the set of all students, the from-clause would be

```
...from S in STUDENT*["August 1, 1980"]
```

Queries can also be executed on all data that is currently in the relation or was in it at some time in the past (i.e., all data). The following query retrieves all students who ever lived in Berkeley:

```
retrieve (S.Name)
from S in STUDENT[]
where S.City = "Berkeley"
```

The notation "[]" can be appended to any relation name.

Queries can also be specified on data that was in the relation during a given time period. The time period is specified by giving a start- and end-time as shown in the following query that retrieves students who lived in Berkeley at any time in August 1980:

```
retrieve (S.Name)
from S in STUDENT*["August 1, 1980",
                   "August 31, 1980"]
where S.City = "Berkeley"
```

Shorthand notations are supported for all tuples in a relation up to some date (e.g., *STUDENT*[,"August 1, 1980"]*) or from some date to the present (e.g., *STUDENT*["August 1, 1980",]*).

The POSTGRES default is to save all data unless the user explicitly requests that data be purged. Data can be purged before a specific data (e.g., before January 1, 1987) or before some time period (e.g., before six months ago). The user may also request that all historical data be purged so that only the current data in the relation is stored.

POSTGRES also supports versions of relations. A version of a relation can be created from a relation or a snapshot. A version is created by specifying the base relation as shown in the command

```
create version MYPEOPLE from PERSON
```

that creates a version, named *MYPEOPLE*, derived from the *PERSON* relation. Data can be retrieved from and updated in a version just like a relation. Updates to the version do not modify the base relation. However, updates to the base relation are propagated to the version unless the value has been modified. For example, if George's birthdate is changed in *MYPEOPLE*, a replace command that changes his birthdate in *PERSON* will not be propagated to *MYPEOPLE*.

If the user does not want updates to the base relation to propagate to the version, he can create a version of a snapshot. A snapshot is a copy of the current contents of a relation [1]. A version of a snapshot is created by the following command:

```
create version YOURPEOPLE
from PERSON["now"]
```

The snapshot version can be updated directly by issuing update commands on the version. But, updates to the base relation are not propagated to the version.

A merge command is provided to merge changes made to a version back into the base relation. An example of this command is

```
merge YOURPEOPLE into PERSON
```

that will merge the changes made to *YOURPEOPLE* back into *PERSON*. The merge command uses a semi-automatic procedure to resolve updates to the underlying relation and the version that conflict [10].

This section described most of the data definition and data manipulation commands in POSTQUEL. The commands that were not described are the commands for defining rules, utility commands that only affect the performance of the system (e.g., **define index** and **modify**), and other miscellaneous utility commands (e.g., **destroy** and **copy**). The next section describes the type system for relation attributes.

3. Data Types

POSTGRES provides a collection of atomic and structured types. The predefined atomic types include: *int2*, *int4*, *float4*, *float8*, *bool*, *char*, and *date*. The standard arithmetic and comparison operators are provided for the numeric and date data types and the standard string and comparison operators for character arrays. Users can extend the system by adding new atomic types using an abstract data type (ADT) definition facility.

All atomic data types are defined to the system as ADT's. An ADT is defined by specifying the type name, the length of the internal

representation in bytes, procedures for converting from an external to internal representation for a value and from an internal to external representation, and a default value. The command

```
define type int4 is (InternalLength = 4,
    InputProc = CharToInt4,
    OutputProc = Int4ToChar, Default = "0")
```

defines the type *int4* which is predefined in the system. *CharToInt4* and *Int4ToChar* are procedures that are coded in a conventional programming language (e.g., C) and defined to the system using the commands described in section 4.

Operators on ADT's are defined by specifying the the number and type of operands, the return type, the precedence and associativity of the operator, and the procedure that implements it. For example, the command

```
define operator "+"(int4, int4) returns int4
    is (Proc = Plus, Precedence = 5,
        Associativity = "left")
```

defines the plus operator. Precedence is specified by a number. Larger numbers imply higher precedence. The predefined operators have the precedences shown in figure 2. These precedences can be changed by changing the operator definitions. Associativity is either left or right depending on the semantics desired. This example defined an operator denoted by a symbol (i.e., "+"). Operators can also be denoted by identifiers as shown below.

Precedence	Operators
80	↑
70	not − (unary)
60	* /
50	+ − (binary)
40	< ≤ > ≥
30	= ≠
20	and
10	or

Figure 2: Predefined operators precedence.

Another example of an ADT definition is the following command that defines an ADT that represents boxes:

```
define type box is (InternalLength = 16,
    InputProc = CharToBox,
    OutputProc = BoxToChar, Default = "")
```

The external representation of a box is a character string that contains two points that represent the upper-left and lower-right corners of the box. With this representation, the constant

```
"20,50:10,70"
```

describes a box whose upper-left corner is at (20, 50) and lower-right corner is at (10, 70). *CharToBox* takes a character string like this one and returns a 16 byte representation of a box (e.g., 4 bytes per x- or y-coordinate value). *BoxToChar* is the inverse of *CharToBox*

Comparison operators can be defined on ADT's that can be used in access methods or optimized in queries. For example, the definition

```
define operator AE(box, box) returns bool
    is (Proc = BoxAE, Precedence = 3,
        Associativity = "left", Sort = BoxArea,
        Hashes, Restrict = AERSelect,
        Join = AEJSelect, Negator = BoxAreaNE)
```

defines an operator "area equals" on boxes. In addition to the semantic information about the operator itself, this specification includes information used by POSTGRES to build indexes and to optimize queries using the operator. For example, suppose the *PICTURE* relation was defined by

```
create PICTURE(Title = char[], Item = box)
```

and the query

```
retrieve (PICTURE.all)
where PICTURE.Item AE "50,100:100,50"
```

was executed. The *Sort* property of the *AE* operator specifies the procedure to be used to sort the relation if a merge-sort join strategy was selected to implement the query. It also specifies the procedure to use when building an ordered index (e.g., B-Tree) on an attribute of type *box*. The *Hashes* property indicates that this operator can be used to build a hash index on a *box* attribute. Note that either type of index can be used to optimize the query above. The *Restrict* and *Join* properties specify the

procedure that is to be called by the query optimizer to compute the restrict and join selectivities, respectively, of a clause involving the operator. These selectivity properties specify procedures that will return a floating point value between 0.0 and 1.0 that indicate the attribute selectivity given the operator. Lastly, the *Negator* property specifies the procedure that is to be used to compare two values when a query predicate requires the operator to be negated as in

```
retrieve (PICTURE.all)
where not (PICTURE.Item
                AE "50,100:100,50")
```

The define operator command also may specify a procedure that can be used if the query predicate includes an operator that is not commutative. For example, the commutator procedure for "area less than" (*ALT*) is the procedure that implements "area greater than or equal" (*AGE*). More details on the use of these properties is given elsewhere [25].

Type-constructors are provided to define structured types (e.g., arrays and procedures) that can be used to represent complex data. An *array* type-constructor can be used to define a variable- or fixed-size array. A fixed-size array is declared by specifying the element type and upper bound of the array as illustrated by

```
create PERSON(Name = char[25])
```

which defines an array of twenty-five characters. The elements of the array are referenced by indexing the attribute by an integer between 1 and 25 (e.g., *"PERSON.Name[4]"* references the fourth character in the person's name).

A variable-size array is specified by omitting the upper bound in the type constructor. For example, a variable-sized array of characters is specified by "char[]." Variable-size arrays are referenced by indexing the attribute by an integer between 1 and the current upper bound of the array. The predefined function *size* returns the current upper bound. POSTGRES does not impose a limit on the size of a variable-size array. Built-in functions are provided to append arrays and to fetch array slices. For example, two character arrays can be appended using the concatenate operator ("+") and an array slice containing characters 2 through 15 in an attribute named *x* can be fetched by the expression "x[2:15]."

The second type-constructor allows values of type procedure to be stored in an attribute. Procedure values are represented by a sequence of POSTQUEL commands. The value of an attribute of type procedure is a relation because that is what a retrieve command returns. Moreover, the value may include tuples from different relations (i.e., of different types) because a procedure composed of two retrieve commands returns the union of both commands. We call a relation with different tuple types a *multirelation*. The POSTGRES programming language interface provides a cursor-like mechanism, called a *portal*, to fetch values from multirelations [23]. However, they are not stored by the system (i.e., only relations are stored).

The system provides two kinds of procedure type-constructors: variable and parameterized. A variable procedure-type allows a different POSTQUEL procedure to be stored in each tuple while parameterized procedure-types store the same procedure in each tuple but with different parameters. We will illustrate the use of a variable procedure-type by showing another way to represent student majors. Suppose a *DEPARTMENT* relation was defined with the following command:

```
create DEPARTMENT(Name = char[25],
    Chair = char[25], ...)
```

A student's major(s) can then be represented by a procedure in the *STUDENT* relation that retrieves the appropriate *DEPARTMENT* tuple(s). The *Majors* attribute would be declared as follows:

```
create STUDENT(..., Majors = postquel, ...)
```

Data type *postquel* represents a procedure-type. The value in *Majors* will be a query that fetches the department relation tuples that represent the student's minors. The following command appends a student to the database who has a double major in mathematics and computer science:

```
append STUDENT( Name = "Smith", ....
    Majors =
        "retrieve (D.all)
         from D in DEPARTMENT
         where D.Name = "Math"
            or D.Name = "CS"")
```

A query that references the *Majors* attribute returns the string that contains the POSTQUEL commands. However, two notations are provided that will execute the query and return the result rather than the definition. First, nested-dot notation implicitly executes the query as illustrated by

```
retrieve (S.Name, S.Majors.Name)
from S in STUDENT
```

which prints a list of names and majors of students. The result of the query in *Majors* is implicitly joined with the tuple specified by the rest of the target-list. In other words, if a student has two majors, this query will return two tuples with the *Name* attribute repeated. The implicit join is performed to guarantee that a relation is returned.

The second way to execute the query is to use the execute command. For example, the query

```
execute (S.Majors)
from S in STUDENT
where S.Name = "Smith"
```

returns a relation that contains *DEPARTMENT* tuples for all of Smith's majors.

Parameterized procedure-types are used when the query to be stored in an attribute is nearly the same for every tuple. The query parameters can be taken from other attributes in the tuple or they may be explicitly specified. For example, suppose an attribute in *STUDENT* was to represent the student's current class list. Given the following definition for enrollments:

```
create ENROLLMENT(Student = char[25],
    Class = char[25])
```

Bill's class list can be retrieved by the query

```
retrieve (ClassName = E.Class)
from E in ENROLLMENT
where E.Student = "Bill"
```

This query will be the same for every student except for the constant that specifies the student's name.

A parameterized procedure-type could be defined to represent this query as follows:

```
define type classes is
    retrieve (ClassName = E.Class)
    from E in ENROLLMENT
    where E.Student = $.Name
end
```

The dollar-sign symbol ("$") refers to the tuple in which the query is stored (i.e., the current tuple). The parameter for each instance of this type (i.e., a query) is the *Name* attribute in the tuple in which the instance is stored. This type is then used in the create command as follows

```
create STUDENT(Name = char[25], ...,
    ClassList = classes)
```

to define an attribute that represents the student's current class list. This attribute can be used in a query to return a list of students and the classes they are taking:

```
retrieve (S.Name, S.ClassList.ClassName)
```

Notice that for a particular *STUDENT* tuple, the expression "*$.Name*" in the query refers to the name of that student. The symbol "$" can be thought of as a tuple-variable bound to the current tuple.

Parameterized procedure-types are extremely useful types, but sometimes it is inconvenient to store the parameters explicitly as attributes in the relation. Consequently, a notation is provided that allows the parameters to be stored in the procedure-type value. This mechanism can be used to simulate attribute types that reference tuples in other relations. For example, suppose you wanted a type that referenced a tuple in the *DEPARTMENT* relation defined above. This type can be defined as follows:

```
define type DEPARTMENT(int4) is
    retrieve (DEPARTMENT.all)
    where DEPARTMENT.oid = $1
end
```

The relation name can be used for the type name because relations, types, and procedures have separate name spaces. The query in type *DEPARTMENT* will retrieve a specific department tuple given a unique object identifier (*oid*) of the tuple. Each relation has an implicitly defined attribute named *oid* that contains the tuple's unique identifier. The *oid* attribute can be accessed but not updated by user queries. *Oid* values are created and maintained by the POSTGRES storage system [26]. The formal argument to this procedure-type is the type of an object identifier. The parameter is referenced inside the definition by "*$n*" where *n* is the parameter number.

An actual argument is supplied when a value is assigned to an attribute of type *DEPARTMENT*. For example, a *COURSE* relation can be defined that represents information about a specific course including the department that offers it. The create command is:

```
create COURSE(Title = char[25],
    Dept = DEPARTMENT, ...)
```

The attribute *Dept* represents the department that offers the course. The following query adds a course to the database:

```
append COURSE(
    Title = "Introductory Programming",
    Dept = DEPARTMENT(D.oid))
from D in DEPARTMENT
where D.Name = "computer science"
```

The procedure *DEPARTMENT* called in the target-list is implicitly defined by the "define type" command. It constructs a value of the specified type given actual arguments that are type compatible with the formal arguments, in this case an *int4*.

Parameterized procedure-types that represent references to tuples in a specific relation are so commonly used that we plan to provide automatic support for them. First, every relation created will have a type that represents a reference to a tuple implicitly defined similar to the *DEPARTMENT* type above. And second, it will be possible to assign a tuple-variable directly to a tuple reference attribute. In other words, the assignment to the attribute *Dept* that is written in the query above as

```
... Dept = DEPARTMENT(D.oid) ...
```

can be written as

```
... Dept = D ...
```

Parameterized procedure-types can also be used to implement a type that references a tuple in an arbitrary relation. The type definition is:

```
define type tuple(char[], int4) is
    retrieve ($1.all)
    where $1.oid = $2
end
```

The first argument is the name of the relation and the second argument is the *oid* of the desired tuple in the relation. In effect, this type defines a reference to an arbitrary tuple in the database.

The procedure-type *tuple* can be used to create a relation that represents people who help with fund raising:

```
create VOLUNTEER(Person = tuple,
    TimeAvailable = integer, ...)
```

Because volunteers may be students, employees, or people who are neither students nor employees, the attribute *Person* must contain a reference to a tuple in an arbitrary relation. The following command appends all students to *VOLUNTEER*:

```
append VOLUNTEER(
    Person = tuple(relation(S), S.oid))
from S in STUDENT*
```

The predefined function *relation* returns the name of the relation to which the tuple-variable *S* is bound.

The type *tuple* will also be special-cased to make it more convenient. *Tuple* will be a predefined type and it will be possible to assign tuple-variables directly to attributes of the type. Consequently, the assignment to *Person* written above as

```
... Person = tuple(relation(S), S.oid) ...
```

can be written

```
... Person = S ...
```

We expect that as we get more experience with POSTGRES applications that more types may be special-cased.

4. User-Defined Procedures

This section describes language constructs for adding user-defined procedures to POSTQUEL. User-defined procedures are written in a conventional programming language and are used to implement ADT operators or to move a computation from a front-end application process to the back-end DBMS process.

Moving a computation to the back-end opens up possibilities for the DBMS to precompute a query that includes the computation. For example, suppose that a front-end application needed to fetch the definition of a form from a database and to construct a main-memory data structure that the run-time forms system used to display the form on the terminal screen for data entry or display. A conventional relation database design would store the form components (e.g., titles and field definitions for different types of fields such as

scalar fields, table fields, and graphics fields) in many different relations. An example database design is:

```
create FORM(FormName, ...)

create FIELDS(FormName, FieldName,
    Origin, Height, Width,
    FieldKind, ...)

create SCALARFIELD(FormName,
    FieldName, DataType,
    DisplayFormat, ...)

create TABLEFIELD(FormName,
    FieldName, NumberOfRows, ...)

create TABLECOLUMNS(FormName,
    FieldName, ColumnName, Height,
    Width, FieldKind, ...)
```

The query that fetches the form from the database must execute at least one query per table and sort through the return tuples to construct the main-memory data structure. This operation must take less than two seconds for an interactive application. Conventional relational DBMS's cannot satisfy this time constraint.

Our approach to solving this problem is to move the computation that constructs the main-memory data structure to the database process. Suppose the procedure *MakeForm* built the data structure given the name of a form. Using the parameterized procedure-type mechanism defined above an attribute can be added to the *FORM* relation that stores the form representation computed by this procedure. The commands

```
define type formrep is
    retrieve (rep = MakeForm($.FormName))
end
addattribute (FormName, ...,
    FormDataStructure = formrep)
to FORM
```

define the procedure type and add an attribute to the *FORM* relation.

The advantage of this representation is that POSTGRES can precompute the answer to a procedure-type attribute and store it in the tuple. By precomputing the main-memory data structure representation, the form can be fetched from the database by a single-tuple retrieve:

```
retrieve (x = FORM.FormDataStructure)
where FORM.FormName = "foo"
```

The real-time constraint to fetch and display a form can be easily met if all the program must do is a single-tuple retrieve to fetch the data structure and call the library procedure to display it. This example illustrates the advantage of moving a computation (i.e., constructing a main-memory data structure) from the application process to the DBMS process.

A procedure is defined to the system by specifying the names and types of the arguments, the return type, the language it is written in, and where the source and object code is stored. For example, the definition

```
define procedure AgeInYears(date) returns int4
    is (language = "C", filename = "AgeInYears")
```

defines a procedure *AgeInYears* that takes a *date* value and returns the age of the person. The argument and return types are specified using POSTGRES types. When the procedure is called, it is passed the arguments in the POSTGRES internal representation for the type. We plan to allow procedures to be written in several different languages including C and Lisp which are the two languages being used to implement the system.

POSTGRES stores the information about a procedure in the system catalogs and dynamically loads the object code when it is called in a query. The following query uses the *AgeInYears* procedure to retrieve the names and ages of all people in the example database:

```
retrieve (P.Name,
        Age = AgeInYears(P.Birthdate))
from P in PERSON*
```

User-defined procedures can also take tuple-variable arguments. For example, the following command defines a procedure, called *Comp*, that takes an EMPLOYEE tuple and computes the person's compensation according to some formula that involves several attributes in the tuple (e.g., the employee's status, job title, and salary):

```
define procedure Comp(EMPLOYEE)
    returns int4 is (language = "C",
    filename = "Comp1")
```

Recall that a parameterized procedure-type is defined for each relation automatically so the type *EMPLOYEE* represents a reference to a tuple in the *EMPLOYEE* relation. This procedure is called in the following query:

```
retrieve (E.Name, Compensation = Comp(E))
from E in EMPLOYEE
```

The C function that implements this procedure is passed a data structure that contains the names, types, and values of the attributes in the tuple.

User-defined procedures can be passed tuples in other relations that inherit the attributes in the relation declared as the argument to the procedure. For example, the *Comp* procedure defined for the *EMPLOYEE* relation can be passed a *STUDEMP* tuple as in

```
retrieve (SE.Name,
      Compensation = Comp(SE))
from SE in STUDEMP
```

because *STUDEMP* inherits data attributes from *EMPLOYEE*.

The arguments to procedures that take relation tuples as arguments must be passed in a self-describing data structure because the procedure can be passed tuples from different relations. Attributes inherited from other relations may be in different positions in the relations. Moreover, the values passed for the same attribute name may be different types (e.g., the definition of an inherited attribute may be overridden with a different type). The self-describing data structure is a list of arguments, one per attribute in the tuple to be passed, with the following structure

```
(AttrName, AttrType, AttrValue)
```

The procedure code will have to search the list to find the desired attribute. A library of routines is provided that will hide this structure from the programmer. The library will include routines to get the type and value of an attribute given the name of the attribute. For example, the following code fetches the value of the *Birthdate* attribute:

```
GetValue("Birthdate")
```

The problem of variable argument lists arises in all object-oriented programming languages and similar solutions are used.

The model for procedure inheritance is nearly identical to method inheritance in object-oriented programming languages [20]. Procedure inheritance uses the data inheritance hierarchy and similar inheritance rules except that a rule is provided to select a procedure when an inheritance conflict arises. For example, suppose that a *Comp* procedure was defined for *STUDENT* as well as for *EMPLOYEE*. The definition of the second procedure might be:

```
define procedure Comp(STUDENT)
    returns int4 is (language = "C",
    filename = "Comp2")
```

A conflict arises when the query on *STUDEMP* above is executed because the system does not know which *Comp* procedure to call (i.e., the one for *EMPLOYEE* or the one for *STUDENT*). The procedure called is selected from among the procedures that take a tuple from the relation specified by the actual argument *STUDEMP* or any relation from which attributes in the actual argument are inherited (e.g., *PERSON*, *EMPLOYEE*, and *STUDENT*).

Each relation has an *inheritance precedence list* (IPL) that is used to resolve the conflict. The list is constructed by starting with the relation itself and doing a depth-first search up the inheritance hierarchy starting with the first relation specified in the inherits-clause. For example, the inherits-clause for *STUDEMP* is

```
... inherits (STUDENT, EMPLOYEE)
```

and its IPL is

```
(STUDEMP, STUDENT,
 EMPLOYEE, PERSON)
```

PERSON appears after *EMPLOYEE* rather than after *STUDENT* where it would appear in a depth-first search because both *STUDENT* and *EMPLOYEE* inherit attributes from *PERSON* (see figure 1). In other words, all but the last occurrence of a relation in the depth-first ordering of the hierarchy is deleted.[4]

When a procedure is called and passed a tuple as the first argument, the actual procedure invoked is the first definition found with the same name when the procedures that take arguments from the relations in the ILP of the argument are searched in order. In the example above, the Comp procedure defined for *STUDENT* is called because there is no

[4] We are using a rule that is similar to the rule for the new Common Lisp object model [4]. It is actually slightly more complicated than described here in order to eliminate some nasty cases that arise when there are cycles in the inheritance hierarchy.

procedure named *Comp* defined for *STUDEMP* and *STUDENT* is the next relation in the IPL.

The implementation of this procedure selection rule is relatively easy. Assume that two system catalogs are defined:

```
PROCDEF(ProcName, ArgName, ProcId)
IPL(RelationName, IPLEntry, SeqNo)
```

where *PROCDEF* has an entry for each procedure defined and *IPL* maintains the precedence lists for all relations. The attributes in *PROCDEF* represent the procedure name, the argument type name, and the unique identifier for the procedure code stored in another catalog. The attributes in *IPL* represent the relation, an IPL entry for the relation, and the sequence number for that entry in the IPL of the relation. With these two catalogs, the query to find the correct procedure for the call

```
Comp(STUDEMP)
```

is[5]

```
retrieve (P.ProcId)
from P in PROCDEF, I in IPL
where P.ProcName = "Comp"
  and I.RelationName = "STUDEMP"
  and I.IPLEntry = P.ArgName
  and I.SeqNo = MIN(I.SeqNo
    by I.RelationName
    where I.IPLEntry = P.ArgName
      and P.ProcName = "Comp"
      and I.RelationName = "STUDEMP")
```

This query can be precomputed to speed up procedure selection.

In summary, the major changes required to support procedure inheritance is 1) allow tuples as arguments to procedures, 2) define a representation for variable argument lists, and 3) implement a procedure selection mechanism. This extension to the relational model is relatively straightforward and only requires a small number of changes to the DBMS implementation.

5. Other Data Models

This section compares the POSTGRES data model to semantic, functional, and object-oriented data models.

Semantic and functional data models [8,11,16,18,19,27] do not provide the flexibility provided by the model described here. They cannot easily represent data with uncertain structure (e.g., objects with shared subobjects that have different types).

Modeling ideas oriented toward complex objects [12,15] cannot deal with objects that have a variety of shared subobjects. POSTGRES uses procedures to represent shared subobjects which does not have limitation on the types of subobjects that are shared. Moreover, the nested-dot notation allows convenient access to selected subobjects, a feature not present in these systems.

Several proposals have been made to support data models that contain non-first normal form relations [3,7,9]. The POSTGRES data model can be used to support non-first normal form relations with procedure-types. Consequently, POSTGRES seems to contain a superset of the capabilities of these proposals.

Object-oriented data models [2,6] have modeling constructs to deal with uncertain structure. For example, GemStone supports union types which can be used to represent subobjects that have different types [6]. Sharing of subobjects is represented by storing the subobjects as separate records and connecting them to a parent object with pointer-chains. Precomputed procedure values will, in our opinion, make POSTGRES performance competitive with pointer-chain proposals. The performance problem with pointer-chains will be most obvious when an object is composed of a large number of subobjects. POSTGRES will avoid this problem because the pointer-chain is represented as a relation and the system can use all of the query processing and storage structure techniques available in the system to represent it. Consequently, POSTGRES uses a different approach that supports the same modeling capabilities and an implementation that may have better performance.

Finally, the POSTGRES data model could claim to be object-oriented, though we prefer not to use this word because few people agree on exactly what it means. The data model provides the same capabilities as an object-oriented model, but it does so without discarding the relational model and without having to introduce a new confusing terminology.

[5] This query uses a QUEL-style aggregate function.

6. Summary

The POSTGRES data model uses the ideas of abstract data types, data of type procedure, and inheritance to extend the relational model. These ideas can be used to simulate a variety of semantic data modeling concepts (e.g., aggregation and generalization). In addition, the same ideas can be used to support complex objects that have unpredicatable composition and shared subobjects.

References

1. M. E. Adiba and B. G. Lindsay, "Database Snapshots", *Proc. 6th Int. Conf. on Very Large Databases*, Montreal, Canada, Oct. 1980, 86-91.

2. T. Anderson and et. al., "PROTEUS: Objectifying the DBMS User Interface", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

3. D. Batory and et.al., "GENESIS: A Reconfigurable Database Management System", Tech. Rep. 86-07, Dept. of Comp. Sci., Univ. of Texas at Austin, 1986.

4. D. B. Bobrow and et.al., "COMMONLOOPS: Merging Lisp and Object-Oriented Programming", *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986, 17-29.

5. E. F. Codd, "A Relational Model of Data for Large Shared Data Bases", *Comm. of the ACM*, JUNE 1970.

6. G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proc. 1984 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1984.

7. P. Dadam and et.al., "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies", *Proc. ACM-SIGMOD Conf. on Mgt. of Data*, Washington, DC, May 1986.

8. U. Dayal and et.al., "A Knowledge-Oriented Database Management System", *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

9. U. Deppisch and et.al., "A Storage System for Complex Objects", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

10. H. Garcia-Molina and et.al., "DataPatch: Integrating Inconsistent Copies of a Database after a Partition", Tech. Rep. Tech. Rep.# 304, Dept. Elec. Eng. and Comp. Sci., Princeton, NJ, 1984.

11. M. Hammer and D. McLeod, "Database Description with SDM", *ACM-Trans. Database Systems*, Sep. 1981.

12. R. Haskins and R. Lorie, "On Extending the Functions of a Relational Database System", *Proc. 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, FL, JUNE 1982.

13. G. Held, M. R. Stonebraker and E. Wong, "INGRES -- A Relational Data Base System", *Proc. AFIPS NCC*, 1975, 409-416.

14. R. Kung and et.al., "Heuristic Search in Database Systems", *Proc. 1st International Workshop on Expert Data Bases*, Kiowah, SC, Oct. 1984.

15. R. Lorie and W. Plouffee, "Complex Objects and Their Use in Design Transactions", *Proc. Engineering Design Applications Stream of ACM-IEEE Data Base Week*, San Jose, CA, May 1983.

16. J. Myloupoulis and et.al., "A Language Facility for Designing Database Intensive Applications", *ACM-Trans. Database Systems*, JUNE 1980.

17. L. A. Rowe, "A Shared Object Hierarchy", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

18. D. Shipman, "The Functional Model and the Data Language Daplex", *ACM-Trans. Database Systems*, Mar. 1981.

19. J. Smith and D. Smith, "Database Abstractions: Aggregation and Generalization", *ACM Trans. Database Systems*, JUNE 1977.

20. M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations", *The AI Magazine 6*, 4

(Winter 1986), 40-62.

21. M. R. Stonebraker and et. al., "QUEL as a Data Type", *Proc. 1984 ACM-SIGMOD Conf. on the Mgt. of Data*, May 1984.

22. M. R. Stonebraker, "Triggers and Inference in Data Base Systems", *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

23. M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1986.

24. M. R. Stonebraker, "Object Management in POSTGRES Using Procedures", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

25. M. R. Stonebraker, "Inclusion of New Types in Relational Data Base Systems", *Proc. Second Int. Conf. on Data Base Eng.*, Los Angeles, CA, Feb. 1986.

26. M. R. Stonebraker, "POSTGRES Storage System", Submitted for publication, 1987.

27. C. Zaniola, "The Database Language GEM", *Proc. 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA., May 1983.

A RULE MANAGER FOR RELATIONAL DATABASE SYSTEMS

Michael Stonebraker, Eric Hanson and Spyros Potamianos

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper explains the rules subsystem that is being implemented in the POSTGRES DBMS. It is novel in several ways. First, it gives to users the capability of defining rules as well as data to a DBMS. Moreover, depending on the scope of each rule defined, optimization is handled differently. This leads to good performance both in the case that there are many rules each of small scope and a few rules each of large scope. In addition, rules provide either a forward chaining control flow or a backward chaining one, and the system will choose the control mechanism that optimizes performance in the cases that it is possible. Furthermore, priority rules can be defined, thereby allowing a user to specify rules systems that have conflicts. This use of exceptions seems necessary in many applications. Lastly, our rule system can provide database services such as views,protection, integrity constraints, and referential integrity simply by applying the rules system in a particular way. Consequently, no special purpose code need be included to handle these tasks.

1. INTRODUCTION

There has been considerable interest in integrating data base managers and software systems for constructing expert systems (e.g. KEE [INTE85], Prolog [CLOC81], and OPS5 [FORG81]). Although it is possible to provide interfaces between such rule processing systems and data base systems (e.g. [ABAR86, CERI86]), such interfaces will only perform well if the rule system can easily identify a small subset of the data to load into the working memory of the rule manager. Such problems have been called ``partitionable´´ Our interest is in a broad class of expert systems which are not partitionable.

An example of such a system would be an automated system for trading stocks on some securities exchange. The trading program would want to be alerted if a variety of data base conditions were true, e.g. any stock was trading excessively frequently, any stock or group of stocks was going up or down excessively

---

This research was sponsored by the National Science Foundation under Grant DMC-8504633 and by the Navy Electronics Systems Command under contract N00039-84-C-0039.

rapidly, etc. It is evident that the trading program does not have any locality of reference in a large data base, and there is no subset of the data base that can be extracted. Moreover, even if one could be identified, it would be out of date very quickly. For such problems, rule processing and data processing must be more closely integrated.

There are many mechanisms through which this integration can take place. In this paper we indicate a rather complete rules system which is quite naturally embedded in a general purpose data base manager. This next-generation system, POSTGRES, is described elsewhere [STON86a]; hence we restrict our attention in this paper solely to the rules component.

There are three design criteria which we strive to satisfy. First, we propose a rule system in which conflicts (or exceptions [BORG85]) are possible. The classic example is the rule ``all birds fly´´ along with the conflicting exception ``penguins are birds which do not fly´´ Another example of conflicting rules is the situation that all executives have a wood desk. However, Jones is an executive who uses a steel desk. It is our opinion that a rule system that cannot support exceptions is of limited utility.

The second goal of a rule system is to optimize processing of rules in two very different situations. First, there are applications where a large number of rules are potentially applicable at any one time, and the key performance issue is the time required to identify which rule or rules to apply. The automated stock trader is an example application of a rule system with a large number of rules each of narrow scope. Here, the system must be able to identify quickly which (of perhaps many) rules apply at a particular point in time. On the other hand, there are applications where the amount of optimization used in the processing of exceptionally complex rules is the key performance indicator. The rule whereby one derives the ANCESTOR relation from a base relation

PARENT (person, offspring)

is an example of this situation. Here, processing the rule in order to satisfy a user query to the ANCESTOR relation is the key task to optimize. A general purpose rules system must be able to perform well in both kinds of situations.

The third goal of a rules system embedded in a data manager should be to support as many data base services as possible. Candidates services include integrity control, referential integrity, transition constraints, and protection. As noted in [STON82], the code needed to perform these tasks correspond to small special purpose rules systems. A robust rules system should be usable for these internal purposes, and the POSTGRES rules system achieves this goal.

In Section 2 of this paper we discuss the syntax of POSTGRES rules and the semantics desired from a rule processing engine. Then, in Section 3 we discuss two optimization issues. First, the time at which a rule can be awakened can be varied, and provides a valuable opportunity for performance improvement. Secondarily, the mechanism that is used to ``fire´´ rules can be used at multiple granularities, and will be a second optimizationm possibility. Then in Section 4 we sketch the algorithms to be run at various times in rule processing. Lastly, Section 5 indicates how our rules system can be used to support views, protection, and integrity control subsystems.

2. POSTGRES RULE SEMANTICS

2.1. Syntax of Rules

POSTGRES supports a query language, POSTQUEL, which borrows heavily from its predecessor, QUEL [HELD75]. The main extensions are syntax to deal with procedural data, extended data types, rules, versions and time. The language is described elsewhere [STON86a, ROWE87], and here we give only one example to motivate our rules system. The following POSTQUEL command sets the salary of Mike to the salary of Bill using the standard EMP relation:

```
replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Mike'' and E.name = ``Bill''
```

POSTGRES allows any such POSTQUEL command to be tagged with three special modifiers which change its meaning. Such tagged commands become rules and can be used in a variety of situations as will be presently noted.

The first tag is ``always'' which is shown below modifying the above POSTQUEL command.

```
always replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Mike'' and E.name = ``Bill''
```

The semantics of this rule is that the associated command should logically appear to run forever. Hence, POSTGRES must ensure that any user who retrieves the salary of Mike will see a value equal to that of Bill. One implementation will be to wake up the above command whenever Bill's salary changes so the salary alteration can be propagated to Mike. This implementation resembles previous proposals [ESWA76, BUNE79] to support triggers, and efficient wake-up services are a challenge to the POSTGRES implementation. A second implementation will be to delay evaluating the rule until a user requests the salary of Mike. With this implementation, rules appear to utilize a form of ``lazy evaluation'' [BUNE82].

If a retrieve command is tagged with ``always'' it becomes a rule which functions as an alerter. For example, the following command will retrieve Mike's salary whenever it changes.

```
always retrieve (EMP.salary) where EMP.name = ``Mike''
```

The second tag which can be applied to any POSTQUEL command is ``refuse''. For example, the above retrieve command can be turned into this second kind of rule as follows:

```
refuse retrieve (EMP.salary) where EMP.name = ``Mike''
```

The semantics of a refuse command is that it should NEVER be run. Hence, if any subsequent request for Mike's salary occurs, POSTGRES should refuse to access it. More precisely, the semantics of any command with a refuse modifier is that the indicated operation cannot be done to any tuple which satisfies the qualification. For qualifications spanning more than one relation, the qualification is true if values for the tuple in question are substituted into the qualification and the result evaluates to true. Syntactically, append and delete commands do not contain a target list when tagged with ''refuse'', while replace and retrieve commands contain only a list of attributes.

Rules with a refuse modifier are generally useful for protection purposes; for example the following rule denies Bill access to Mike's salary.

```
refuse retrieve (EMP.salary) where EMP.name = ``Mike''
                    and user() = ``Bill''
```

In this command, user() is a POSTGRES function which returns the login name of the user who is running the current query. Commands with a refuse modifier are also useful for integrity control when tagged to update commands. For example, the following rule refuses to insert employees who earn more than 30000.

```
refuse append to EMP where EMP.salary > 30000
```

One final example illustrates integrity control using a refuse modifier. The following rule disallows the deletion of a department as long as there is at least one employee working in the department. This corresponds to one situation that arises in referential integrity [DATE81].

```
refuse delete DEPT where DEPT.dname = EMP.dept
```

The final tag which can be applied to a POSTQUEL command is the modifier ``one-time''. For example:

```
one-time replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Mike'' and E.name = ``Bill''
```

The semantics of this command is that it should be done exactly once when the qualification is true. In this case, the effect is exactly the same as if the command was submitted directly with no modifier. However, the following example shows the utility of this kind of rule in providing so-called ``one shots''.

```
one-time retrieve (EMP.salary) where EMP.name = ``Mike''
                    and time() >= ``April 15''
```

This command will be run once at some time subsequent to April 15th to retrieve Mike's salary.

There is great leverage in these three simple rule constructs. However the semantics of always and one-time commands present a problem as explored in the next subsection.

2.2. Semantics of Always and One-time Rules

Always and one-time rules share a common semantic problem which can be illustrated by the following rules that provide a salary for Mike.

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

always replace EMP (salary = E.salary) using E in EMP
where E.name = ``Bill''
and EMP.name = ``Mike''
```

There are several possible outcomes which might be desired from this collection of commands. The first option would be to reject this set of rules because it constitutes an attempt to assign two different values to the salary of Mike. Moreover, these two commands could be combined into a single POSTQUEL update, e.g.:

always replace EMP (salary = E.salary)
where EMP.name = ``Mike''
and (E.name = ``Bill'' or E.name = ``Fred'')

Such updates are **non-functional** and are disallowed by most data base systems (e.g INGRES [RTI85]) which detect them at run time and abort command processing. Hence the first semantics for always and onetime rules would be to demand functionality and refuse to process non-functional collections.

Of course functionality is not always desirable for a collection of rules. Moreover, as noted in [KUNG84], there are cases where non-functional updates should also be allowed in normal query processing. Hence, we now turn to other possible definitions for this rule collection.

The second definition would be to support **random** semantics. If both rules were run repeatedly, the salary of Mike would cycle between the salary of Bill and that of Fred. Whenever, it was set to one value the other rule would be run to change it back. Hence, a retrieve command would see one salary or the other depending on which rule had run most recently. With random semantics, the user should see one salary or the other, and POSTGRES should ensure that no computation time is wasted in looping between the values.

The third possibility would be to support **union** semantics for a collection of rules. Since POSTQUEL supports columns of a relation of data type procedure, one could define salary as a procedural field. Hence, commands in POSTQUEL would be the value of this field and would generate the ultimate field value when executed. In the salary field for Mike, the following two commands would appear:

retrieve (EMP.salary) where EMP.name = ``Bill''
retrieve (EMP.salary) where EMP.name = ``Fred''

If Mike's salary was retrieved, both Fred's salary and Bill's salary would be returned. Hence, when multiple rules can produce values, a user should see the union of what the rules produce if union semantics are used.

To support exceptions, one requires a final definition of the semantics of rules, namely **priority** semantics. In this situation, a priority order among the rules would be established by tagging each with a priority. Priorities are unsigned integers in the range 0 to 15, and may optionally appear at the end of a command, e.g:

always retrieve (EMP.salary) where EMP.name = ``Mike'' at priority = 7

If a priority is not specified by a user, then POSTGRES assumes a default of 0. When more than one rule can produce a value, POSTGRES should use the rule with highest priority. For example, suppose the priority for the ``Fred'' rule is 7 and for the ``Bill'' rule is 5. Using priority semantics the salary of Mike should be equal to the salary of Fred.

Since one of the goals of the POSTGRES rules systems is to support exceptions, we choose to implement priority semantics. Hence a user can optionally specify the relative priorities of any collection of tagged commands that he introduced and the highest priority rule will be used. If multiple rules have the same priority then POSTGRES chooses to implement random semantics for conflicting rules, and can return the result specified by any one of them.

In summary, POSTGRES will implement priority semantics and use the highest priority rule when multiple ones apply. Moreover, if multiple rules have the same priority, POSTGRES will use random semantics. It would have been possible (in fact easy) to insist on functional semantics. However, we feel that this is a less useful choice for rule driven applications.

Notice that collections of rules can be defined which produce a result which depends on the order of execution of the rules. For example, consider the following rules:

```
always delete EMP where EMP.salary = 1000

always replace EMP (salary = 2000)
where EMP.name = ``Mike''
```

If Mike receives a salary adjustment from 2000 to 1000, then the delete would remove him while the replace would change his salary back to 2000. The final outcome is clearly order sensitive. If these commands were run concurrently from an application program, then two outcomes are possible depending on which command happened to execute first. POSTGRES does not alter these semantics in any way. Hence, rules are awakened in a POSTGRES determined order, and the ultimate result may depend on the order of execution.

It is also possible for a user to define ill-formed rule systems, e.g.:

```
always replace EMP (salary = 1.1 * E.salary) using E in EMP
where EMP.name = ``Mike''
and E.name = ``Fred''

always replace EMP (salary = 1.1 * E.salary) using E in EMP
where EMP.name = ``Fred''
and E.name = ``Mike''
```

This set of rules says Fred makes 10 percent more than Mike who in turn makes 10 percent more than Fred. Clearly, these rules will never produce a salary for either Mike or Fred. In these situations, the goal of POSTGRES is to avoid going into an infinite loop. The algorithms we use are discusses in Sections 5 and 6.

We now turn to a discussion of the optimization tactics which POSTGRES employs.

3. OPTIMIZATION OF RULES

3.1. Time of Awakening of Always and Once Commands

Consider the following collection of rules:

```
always replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Mike''
and E.name = ``Bill''

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

Clearly Mike's salary must be set to Bill's which must be set to Fred's. If the

salary of Fred is changed, then the second rule can be awakened to change the salary of Bill which can be followed by the first rule to alter the salary of Mike. In this case an update to the data base awakens a collection of rules which in turn awaken a subsequent collection. This control structure is known as **forward chaining**, and we will term it early evaluation. The first option available to POSTGRES is to perform early evaluation of rules, and a forward chaining control flow will result.

A second option is to delay the awakening of either of the above rules until a user requests the salary of Bill or Mike. Hence, neither rule will be run when Fred's salary is changed. Rather, if a user requests Bill's salary, then the second rule must be run to produce it on demand. Similarly, if Mike's salary is requested, then the first rule is run to produce it requiring in turn the second rule to be run to obtain needed data. This control structure is known as **backward chaining**, and we will term it late evaluation. The second option available to POSTGRES is to delay evaluation of a rule until a user requires something it will write. At this point POSTGRES must produce the needed answer as efficiently as possible using an algorithm to be described in Section 5, and a backward chaining control flow will result.

Clearly, the choice of early or late evaluation has important performance consequences. If Fred's salary is updated often and Mike's and Bill's salaries are read infrequently, then late evaluation is appropriate. If Fred does not get frequent raises, then early evaluation may perform better. Moreover, response time to a request to read Mike's salary will be very fast if early evaluation is selected, while late evaluation will generate a considerably longer delay in producing the desired data. Hence, response time to user commands will be faster with early evaluation.

The choice of early or late evaluation is an optimization which POSTGRES will make internally in all possible situations. However, there are two important restrictions which limit the available options.

The first concerns indexing. Fields for which there are late rules cannot be indexed, because there is no way of knowing what values to index. Hence, a secondary index on the salary column of EMP cannot be constructed if there are any late rules which write salary data. On the other hand, early rules are compatible with indexes on fields which they update.

A second restriction concerns the mixing of late and early rules. Consider, for example, the situation where the Bill-to-Mike salary rule is evaluated early while the Fred-to-Bill salary rule is evaluated late. A problem arises when Fred receives a salary adjustment. The rule to propagate this adjustment on to Bill will not be awakened until somebody proposes to read Bill's salary. On the other hand, a request for Mike's salary will retrieve the old value because there is no way for the Bill-to-Mike rule to know that the value of Bill's salary will be changed by a late rule. To avoid this problem, POSTGRES must ensure that no late rules write any data objects read by early rules.

To deal with these two restrictions, POSTGRES takes the following precautions. Every column of a POSTGRES relation must be tagged as ``indexable'' or ``non-indexable''. Indexable columns cannot be written by late rules, while non-indexable columns permit late writes. To ensure that no late rule writes data read

by an early rule, POSTGRES enforces the restriction that early reads cannot access data from non-indexable columns. To support this, the POSTGRES parser produces two lists of columns, those in the target list to the left of an equals sign and those appearing elsewhere in the rule. These lists are the write-set and read-set respectively for a rule. If the read-set contains an indexable field, we tag the rule ``read I''. Similarly, a rule that writes an indexed field is tagged ``write I''. For non-indexed fields, the corresponding tags are ``read NI'' and ``write NI''. Table 1 shows the allowable execution times for the various rule tags. The consequences of Table 1 are that some rules are not allowable, some must be evaluated early, some must be evaluated late, and some can be evaluated at either time. This last collection can be optimized by POSTGRES. In a well designed data base we expect most rules to read indexed fields for fast access. Hence, if they write non-indexable fields they are optimizable.

To achieve further optimization, POSTGRES can temporarily change the time of evaluation of any late rule to ``temporarily early'' if the rule does not read any data written by a late rule. Similarly, an early rule can be changed to temporarily late if it does not write an indexed field or an object read by an early rule. If at some subsequent time these conditions become false, then the rule must revert from its temporary status back to its permanent status.

An unfortunate consequence of Table 1 is that permanent status of all inserts and deletes is early, since all relations will have at least one indexable field. Moreover, we will make no effort in the initial implementation to support moving either kind of command to temporarily late.

Within these constraints and considerations, POSTGRES will attempt to optimize the early versus late decision on a rule by rule basis. Not only will a decision be made when a rule is first inserted, but also an asynchronous demon, REVEILLE/TAPS (Rule EValuation EIther earLy or LatE for the Trigger Application Performance System), will run in background to make decisions on which rules should be converted temporarily or permanently from late to early execution

rule status	Time of Awakening
read I write NI	early or late
read NI write I	not permitted
read I write I	early
read NI write NI	late

Time of Rule Awakening

Table 1

and vice-versa. The architecture of REVEILLE/TAPS is currently under investigation.

3.2. Granularity of Locking for Refuse and Always Rules

POSTGRES must wake-up rules at appropriate times and perform specific processing with them. In [STON86b] we analyzed the performance of a rule indexing structure and various structures based on physical marking (locking) of objects. When the average number of rules that covered a particular tuple was low, locking was preferred. Moreover, rule indexing could not be easily extended to handle rules with join terms in the qualification. Because we expect there will be a small number of rules which cover each tuple in practical applications, we are utilizing a locking scheme.

When a rule is installed into the data base for either early or late evaluation, POSTGRES is run in a special mode and sets appropriate locks at the individual attribute level or at the tuple level. There are a total of 13 kinds of locks which will be detailed in the next section. These locks differ from normal read and write locks in several ways. First, normal locks are set and released at high frequency and exist in relatively small numbers. When a crash occurs, the lock table is not needed because recovery can be accomplished solely from the log. Hence, virtually all systems utilize a main memory lock table for normal locks. On the other hand, locks set by rules exist in perhaps vast numbers since POSTGRES must be prepared to accommodate a large collection of rules. Secondly, locks are set and reset at fairly low frequency. They are only modified when rules are inserted, deleted, their time of evaluation is changed, or in certain other cases to be explained. Lastly, if a crash occurs one must not lose the locks set by rules. The consequences of losing rule locks is the requirement that they be reinstalled in the data base and recovery time will become unacceptably long. As a result, rule locks must persist over crashes.

Because of these differences, we are storing rule locks as normal data in POSTGRES tuples. This placement has a variety of advantages and a few disadvantages. First, they are automatically persistent and recoverable and space management for a perhaps large number of locks is easily dealt with. Second, since they are stored data, POSTGRES queries can be run to retrieve their values. Hence, queries can be run of the form ``If I update Mike's salary, what rules will be affected?'' This is valuable in providing a debugging and query environment for expert system construction. The disadvantage of storing the locks on the data records is that setting or resetting a lock requires writing the data page. Hence, locks associated with rules are expensive to set and reset.

Like normal locks, there is a phantom problem to contend with. For example, consider the rule to set Mike's salary to be the same as Bill's. If Bill is not yet an employee, then the rule has no effect. However, when Bill is hired, the rule must be awakened to propagate his salary. Setting locks on tuples and attributes will not accomplish the desired effect because one can only lock actual data read or written. To deal with phantoms, POSTGRES also sets rule locks on each index record that is read during query processing and on a ``stub record'' which it inserts in the index to denote the beginning and end of a scan. Whenever a data record is inserted into a POSTGRES relation, appropriate index records must be

added to each existing secondary index. The POSTGRES run time system must note all locks held on index records which are adjacent to any inserted secondary index record. Not only must these locks be inherited by the corresponding data record, but also they must be inherited by the secondary index record itself. The above mechanism must be adjusted slightly to work correctly with hashed secondary indexes. In particular, a secondary index record must inherit all locks in the same hash bucket. Hence, ``adjacent'' must be interpreted to mean ``in the same hash bucket''. This mechanism is essentially the same one used by System R to detect phantoms. Although cumbersome and somewhat complex, it appears to work and no other alternative is readily available. Since POSTGRES supports user-defined secondary indexes [STON86d], this complexity must be dealt with by index code written by others.

Locks may be set at attribute or record level granularity as noted above. However, there are situations where lock escalation may be desirable. For example, consider the rule:

always replace EMP (salary = avg (EMP.salary where EMP.dept = ``shoe''))
where EMP.name = ``Mike''

This rule will read the salaries of all shoe department employees to compute the aggregate. Rather than setting a large number of attribute or record level locks, it may be preferable to escalate to a relation level lock. Hence, all rule locks can also be set at the relation level. In this case they become tuple level locks set on the tuple in the RELATION relation which exists for the particular relation to be locked. A lock can be set only on a column of a relation by setting a tuple level lock on the appropriate row in the ATTRIBUTE relation.

POSTGRES will choose either fine granularity or coarse granularity as an optimization issue. It can either escalate after it sets too many fine granularity locks or guess at the beginning of processing based on heuristics. The current wisdom for conventional locks is to escalate after a certain fixed number of locks have been set [GRAY78, KOOI82]. For simplicity in the first implementation, POSTGRES will guess one granularity for the rule in advance and set either record or table level locks for the rule. The extension to multiple concurrent granularities is left as a future enhancement.

The decision on lock granularity in this new context has a crucial performance implication. In particular, one does not know what record level locks will be observed during the processing of a query plan until specific tuples are inspected. Hence, if late evaluation is used, one or more additional queries may be run to produce values needed by the user query. Consequently, in addition to the user's plan, N extra plans must be run which correspond to the collection of N late rules that are encountered. These N+1 queries are all optimized separately when record level locks are used. Moreover, these plans may awaken other plans which are also independently optimized.

On the other hand, if all locks are escalated to the relation level, the query optimizer knows what late rules will be utilized and can generate a composite optimized plan for the command as discussed in Section 6. This composite plan is very similar to what is produced by query modification [STON75] and is a simplified version of the sort of processing in [ULLM85]. It will sometimes result in a more efficient total execution. However, setting relation level locks has an

important performance disadvantage. For example, if the rules noted earlier that set Mike's and Bill's salaries are escalated to the relation level, then ALL incoming commands will use the rules whether or not they read Mike's or Bill's salary. This will result in considerable wasted overhead in using rules which don't apply. Like the decision of early versus late evaluation, the decision of lock granularity is a complex optimization problem. Initial investigation [HONG87] suggests that record level locking is preferred in a large variety of cases; however a more detailed study is underway.

Unfortunately, there appears to be no way to prioritize two commands which lock at different granularities. Hence, priorities can only be established for collections of table locking rules or record locking rules.

4. SETTING LOCKS

4.1. Introduction

POSTGRES rules are supported by setting various kinds of locks as noted in the previous section. One-time rules are the same as always rules except that there is an automatic deletion of the rule when a successful firing takes place. The only special case code required for one-time commands pertains to ones which have a time clause present. For those, POSTGRES will perform an insert into a calendar relation and have a system demon which will wake up periodically and see if there are rules in calendar to awaken. Consequently, we will concentrate on always and refuse rules.

When an early rule is installed, it must set early read and early write locks on all objects that it reads and writes respectively. Moreover, late rules must set similar late read and late write locks. However, it will be desirable to distinguish three different kinds of read locks for the following three situations.

Consider the rule which propagates Fred's salary on to Bill, i.e:

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

If this rule is evaluated early and Fred's salary changes, then this rule must be awakened to propagate the change on to Bill. Clearly, no new objects will be read or written because of this salary adjustment. Hence, the recalculation of Bill's salary is the only task which must be accomplished, and no locks will change. Fred's salary field will be marked with an R1 lock to indicate this cheapest mode of rule wake-up.

On the other hand, suppose that Bill does not exist as an employee yet. Obviously, this rule will not be able to give Bill a salary. However, at the time he is inserted, the rule must be awakened to give him a salary. In this case, the rule must be run but the only locks affected will be on the tuple just inserted. This second wake-up mode is indicated by placing an R2 lock on the name of Bill. Lastly, if Fred is not yet an employee, then clearly the rule cannot propagate a salary on to Bill. When Fred is inserted, the rule must wake up to do the appropriate salary modification and must also set locks on records in the data base other than the one just updated. This third wake-up mode is indicated by placing

an R3 lock on the name of Fred.

As a result, always commands can set the following locks:

ER1: early read lock -- cheapest wake-up
ER2: early read lock -- more expensive wake-up
ER3: early read lock -- most expensive wake-up
EW : early write lock
LR1: late read lock -- cheapest wake-up
LR2: late read lock -- more expensive wake-up
LR3: late read lock -- most expensive wake-up
LW : late write lock

Refuse rules will set late read locks in the same way as always commands. However, they must also set a special kind of write lock on objects they would propose to change. These locks are:

RR: refuse retrieve
RA: refuse append
RD: refuse delete
RU: refuse update
RE: refuse execute

The next three subsections discusses now these 13 kinds of locks get set.

4.2. Set-up Needed

When a refuse or always command is entered by a user, the query tree for the new rule must be decorated with a read marker or a write marker on certain nodes. For each node which corresponds to an attribute in some relation, the parser must place markers as follows:

read markers:

R1: attributes on right hand side of an assignment in the target list
R2: any attribute in the qualification with the same tuple variable as the relation being updated
R3: other attributes in qualification

write markers:

W : all attributes on left hand side of a target list assignment for always commands
RA: the relation affected for refuse append command
RD: the relation affected for refuse delete commands
RE: all attributes in the target list for refuse execute commands
RR: all attributes in the target list for refuse retrieve commands
RU: all attributes in the target list for refuse replace commands

If a field name appears more than once in the qualification then each marker must identify the particular node in the tree that it is associated with.

Lastly, the parser must tag the rule with ``early'' ``late'' ``either'' or return an error message according to Table 1 of the previous section.

4.3. Insertion of Rules

REVEILLE/TAPS will make the early/late decision for always commands with a status of ``either''. and the lock granularity decision for all rules. If a complete scan of any relation is done, table level locking will be used. Otherwise, REVEILLE/TAPS can freely choose the granularity. Then, POSTGRES will insert an entry into a system relation holding rules and change the decorations in the parse tree to EW, ER1, ER2, and ER3 for early rules and LW, LR1, LR2, and LR3 for late rules. The command will now be optimized and then executed normally. During each scan of a relation, the attributes being accessed will be identified in the plan. Hence, a marker for each attribute along with its attribute number and the rule identifier can be packaged into a ``lock structure'' In addition, the lock structure must include the rule priority for write locks. If relation granularity has been chosen, then this lock structure will be placed in the RELATION relation tuple for this particular relation. Moreover, if early evaluation is used, then the rule will be run to update appropriate data values. First, the negation of all the higher priority rules must be ANDed onto the rule qualification.

If record level granularity has been selected, the lock structure will be put on each tuple accessed in the secondary index used in the scan. Additionally, a ``stub record'' will be inserted in the index at each end of the scan giving an ``end of scan'' marker and the data value of the end of the scan. Lastly, the read locks in the lock structure will be placed on each data tuple accessed independent of whether it actually satisfies the qualification. In addition, the write locks in the lock structure are placed on the data records that would actually be updated by the rule. However, if the rule being processed is a refuse command or an always command with late execution, write locks are installed but no updates of data records are actually performed, and the insertion of the rule is now complete. If the rule is an always command with early execution, POSTGRES must calculate the proposed data values and place them in the data records if there is no higher priority EW lock already on this field.

4.4. Deletion of a Rule

To delete a rule, the run-time system must execute the rule in a special mode to find all the read and write locks set on behalf of the rule. Then, it must update all such data and index records to remove the locks. Finally, other rules with EW locks on fields written by the deleted rule must be awakened.

5. RECORD LEVEL LOCK PROCESSING

The execution routines in POSTGRES must perform certain actions when a tuple is retrieved, modified, deleted, inserted or executed. These actions make use of a common module called the ``rule manager'' where much of the algorithm resides. We discuss the tuple level routines followed by the rule manager.

5.1. Tuple Processing

When a tuple is inserted, the appropriate keys must be inserted into all secondary indexes. These secondary index records plus the data record must inherit all appropriate lock structures as noted in the previous section. Now the tuple with all its proposed lock structures should be passed to the rule manager.

When a tuple is to be deleted, the tuple together with all its locks will be passed to the rule manager for processing. When a collection of fields in a tuple are retrieved or executed, the appropriate fields and their lock structures must be passed to the rule manager.

When a tuple is modified, all the changes must be installed in the appropriate secondary indexes and new locks must be inherited as in the case of insertions. In addition, all lock structures that were deleted by the index deletions must be noted. A data structure will be passed to the rule manager consisting of:

the old values of the updated fields
the locks to be deleted from the updated fields
the new values of the updated fields
the continuing locks on the updated fields
the locks to be added to the updated fields
the fields which are not being updated

5.2. The Rule Manager

The rule manager processes inserted, deleted, retrieved, executed, and replaced tuples and returns a revised tuple or an error message to the execution routine. For inserts and deletes, it looks at all fields. For each one with a lock, it does the action indicated in Tables 2 and 3 below. For retrieves and executes, it looks only at the fields retrieved or executed, and does the action indicated in the tables below. For replaces, things are a bit more complex. It should process the refuse replace locks first according to Table 2. Then, it should process all the continuing locks on the updated fields according to the replace column in Table 3. The last step is to process the new locks and the no longer valid locks using the append and delete columns respectively in Table 3.

In Table 3 there are no actions to take when LR1 or LR2 locks are observed; hence their is no row for them and they need never be set. In Tables 2 and 3, the symbols have the following meaning:

Refuse-Lock	retrieve	execute	replace	delete	append
RR	a				
RE		a			
RU			a		
RD				a	
RA					a

Actions for Refuse Locks

Table 2

Always-Lock	retrieve	execute	delete	append	replace
EW				b	c
LW	d	d			
ER1			e	e	f
ER2					g
ER3 or LR3			h	i	j

Actions for Always Locks

Table 3

a: Generate an error message for the executor if the tuple satisfies the qualification.

b: Check if the tuple actually satisfies the rule. If not remove the EW lock. Take the value returned by the highest priority rule and put it in the tuple. If the highest priority rule is a delete, then remove the tuple.

c: Refuse the offered value unless it is made on behalf of the rule holding the lock or a higher priority rule.

d: substitute the current tuple into the query plan for the rule and run the rule as a retrieve command. Take the first returned value and plug it into the tuple as a value, thereby implementing random semantics. For example, consider a query to retrieve the salary of Bill and a late rule that ensures Bill's salary is the same as that of Fred, i.e.:

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

In this case the user read of the salary field will conflict with the LW lock from the rule. The rule will be turned into the following retrieve command:

retrieve (salary = E.salary)
where ``Bill'' = ``Bill'' and E.name = ``Fred''

The salary of the first Fred to be returned is placed in the record returned by the rule manager.

e: All records that have an ER1 lock must have an ER3 lock elsewhere in the tuple. In the case that a delete or insert occurs, the field having an ER3 lock will also be deleted or inserted and the processing appropriate to that stronger lock will have precedence.

f: Substitute the proposed tuple into the rule and run it as a normal command to update appropriate data items.

g: Substitute the new value of the tuple into the rule and see if the rule evaluates

to true. If not remove the EW locks for the fields in this tuple associated with the ER2 lock. Execute step b: to find a replacement value for the field.

h: In this case some locks may have to be deleted. Hence, substitute the values for the current tuple into the rule, add on the qualification

and object-identifier = ``this-tuple''

and execute it in ``rule deletion'' mode to find the locks to delete. The second step is to reinsert locks on data items that can be found from duplicates of the deleted data item. To perform this function, the rule should be run in ``rule insertion'' mode with the the following qualification appended:

and object-identifier not equal ``this tuple''

For example, consider the Fred-to-Bill salary rule above and suppose that Fred is deleted. The first step is to run the following command in rule deletion mode:

always replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Bill''
and ``Fred'' = ``Fred'' and E.OID = ``Fred's OID''

The second step is to run the following command in rule insertion mode.

always replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Bill''
and E.name = ``Fred'' and E.OID != ``Fred's OID''

i: In this case some locks may have to be inserted. Hence, substitute the new tuple into the rule and execute it in ``rule insertion'' mode. Place locks and data values in records as appropriate.

j: Do both h: and i:

The transformations in i: and j: can be performed in parallel with processing the remainder of the query as long as the rule runs with an effective command identifier which is the same as the current command. This will ensure that the command does not see any of the modifications performed by rule processing. The details of why the POSTGRES storage system supports this parallelism are contained in [STON87a]. Alternatively, these modifications can be executed at the conclusion of a user command by saving them in virtual memory or in a file. If the user command writes data on a substantial number of fields holding ER3 or LR3 locks belonging to a single rule, then it may be advantageous to simply delete and reinstall the complete rule. In the first implementation we will process modifications synchronously at the end of a command, leaving the other options as future optimizations.

If both read and write locks are held on a single field by different rules, then care must be exercised concerning the order of execution. The rule manager must construct a dependency graph to control processing order. In this graph an arc is placed from any rule holding a LW lock on a field to all the rules holding LR1, LR2 or LR3 locks. If this graph is a tree, then process the rules from root to leaf.

If the graph is not a tree, then the rules involved in the loop are probably not well formed, and an error message will be signaled.

6. PROCESSING RELATION LEVEL LOCKS

When POSTGRES begins to process a user command which involves a relation R, it must process all the locks held at the relation level on R. To do so, it checks whether the proposed command is reading or writing any field on which a rule holds a lock and uses Tables 4 and 5 to resolve the conflict: In Tables 4 and 5 the symbols denote the following actions:

k: Add the negation of the rule qualification to the query qualification and continue.

l: The action to take is a little different depending on whether the rule holding the EW lock is an append, replace or delete command. If it is an append, then do nothing. If it is a delete, then AND the negation of the delete qualification to the

Refuse-Lock	retrieve	execute	replace	delete	append
RR	k				
RE		k			
RU			k		
RD				k	
RA					k

Table Level Refuse Locks

Table 4

Always-Lock	retrieve	execute	delete	append	replace
EW			l	l	l
LW	m	m			
ER1			n	n	n
ER2			n	n	n
ER3			n	n	n

Table Level Always Locks

Table 5

user's command. If it is a replace, then two commands must be run. The first one results from ANDing the rule qualification onto the command and replacing appropriate fields in the user's target list with target list entries from the rule. The second command results from ANDing the negation of the rule qualification to the user's command. When multiple EW locks occur, process the highest priority one first. Then proceed iteratively with the next highest one, applying it to the modified command for deletes and to the second command resulting from replace rules.

m: Since only replace commands can hold LW locks, the action to take here is to run two commands. The first results by ANDing the rule qualification to the user retrieval and substituting the rule target list for appropriate elements of the user's target list. The second command results from ANDing the negation of the rule qualification onto the user command.

n: Wake up the rule after the user qualification has been ANDed onto it to refresh its values.

When both read and write locks are held on a column of a relation by different rules, then care must again be exercised in choosing the order of rule evaluation. Construct a dependency graph as in the previous section and process the rules in the appropriate order. If the graph is not a tree, signal an error.

7. DATA BASE SERVICES

7.1. Views

POSTGRES supports updatable views using procedural fields as explained in [STON87b]. However, the rules system can be used to construct two other kinds of views, partial views, and read-only views. A read-only view is specified by creating a relation, say VIEW, and then defining the rule:

always retrieve into VIEW (any-target-list)
where any-qualification

This rule can be executed either early or late if all accessed fields are indexable. Otherwise, the permanent status of the rule is late and REVEILLE/TAPS may temporarily move it to early if no other rule performs late writes on data this rule reads. Late evaluation leads to conventional view processing by query modification, while early evaluation will cause the view to be physically materialized. In this latter case, updates to the base relation will cause the materialization to be invalidated and excessive recomputation of the whole view will be required. In the future we hope to avoid this recomputation and instead incrementally update the result of the procedure. The tactics of [BLAK86] are a step in this direction.

On the other hand, partial views are relations which have a collection of real data fields and additionally a set of fields which are expected to be supplied by rules. Such views can be specified by as large a number of rules as needed. Moreover, priorities can be used to resolve conflicts. As a result partial views can be utilized to define relations which are impossible with a conventional view

mechanism. Such extended views have some of the flavor proposed in [IONN84].

Moreover, all retrieves to such relations function correctly. Updates to such relations are processed as conventional updates which install actual data values in their fields, as long as all the rules are evaluated late.

7.2. Integrity Control

Integrity control is readily achieved by using delete rules. For example the following rule enforces the constraint that all employees earn more than 3000:

delete always EMP where EMP.salary < 3000

Since this is an early rule, it will be awakened whenever a user installs an overpaid employee and the processing is similar to that of current integrity control systems [STON75].

Referential integrity is easily accomplished using the mechanisms we have defined. The modes that refuse insertions and deletions can be accomplished with refuse rules as noted in Section 2.1. The other modes can all be accomplished using always rules.

7.3. Protection

Protection is normally specified by refuse rules which have a user() in the qualification. The only abnormal behavior exhibited by this application of the rules system is that the system defaults to ``open access´´. Hence, unless a rule is stated to the contrary, any user can freely access and update all relations. Although a cautious approach would default to ``closed access´´, it is our experience that open access is just as reasonable.

A useful future extension would be a rule which hides data items by returning an incorrect value. For example, consider the following rule:

hide EMP (salary = 0)
where EMP.name = ``Mike´´
and user() = ``Sam´´

This rule should be evaluated just like a refuse rule except it must return the value in its qualification instead of the one in the data record. This would allow the protection system to lie to users, rather than simply allow or decline access to objects. Such a facility allows greatly expanded capabilities over ordinary protection systems.

8. CONCLUSIONS

This paper has presented a rules system with a considerable number of advantages. First, the rule system consists of tagged query language commands. Since a user must learn the query language anyway, there is marginal extra complexity to contend with. In addition, specifying rules as commands which run indefinitely appears to be an easy paradigm to grasp. Moreover, rules may conflict and a priority system can be used to specify conflict resolution.

Two different optimizations were proposed for the implementation. The first optimization concerns the time that rules are evaluated. If they are evaluated early, then a forward chaining control flow results, while late evaluation leads to

backward chaining. Response time considerations, presence or absence of indexes, and frequency of read and write operations will be used to drive REVEILLE/TAPS which will decide on a case by case basis whether to use early evaluation. Study of the organization of this module is underway. In addition, the locking granularity can be either at the tuple level or at the relation level. Tuple level locking will optimize the situation where a large number of rules exist each with a small scope. Finding the one or ones that actually apply from the collection that might apply is efficiently accomplished. On the other hand, relation level locking will allow the query optimizer to construct plans for composite queries, and more efficient global plans will certainly result. Hence, we accomplish our objective of designing a rule system which can be optimized for either case. Lastly, the rule system was shown to be usable to implement integrity control, a novel protection system and to support access to two different kinds of views.

However, much work remains to be done. Optimizing the updating of locks when data items change is complex and possibly slow. Deleting and reinserting locks should be optimized better. Moreover, the implementation is complex and difficult to understand. Hence, a simpler implementation would be highly desirable. In general, a mechanism to update the result of a procedure is required rather than simply invalidating it and recomputing it. The efforts of [BLAK86] are a start in this direction, and we expect to search for algorithms appropriate to our environment. Moreover, it is a frustration that the rule system cannot be used to provide view update semantics. The general idea would be to provide a rule to specify the mapping from base relations to the view and then another rule(s) to provide the reverse mapping. Since it is well known that non-invertible view definitions generate situations where there is no unambiguous way to map backward from the view to base relations, one must require an extra semantic definition of what this inverse mapping should be. We hope to extend our rules system so it can be used to provide both directions of this mapping rather than only one way. Lastly, we are searching for a clean and efficient way to eliminate the annoying restrictions of our rule system, including the fact that priorities cannot be used with different granularity rules, and some rules are forced to a specific time of awakening.

REFERENCES

[ABAR86] Abarbanel, R. and Williams, M., ``A Relational Representation for Knowledge Bases,'' Proc. 1st International Conference on Expert Database Systems, Charleston, S.C., April 1986.

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

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

[BUNE79] Buneman, P. and Clemons, E., ``Efficiently Monitoring Relational Data Bases,'' ACM-TODS, Sept. 1979.

[BUNE82] Buneman, P. et. al., ``An Implementation Technique for Database Query Languages,'' ACM-TODS, June 1982.

[CERI86] Ceri, S. et. al., ``Interfacing Relational Databases and Prolog Efficiently,'' Proc 1st International Conference on Expert Database Systems, Charleston, S.C., April 1986.

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

[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.

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

[GRAY78] Gray, J., ``Notes on Data Base Operating Systems,'' IBM Research, San Jose, Ca., RJ 2254, August 1978.

[HELD75] Held, G. et. al., ``INGRES: A Relational Data Base System,'' Proc 1975 National Computer Conference, Anaheim, Ca., June 1975.

[HONG87] Hong, C., ``An Analysis of Rule Locking Granularities,'' Master's Report, Computer Science Division, University of California, Berkeley, Ca., 1987.

[INTE85] IntelliCorp, ``KEE Software Development System User's Manual,'' IntelliCorp, Mountain View, Ca., 1985.

[IONN84] Ionnidis, Y. et. al., ``Enhancing INGRES with Deductive Power,'' Proceedings of the 1st International Workshop on Expert Data Base Systems, Kiowah SC, October 1984.

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

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

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

[ROWE87] Rowe, L. and Stonebraker, M., ``The POSTGRES Data Model,'' (submitted for publication).

[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.

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

[STON86b] Stonebraker, M. et. al., ``An Analysis of Rule Indexing Implementations in Data Base Systems,'' Proc. 1st International Conference on Expert Data Base Systems, Charleston, S.C., April 1986.

[STON86c] Stonebraker, M., ``Object Management in POSTGRES using Procedures,'' Proc. 1986 International Workshop on Object-oriented Database Systems, Asilomar, Ca., Sept 1986. (available from IEEE)

[STON86d] Stonebraker, M., ``Inclusion of New Types in Relational Data Base Systems,'' Proc. IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1986.

[STON87a] Stonebraker, M., ``The POSTGRES Storage System,'' (submitted for publication).

[STON87b] Stonebraker, M. et. al., ``Extending a Relational Data Base System with Procedures,'' ACM-TODS (to appear).

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

THE DESIGN OF THE POSTGRES STORAGE SYSTEM

Michael Stonebraker

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper presents the design of the storage system for the POSTGRES data base system under construction at Berkeley. It is novel in several ways. First, the storage manager supports transaction management but does so without using a conventional write ahead log (WAL). In fact, there is no code to run at recovery time, and consequently recovery from crashes is essentially instantaneous. Second, the storage manager allows a user to optionally keep the entire past history of data base objects by closely integrating an archival storage system to which historical records are spooled. Lastly, the storage manager is consciously constructed as a collection of asynchronous processes. Hence, a large monolithic body of code is avoided and opportunities for parallelism can be exploited. The paper concludes with a analysis of the storage system which suggests that it is performance competitive with WAL systems in many situations.

1. INTRODUCTION

The POSTGRES storage manager is the collection of modules that provide transaction management and access to data base objects. The design of these modules was guided by three goals which are discussed in turn below. The first goal was to provide transaction management without the necessity of writing a large amount of specialized crash recovery code. Such code is hard to debug, hard to write and must be error free. If it fails on an important client of the data manager, front page news is often the result because the client cannot access his data base and his business will be adversely affected. To achieve this goal, POSTGRES has adopted a novel storage system in which no data is ever overwritten; rather all updates are turned into insertions.

The second goal of the storage manager is to accomodate the historical state of the data base on a write-once-read-many (WORM) optical disk (or other archival medium) in addition to the current state on an ordinary magnetic disk. Consequently, we have designed an asynchronous process, called the vacuum cleaner

This research was sponsored by the Navy Electronics Systems Command under contract N00039-84-C-0039.

which moves archival records off magnetic disk and onto an archival storage system.

The third goal of the storage system is to take advantage of specialized hardware. In particular, we assume the existence of non-volatile main memory in some reasonable quantity. Such memory can be provide through error correction techniques and a battery-back-up scheme or from some other hardware means. In addition, we expect to have a few low level machine instructions available for specialized uses to be presently explained. We also assume that architectures with several processors will become increasingly popular. In such an environment, there is an opportunity to apply multiple processors to running the DBMS where currently only one is utilized. This requires the POSTGRES DBMS to be changed from the monolithic single-flow-of-control architectures that are prevalent today to one where there are many asynchronous processes concurrently performing DBMS functions. Processors with this flavor include the Sequent Balance System [SEQU85], the FIREFLY, and SPUR [HILL85].

The remainder of this paper is organized as follows. In the next section we present the design of our magnetic disk storage system. Then, in Section 3 we present the structure and concepts behind our archival system. Section 4 continues with some thoughts on efficient indexes for archival storage. Lastly, Section 5 presents a performance comparison between our system and that of a conventional storage system with a write-ahead log (WAL) [GRAY78].

2. THE MAGNETIC DISK SYSTEM

2.1. The Transaction System

Disk records are changed by data base transactions, each of which is given a unique transaction identifier (XID). XIDs are 40 bit unsigned integers that are sequentially assigned starting at 1. At 100 transactions per second (TPS), POSTGRES has sufficient XIDs for about 320 years of operation. In addition, the remaining 8 bits of a composite 48 bit interaction identifier (IID) is a command identifier (CID) for each command within a transaction. Consequently, a transaction is limited to executing at most 256 commands.

In addition there is a transaction log which contains 2 bits per transaction indicating its status as:

committed
aborted
in progress

A transaction is started by advancing a counter containing the first unassigned XID and using the current contents as a XID. The coding of the log has a default value for a transaction as "in progress" so no specific change to the log need be made at the start of a transaction. A transaction is committed by changing its status in the log from "in progress" to "committed" and placing the appropriate disk block of the log in stable storage. Moreover, any data pages that were changed on behalf of the transaction must also be placed in stable storage. These pages can either be forced to disk or moved to stable main memory if any is available. Similarly, a transaction is aborted by changing its status from "in progress" to "aborted".

The tail of the log is that portion of the log from the oldest active transaction up to the present. The body of the log is the remainder of the log and transactions in this portion cannot be "in progress" so only 1 bit need be allocated. The body of the log occupies a POSTGRES relation for which a special access method has been built. This access method places the status of 65536 transactions on each POSTGRES 8K disk block. At 1 transaction per second, the body increases in size at a rate of 4 Mbytes per year. Consequently, for light applications, the log for the entire history of operation is not a large object and can fit in a sizeable buffer pool. Under normal circumstances several megabytes of memory will be used for this purpose and the status of all historical transactions can be readily found without requiring a disk read.

In heavier applications where the body of the log will not fit in main memory, POSTGRES applies an optional compression technique. Since most transactions commit, the body of the log contains almost all "commit" bits. Hence, POSTGRES has an optional bloom filter [SEVR76] for the aborted transactions. This tactic compresses the buffer space needed for the log by about a factor of 10. Hence, the bloom filter for heavy applications should be accomodatable in main memory. Again the run-time system need not read a disk block to ascertain the status of any transaction. The details of the bloom filter design are presented in [STON86].

The tail of the log is a small data structure. If the oldest transaction started one day ago, then there are about 86,400 transactions in the tail for each 1 transaction per second processed. At 2 bits per entry, the tail requires 21,600 bytes per transaction per second. Hence, it is reasonable to put the tail of the log in stable main memory since this will save the pages containing the tail of the log from being forced to disk many times in quick succession as transactions with similar transaction identifiers commit.

2.2. Relation Storage

When a relation is created, a file is allocated to hold the records of that relation. Such records have no prescribed maximum length, so the storage manager is prepared to process records which cross disk block boundaries. It does so by allocating continuation records and chaining them together with a linked list. Moreover, the order of writing of the disk blocks of extra long records must be carefully controlled. The details of this support for multiblock records are straightforward, and we do not discuss them further in this paper. Initially, POSTGRES is using conventional files provided by the UNIX operating system; however, we may reassess this decision when the entire system is operational. If space in a file is exhausted, POSTGRES extends the file by some multiple of the 8K page size.

If a user wishes the records in a relation to be approximately clustered on the value of a designated field, he must declare his intention by indicating the appropriate field in the following command

cluster rel-name on {(field-name using operator)}

POSTGRES will attempt to keep the records approximately in sort order on the field name(s) indicated using the specified operator(s) to define the linear ordering. This will allow clustering secondary indexes to be created as in [ASTR76].

Each disk record has a bit mask indicating which fields are non-null, and only these fields are actually stored. In addition, because the magnetic disk storage system is fundamentally a versioning system, each record contains an additional 8 fields:

OID a system-assigned unique record identifier
Xmin the transaction identifier of the interaction inserting the record
Tmin the commit time of Xmin (the time at which the record became valid)
Cmin the command identifier of the interaction inserting the record
Xmax the transaction identifier of the interaction deleting the record
Tmax the commit time of Xmax (the time at which the record stopped being valid)
Cmax the command identifier of the interaction deleting the record
PTR a forward pointer

When a record is inserted it is assigned a unique OID, and Xmin and Cmin are set to the identity of the current interaction. the remaining five fields are left blank. When a record is updated, two operations take place. First, Xmax and Cmax are set to the identity of the current interaction in the record being replaced to indicate that it is no longer valid. Second, a new record is inserted into the data base with the proposed replacement values for the data fields. Moreover, OID is set to the OID of the record being replaced, and Xmin and Cmin are set to the identity of the current interaction. When a record is deleted, Xmax and Cmax are set to the identity of the current interaction in the record to be deleted.

When a record is updated, the new version usually differs from the old version in only a few fields. In order to avoid the space cost of a complete new record, the following compression technique has been adopted. The initial record is stored uncompressed and called the **anchor point.** Then, the updated record is differenced against the anchor point and only the actual changes are stored. Moreover, PTR is altered on the anchor point to point to the updated record, which is called a **delta record.** Successive updates generate a one-way linked list of delta records off an initial anchor point. Hopefully most delta record are on the same operating system page as the anchor point since they will typically be small objects.

It is the expectation that POSTGRES would be used as a local data manager in a distributed data base system. Such a distributed system would be expected to maintain multiple copies of all important POSTGRES objects. Recovery from hard crashes, i.e. one for which the disk cannot be read, would occur by switching to some other copy of the object. In a non-distributed system POSTGRES will allow a user to specify that he wishes a second copy of specific objects with the command:

mirror rel-name

Some operating systems (e.g. VMS [DEC86] and Tandem [BART81]) already support mirrored files, so special DBMS code will not be necessary in these environments. Hopefully, mirrored files will become a standard operating systems service in most environments in the future.

2.3. Time Management

The POSTGRES query language, POSTQUEL allows a user to request the salary of Mike using the following syntax.

retrieve (EMP.salary) where EMP.name = "Mike"

To support access to historical tuples, the query language is extended as follows:

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

The scope of this command is the EMP relation as of a specific time, T, and Mike's salary will be found as of that time. A variety of formats for T will be allowed, and a conversion routine will be called to convert times to the 32 bit unsigned integers used internally. POSTGRES constructs a query plan to find qualifying records in the normal fashion. However, each accessed tuple must be additionally checked for validity at the time desired in the user's query. In general, a record is valid at time T if the following is true:

Tmin < T and Xmin is a committed transaction and either:
  Xmax is not a committed transaction or
  Xmax is null or
  Tmax > T

In fact, to allow a user to read uncommitted records that were written by a different command within his transaction, the actual test for validity is the following more complex condition.

Xmin = my-transaction and Cmin != my-command and T = "now"
  or
Tmin < T and Xmin is a committed transaction and either:
  (Xmax is not a committed transaction and Xmax != my-transaction) or
  (Xmax = my-transaction and Cmax = my-command) or
  Xmax is null or
  Tmax > T or

If T is not specified, then T = "now" is the default value, and a record is valid at time, "now" if

Xmin = my-transaction and Cmin != my-command
  or
Xmin is a committed transaction and either
  (Xmax is not a committed transaction and Xmax != my-transaction) or
  (Xmax = my-transaction and Cmax = my-command) or
  Xmax is null

More generally, Mike's salary history over a range of times can be retrieved by:

retrieve (EMP.Tmin, EMP.Tmax, EMP.salary)
using EMP[T1,T2] where EMP.name = "Mike"

This command will find all salaries for Mike along with their starting and ending times as long as the salary is valid at some point in the interval, [T1, T2]. In general, a record is valid in the interval [T1,T2] if:

Xmin = my-transaction and Cmin != my-command and T2 >= "now"
  or
Tmin < T2 and Xmin is a committed transaction and either:
  (Xmax is not a committed transaction and Xmax != my-transaction) or
  (Xmax = my-transaction and Cmax = my-command) or

Xmax is null or
Tmax > T1

Either T1 or T2 can be omitted and the defaults are respectively T1 = 0 and T2 = +infinity

Special programs (such as debuggers) may want to be able to access uncommitted records. To facilitate such access, we define a second specification for each relation, for example:

retrieve (EMP.salary) using all-EMP[T] where EMP.name = "Mike"

An EMP record is in all-EMP at time T if

Tmin < T and (Tmax > T or Tmax = null)

Intuitively, all-EMP[T] is the set of all tuples committed, aborted or in-progress at time T.

Each accessed magnetic disk record must have one of the above tests performed. Although each test is potentially CPU and I/O intensive, we are not overly concerned with CPU resources because we do not expect the CPU to be a significant bottleneck in next generation systems. This point is discussed further in Section 5. Moreover, the CPU portion of these tests can be easily committed to custom logic or microcode or even a co-processor if it becomes a bottleneck.

There will be little or no I/O associated with accessing the status of any transaction, since we expect the transaction log (or its associated bloom filter) to be in main memory. We turn in the next subsection to avoiding I/O when evaluating the remainder of the above predicates.

2.4. Concurrency Control and Timestamp Management

It would be natural to assign a timestamp to a transaction at the time it is started and then fill in the timestamp field of each record as it is updated by the transaction. Unfortunately, this would require POSTGRES to process transactions logically in timestamp order to avoid anomolous behavior. This is equivalent to requiring POSTGRES to use a concurrency control scheme based on timestamp ordering (e.g. [BERN80]. Since simulation results have shown the superiority of conventional locking [AGRA85], POSTGRES uses instead a standard two-phase locking policy which is implemented by a conventional main memory lock table.

Therefore, Tmin and Tmax must be set to the commit time of each transaction (which is the time at which updates logically take place) in order to avoid anomolous behavior. Since the commit time of a transaction is not known in advance, Tmin and Tmax cannot be assigned values at the time that a record is written.

We use the following technique to fill in these fields asynchronously. POSTGRES contains a TIME relation in which the commit time of each transaction is stored. Since timestamps are 32 bit unsigned integers, byte positions 4*j through 4*j + 3 are reserved for the commit time of transaction j. At the time a transaction commits, it reads the current clock time and stores it in the appropriate slot of TIME. The tail of the TIME relation can be stored in stable main memory to avoid the I/O that this update would otherwise entail.

Moreover, each relation in a POSTGRES data base is tagged at the time it is created with one of the following three designations:

no archive: This indicates that no historical access to relations is required.

light archive: This indicates that an archive is desired but little access to it is expected.

heavy archive: This indicates that heavy use will be made of the archive.

For relations with "no archive" status, Tmin and Tmax are never filled in, since access to historical tuples is never required. For such relations, only POSTQUEL commands specified for T = "now" can be processed. The validity check for T = "now" requires access only to the POSTGRES LOG relation which should be contained in the buffer pool. Hence, the test consumes no I/O resources.

If "light archive" is specified, then access to historical tuples is allowed. Whenever Tmin or Tmax must be compared to some specific value, the commit time of the appropriate transaction is retrieved from the TIME relation to make the comparison. Access to historical records will be slowed in the "light archive" situation by this requirement to perform an I/O to the TIME relation for each timestamp value required. This overhead will only be tolerable if archival records are accessed a very small number of times in their lifetime (about 2-3).

In the "heavy archive" condition, the run time system must look up the commit time of a transaction as in the "light archive" case. However, it then writes the value found into Tmin or Tmax, thereby turning the read of a historical record into a write. Any subsequent accesses to the record will then be validatable without the extra access to the TIME relation. Hence, the first access to an archive record will be costly in the "heavy archive" case, but subsequent ones will will incur no extra overhead.

In addition, we expect to explore the utility of running another system demon in background to asynchronously fill in timestamps for "heavy archive" relations.

2.5. Record Access

Records can be accessed by a sequential scan of a relation. In this case, pages of the appropriate file are read in a POSTGRES determined order. Each page contains a pointer to the next and the previous logical page; hence POSTGRES can scan a relation by following the forward linked list. The reverse pointers are required because POSTGRES can execute query plans either forward or backward. Additionally, on each page there is a line table as in [STON76] containing pointers to the starting byte of each anchor point record on that page.

Once an anchor point is located, the delta records linked to it can be constructed by following PTR and decompressing the data fields. Although decompression is a CPU intensive task, we feel that CPU resources will not be a bottleneck in future computers as noted earlier. Also, compression and decompression of records is a task easily committed to microcode or a separate co-processor.

An arbitrary number of secondary indexes can be constructed for any base relation. Each index is maintained by an access method. and provides keyed access on a field or a collection of fields. Each access method must provide all the procedures for the POSTGRES defined abstraction for access methods. These include get-record-by-key, insert-record, delete-record, etc. The POSTGRES run

time system will call the various routines of the appropriate access method when needed during query processing.

Each access method supports efficient access for a collection of operators as noted in [STON86a]. For example, B-trees can provide fast access for any of the operators:

$$\{=, <=, <, >, >=\}$$

Since each access method may be required to work for various data types, the collection of operators that an access methods will use for a specific data type must be registered as an **operator class.** Consequently, the syntax for index creation is:

index on rel-name is index-name ({key-i with operator-class-i})
using access-method-name and performance-parameters

The performance-parameters specify the fill-factor to be used when loading the pages of the index, and the minimum and maximum number of pages to allocate. The following example specifies a B-tree index on a combined key consisting of an integer and a floating point number.

index on EMP is EMP-INDEX (age with integer-ops, salary with float-ops)
using B-tree and fill-factor = .8

The run-time system handles secondary indexes in a somewhat unusual way. When a record is inserted, an anchor point is constructed for the record along with index entries for each secondary index. Each index record contains a key(s) plus a pointer to an entry in the line table on the page where the indexed record resides. This line table entry in turn points to the byte-offset of the actual record. This single level of indirection allows anchor points to be moved on a data page without requiring maintenance of secondary indexes.

When an existing record is updated, a delta record is constructed and chained onto the appropriate anchor record. If no indexed field has been modified, then no maintenance of secondary indexes is required. If an indexed field changed, then an entry is added to the appropriate index containing the new key(s) and a pointer to the anchor record. There are no pointers in secondary indexes directly to delta records. Consequently, a delta record can only be accessed by obtaining its corresponding anchor point and chaining forward.

The POSTGRES query optimizer constructs plans which may specify scanning portions of various secondary indexes. The run time code to support this function is relatively conventional except for the fact that each secondary index entry points to an anchor point and a chain of delta records, all of which must be inspected. Valid records that actually match the key in the index are then returned to higher level software.

Use of this technique guarantees that record updates only generate I/O activity in those secondary indexes whose keys change. Since updates to keyed fields are relatively uncommon, this ensures that few insertions must be performed in the secondary indexes.

Some secondary indexes which are hierarchical in nature require disk pages to be placed in stable storage in a particular order (e.g. from leaf to root for page splits in B+-trees). POSTGRES will provide a low level command

order block-1 block-2

to support such required orderings. This command is in addition to the required pin and unpin commands to the buffer manager.

3. THE ARCHIVAL SYSTEM

3.1. Vacuuming the Disk

An asynchronous demon is responsible for sweeping records which are no longer valid to the archive. This demon, called the **vacuum cleaner**, is given instructions using the following command:

vacuum rel-name after T

Here T is a time relative to "now". For example, the following vacuum command specifies vacuuming records over 30 days old:

vacuum EMP after "30 days"

The vacuum cleaner finds candidate records for archiving which satisfy one of the following conditions:

Xmax is non empty and is a committed transaction and "now" - Tmax >= T
Xmax is non empty and is an aborted transaction
Xmin is non empty and is an aborted transaction

In the second and third cases, the vacuum cleaner simply reclaims the space occupied by such records. In the first case, a record must be copied to the archive unless "no-archive" status is set for this relation. Additionally, if "heavy-archive" is specified, Tmin and Tmax must be filled in by the vacuum cleaner during archiving if they have not already been given values during a previous access. Moreover, if an anchor point and several delta records can be swept together, the vacuuming process will be more efficient. Hence, the vacuum cleaner will generally sweep a chain of several records to the archive at one time.

This sweeping must be done very carefully so that no data is irrecoverably lost. First we discuss the format of the archival medium, then we turn to the sweeping algorithm and a discussion of its cost.

3.2. The Archival Medium

The archival storage system is compatible with WORM devices, but is not restricted to such systems. We are building a conventional extent-based file system on the archive, and each relation is allocated to a single file. Space is allocated in large extents and the next one is allocated when the current one is exhausted. The space allocation map for the archive is kept in a magnetic disk relation. Hence, it is possible, albeit very costly, to sequentially scan the historical version of a relation.

Moreover, there are an arbitrary number of secondary indexes for each relation in the archive. Since historical accessing patterns may be different than accessing patterns for current data, we do not restrict the archive indexes to be the same as those for the magnetic disk data base. Hence, archive indexes must be explicitly created using the following extension of the indexing command:

index on {archive} rel-name is index-name ({key-i with operator-class-i})

using access-method-name and performance-parameters

Indexes for archive relations are normally stored on magnetic disk. However, since they may become very large, we will discuss mechanisms in the next section to support archive indexes that are partly on the archive medium.

The anchor point and a collection of delta records are concatenated and written to the archive as a single variable length record. Again secondary index records must be inserted for any indexes defined for the archive relation. An index record is generated for the anchor point for each archive secondary index. Moreover, an index record must be constructed for each delta record in which a secondary key has been changed.

Since the access paths to the portion of a relation on the archive may be different than the access paths to the portion on magnetic disk, the query optimizer must generate two plans for any query that requests historical data. Of course, these plans can be executed in parallel if multiple processors are available. In addition, we are studying the decomposition of each of these two query plans into additional parallel pieces. A report on this subject is in preparation [BHID87].

3.3. The Vacuum Process

Vacuuming is done in three phases, namely:

phase 1: write an archive record and its associated index records
phase 2: write a new anchor point in the current data base
phase 3: reclaim the space occupied by the old anchor point and its delta records

If a crash occurs while the vacuum cleaner is writing the historical record in phase 1, then the data still exists in the magnetic disk data base and will be revacuumed again at some later time. If the historical record has been written but not the associated indexes, then the archive will have a record which is reachable only through a sequential scan. If a crash occurs after some index records have been written, then it will be possible for the same record to be accessed in a magnetic disk relation and in an archive relation. In either case, the duplicate record will consume system resources; however, there are no other adverse consequences because POSTGRES is a relational system and removes duplicate records during processing.

When the record is safely stored on the archive and indexed appropriately, the second phase of vacuuming can occur. This phase entails computing a new anchor point for the magnetic disk relation and adding new index records for it. This anchor point is found by starting at the old anchor point and calculating the value of the last delta that satisfies

"now" - Tmax >= T

by moving forward through the linked list. The appropriate values are inserted into the magnetic disk relation, and index records are inserted into all appropriate index. When this phase is complete, the new anchor point record is accessible directly from secondary indexes as well as by chaining forward from the old anchor point. Again, if there is a crash during this phase a record may be accessible twice in some future queries, resulting in additional overhead but no other consequences.

The last phase of the vacuum process is to remove the original anchor point followed by all delta records and then to delete all index records that pointed to this deleted anchor point. If there is a crash during this phase, index records may exist that do not point to a correct data record. Since the run-time system must already check that data records are valid and have the key that the appropriate index record expects them to have, this situation can be checked using the same mechanism.

Whenever there is a failure, the vacuum cleaner is simply restarted after the failure is repaired. It will re-vacuum any record that was in progress at some later time. If the crash occurred during phase 3, the vacuum cleaner could be smart enough to realize that the record was already safely vacuumed. However, the cost of this checking is probably not worthwhile. Consequently, failures will result in a slow accumulation of extra records in the archive. We are depending on crashes to be infrequent enough that this is not a serious concern.

We now turn to the cost of the vacuum cleaner.

3.4. Vacuuming Cost

We examine two different vacuuming situations. In the first case we assume that a record is inserted, updated K times and then deleted. The whole chain of records from insertion to deletion is vacuumed at once. In the second case, we assume that the vacuum is run after K updates, and a new anchor record must be inserted. In both cases, we assume that there are Z secondary indexes for both the archive and magnetic disk relation, that no key changes are made during these K updates, and that an anchor point and all its delta records reside on the same page. Table 1 indicates the vacuum cost for each case. Notice that vacuuming consumes a constant cost. This rather surprising conclusion reflects the fact that a new anchor record can be inserted on the same page from which the old anchor point is being deleted without requiring the page to be forced to stable memory in between the operations. Moreover, the new index records can be inserted on the same page from which the previous entries are deleted without an intervening I/O. Hence, the cost PER RECORD of the vacuum cleaner decreases as the length of the

	whole chain	K updates
archive-writes	1+Z	1+Z
disk-reads	1	1
disk-writes	1+Z	1+Z

I/O Counts for Vacuuming
Table 1

chain, K, increases. As long as an anchor point and several delta records are vacuumed together, the cost should be marginal.

4. INDEXING THE ARCHIVE

4.1. Magnetic Disk Indexes

The archive can be indexed by conventional magnetic disk indexes. For example, one could construct a salary index on the archive which would be helpful in answering queries of the form:

retrieve (EMP.name) using EMP [,] where EMP.salary = 10000

However, to provide fast access for queries which restrict the historical scope of interest, e.g:

retrieve (EMP.name) using EMP [1/1/87,] where EMP.salary = 10000

a standard salary index will not be of much use because the index will return all historical salaries of the correct size whereas the query only requested a small subset. Consequently, in addition to conventional indexes, we expect time-oriented indexes to be especially useful for archive relations. Hence, the two fields, Tmin and Tmax, are stored in the archive as a single field, I, of type interval. An R-tree access method [GUTM84] can be constructed to provide an index on this interval field. The operators for which an R-tree can provide fast access include "overlaps" and "contained-in". Hence, if these operators are written for the interval data type, an R-tree can be constructed for the EMP relation as follows:

index on archive EMP is EMP-INDEX (I with interval-ops)
using R-tree and fill-factor = .8

This index can support fast access to the historical state of the EMP relation at any point in time or during a particular period.

To utilize such indexes, the POSTGRES query planner needs to be slightly modified. Note that POSTGRES need only run a query on an archive relation if the scope of the relation includes some historical records, Hence, the query for an archive relation must be of the form:

...using EMP[T]

or

...using EMP[T1,T2]

The planner converts the first construct into:

...where T contained-in EMP.I

and the second into:

...where interval(T1,T2) overlaps EMP.I

Since all records in the archive are guaranteed to be valid, these two qualifications can replace all the low level code that checks for record validity on the magnetic disk described in Section 2.3. With this modification, the query optimizer can use the added qualification to provide a fast access path through an interval index if one exists.

Moreover, we expect combined indexes on the interval field along with some data value to be very attractive, e.g:

index on archive EMP is EMP-INDEX
(I with interval-ops, salary with float-ops)
using R-tree and fill-factor = .8

Since an R-tree is a multidimensional index, the above index supports intervals which exist in a two dimensional space of time and salaries. A query such as:

retrieve (EMP.name) using EMP[T1,T2] where EMP.salary = 10000

will be turned into:

retrieve (EMP.name) where EMP.salary = 10000
and interval(T1,T2) overlaps EMP.I

The two clauses of the qualification define another interval in two dimensions and conventional R-tree processing of the interval can be performed to use both qualifications to advantage.

Although data records will be added to the archive at the convenience of the vacuum cleaner, records will be generally inserted in ascending time order. Hence, the poor performance reported in [ROUS85] for R-trees should be averted by the nearly sorted order in which the records will be inserted. Performance tests to ascertain this speculation are planned. We now turn to a discussion of R-tree indexes that are partly on both magnetic and archival mediums.

4.2. Combined Media Indexes

We begin with a small space calculation to illustrate the need for indexes that use both media. Suppose a relation exists with 10**6 tuples and each tuple is modified 30 times during the lifetime of the application. Suppose there are two secondary indexes for both the archive and the disk relation and updates never change the values of key fields. Moreover, suppose vacuuming occurs after the 5th delta record is written, so there are an average of 3 delta records for each anchor point. Assume that anchor points consume 200 bytes, delta records consume 40 bytes, and index keys are 10 bytes long.

With these assumptions, the sizes in bytes of each kind of object are indicated in Table 2. Clearly, 10**6 records will consume 200 mbytes while 3 x 10**6 delta records will require 120 mbytes. Each index record is assumed to require a four byte pointer in addition to the 10 byte key; hence each of the two indexes will take up 14 mbytes. There are 6 anchor point records on the archive for each of the 10**6 records each concatenated with 4 delta records. Hence, archive records will be 360 bytes long, and require 2160 mbytes. Lastly, there is an index record for each of the archive anchor points; hence the archive indexes are 6 times as large as the magnetic disk indexes.

Two points are evident from Table 2. First, the archive can become rather large. Hence, one should vacuum infrequently to cut down on the number of anchor points that occur in the archive. Moreover, it might be desirable to differentially code the anchor points to save space. The second point to notice is that the archive indexes consume a large amount of space on magnetic disk. if the

object	mbytes
disk relation anchor points	200
deltas	120
secondary indexes	28
archive	2160
archive indexes	168

Sizes of the Various Objects
Table 2

target relation had three indexes instead of two, the archive indexes would consume a greater amount of space than the magnetic disk relation. Hence, we explore in this section data structures that allow part of the index to migrate to the archive. Although we could alternatively consider index structures that are entirely on the archive, such as those proposed in [VITT85], we believe that combined media structures will substantially outperform structures restricted to the archive. We plan performance comparisons to demonstrate the validity of this hypothesis.

Consider an R-tree storage structure in which each pointer in a non-leaf node of the R-tree is distinguished to be either a magnetic disk page pointer or an archive page pointer. If pointers are 32 bits, then we can use the high-order bit for this purpose thereby allowing the remaining 31 bits to specify 2**31 pages on magnetic disk or archive storage. If pages are 8K bytes, then the maximum size of an archive index is 2**44 bytes (about 1.75 x 10**13 bytes), clearly adequate for almost any application. Moreover, the leaf level pages of the R-tree contain key values and pointers to associated data records. These data pointers can be 48 bytes long, thereby allowing the data file corresponding to a single historical relation to be 2**48 bytes long (about 3.0 x 10**14 bytes), again adequate for most applications.

We assume that the archive may be a write-once-read-many (WORM) device that allows pages to be initially written but then does not allow any overwrites of the page. With this assumption, records can only be dynamically added to pages

that reside on magnetic disk. Table 3 suggests two sensible strategies for the placement of new records when they are not entirely contained inside some R-tree index region corresponding to a magnetic disk page.

Moreover, we assume that any page that resides on the archive contains pointers that in turn point only to pages on the archive. This avoids having to contend with updating an archive page which contains a pointer to a magnetic disk page that splits.

Pages in an R-tree can be moved from magnetic disk to the archive as long as they contain only archive page pointers. Once a page moves to the archive, it becomes read only. A page can be moved from the archive to the magnetic disk if its parent page resides on magnetic disk. In this case, the archive page previously inhabited by this page becomes unusable. The utility of this reverse migration seems limited, so we will not consider it further.

We turn now to several page movement policies for migrating pages from magnetic disk to the archive and use the parameters indicated in Table 4 in the discussion to follow. The simplist policy would be to construct a system demon to "vacuum" the index by moving the leaf page to the archive that has the smallest value for Tmax, the left-hand end of its interval. This vacuuming would occur whenever the R-tree structure reached a threshold near its maximum size of F disk pages. A second policy would be to choose a worthy page to archive based both on its value of Tmax and on percentage fullness of the page. In either case, insertions would be made into the R-tree index at the lower left-hand part of the index while

---

P1	allocate to the region which has to be expanded the least
P2	allocate to the region whose maximum time has to be expanded the least

Record Insertion Strategies
Table 3

F	number of magnetic disk blocks usable for the index
U	update frequency of the relation being indexed
L	record size in the index being constructed
B	block size of magnetic disk pages

Parameters Controlling Page Movement
Table 4

---

the demon would be archiving pages in the lower right hand part of the index. Whenever an intermediate R-tree node had descendents all on the archive, it could in turn be archived by the demon.

For example, if B is 8192 bytes, L is 50 bytes and there is a five year archive of updates at a frequency, U of 1 update per second, then 1.4 x 10**6 index blocks will be required resulting in a four level R-tree. F of these blocks will reside on magnetic disk and the remainder will be on the archive. Any insertion or search will require at least 4 accesses to one or the other storage medium.

A third movement policy with somewhat different performance characteristics would be to perform "batch movement". In this case one would build a magnetic disk R-tree until its size was F blocks. Then, one would copy the all pages of the R-tree except the root to the archive and allocate a special "top node" on magnetic disk for this root node. Then, one would proceed to fill up a second complete R-tree of F-1 pages. While the second R-tree was being built, both this new R-tree and the one on the archive would be searched during any retrieval request. All inserts would, of course, be directed to the magnetic disk R-tree. When this second R-tree was full, it would be copied to the archive as before and its root node added to the existing top node. The combination might cause the top node to overflow, and a conventional R-tree split would be accomplished. Consequently, the top node would become a conventional R-tree of three nodes. The filling process would start again on a 3rd R-tree of F-3 nodes. When this was full, it would be archived and its root added to the lower left hand page of the 3 node R-tree.

Over time, there would continue to be two R-trees. The first would be completely on magnetic disk and periodically archived. As long as the height of this R-tree at the time it is archived is a constant, H, then the second R-tree of height, H1, will have the bottom H-1 levels on the archive. Moreover, insertions into the magnetic disk portion of this R-tree are always on the left-most page. Hence, the pages along the left-side of the tree are the only ones which will be modified; other pages can be archived if they point entirely to pages on the archive. Hence, some subcollection of the pages on the top H1-H+1 levels remain on the magnetic disk. Insertions go always to the first R-tree while searches go to both R-trees. Of course, there are no deletions to be concerned with.

Again if B is 8192 bytes, L is 50 bytes and F is 6000 blocks, then H will be 3 and each insert will require 3 magnetic disk accesses. Moreover, at 1 update per second, a five year archive will require a four level R-tree whose bottom two levels will be on the archive and a subcollection of the top 2 levels of 100-161 blocks will be on magnetic disk. Hence, searches will require descending two R-trees with a total depth of 7 levels and will be about 40 percent slower than either of the single R-tree structures proposed. On the other hand, the very common operation of insertions will be approximately 25 percent faster.

5. PERFORMANCE COMPARISON

5.1. Assumptions

In order to compare our storage system with a conventional one based on write-ahead logging (WAL), we make the following assumptions.

1) Portions of the buffer pool may reside in non-volatile main memory

2) CPU instructions are not a critical resource, and thereby only I/O operations are counted.

The second assumption requires some explanation. Current CPU technology is driving down the cost of a MIP at a rate of a factor of two every couple of years. Hence, current low-end workstations have a few MIPs of processing power. On the other hand, disk technology is getting denser and cheaper. However, disks are not getting faster at a significant rate. Hence, one can still only expect to read about 30 blocks per second off of a standard disk drive. Current implementations of data base systems require several thousand instructions to fetch a page from the disk followed by 1000-3000 instructions per data record examined on that page. As a simple figure of merit, assume 30000 instructions are required to process a disk block. Hence, a 1 MIP CPU will approximately balance a single disk. Currently, workstations with 3-5 MIPs are available but are unlikely to be configured with 3-5 disks. Moreover, future workstations (such as SPUR and FIREFLY) will have 10-30 MIPs. Clearly, they will not have 10-30 disks unless disk systems shift to large numbers of SCSI oriented single platter disks and away from current SMD disks.

Put differently, a SUN 3/280 costs about $5000 per MIP, while an SMD disk and controller costs about $12,000. Hence, the CPU cost to support a disk is much smaller than the cost of the disk, and the major cost of data base hardware can be expected to be in the disk system. As such, if an installation is found to be CPU bound, then additional CPU resources can be cheaply added until the system becomes balanced.

We analyze three possible situations:

large-SM:	an ample amount of stable main memory is available
small-SM:	a modest amount of stable main memory is available
no-SM:	no stable main memory is availble

In the first case we assume that enough stable main memory is available for POSTGRES and a WAL system to use so that neither system is required to force disk pages to secondary storage at the time that they are updated. Hence, each system will execute a certain number of I/O operations that can be buffered in stable memory and written out to disk at some convenient time. We count the number of such non-forced I/O operations that each system will execute, assuming all writes cost the same amount. For both systems we assume that records do not cross page boundaries, so each update results in a single page write. Moreover, we assume that each POSTGRES delta record can be put on the same page as its anchor point. Next, we assume that transactions are a single record insertion, update, deletion or an aborted update. Moreover, we assume there are two secondary indexes on the relation affected and that updates fail to alter either key field. Lastly, we assume that a write ahead log will require 3 log records (begin transaction, the data modification, and end transaction), with a total length of 400 bytes. Moreover, secondary index operations are not logged and thereby the log records for 10 transactions will fit on a conventional 4K log page.

In the second situation we assume that a modest amount of stable main memory is available. We assume that the quantity is sufficient to hold only the tail of the POSTGRES log and the tail of the TIME relation. In a WAL system, we assume that stable memory can buffer a conventional log turning each log write into one that need not be synchronously forced out to disk. This situation (small-SM) should be contrasted with the third case where no stable memory at all is available (no-SM). In this latter cases, some writes must be forced to disk by both types of storage systems.

In the results to follow we ignore the cost that either kind of system would incur to mirror the data for high availability. Moreover, we are also ignoring the WAL cost associated with checkpoints. In addition, we assume that a WAL system never requires a disk read to access the appropriate un-do log record. We are also ignoring the cost of vacuuming the disk in the POSTGRES architecture.

5.2. Performance Results

Table 5 indicates the number of I/O operations each of the four types of transactions must execute for the assumed large-SM configuration. Since there is ample stable main memory, neither system must force any data pages to disk and only non-forced I/Os must be done. An insert requires that a data record and two index records be written by either system. Moreover, 1/10th of a log page will be filled by the conventional system, so every 10 transactions there will be another log page which must be eventually written to disk. In POSTGRES the insertions to the LOG relation and the TIME relation generate an I/O every 65536 and 2048 transactions respectively, and we have ignored this small number in Table 5. Consequently, one requires 3 non-forced I/Os in POSTGRES and 3.1 in a conventional system. The next two columns in Table 1 can be similarly computed. The last column summarizes the I/Os for an aborted transaction. In POSTGRES the updated page need not be rewritten to disk. Hence, no I/Os are strictly necessary; however, in all liklihood, this optimization will not be implemented. A WAL system will update the data and construct a log record. Then the log record must be read and the data page returned to its original value. Again, a very clever system could avoid writing the page out to disk, since it is identical to the disk copy. Hence, for both systems we indicate both the optimized number of writes and the non-optimized number. Notice in Table 5 that POSTGRES is marginally better than a WAL system except for deletes where it is dramatically better because it does not delete the 2 index records. We now turn to cases where POSTGRES is less attractive.

Table 6 repeats the I/O counts for the small-SM configuration. The WAL configuration performs exactly as in Table 5 while the the POSTGRES data pages must now be forced to disk since insufficient stable main memory is assumed to hold them. Notice that POSTGRES is still better in the total number of I/O operations; however the requirement to do them synchronously will be a major disadvantage.

Table 7 then indicates the I/O counts under the condition that NO stable main memory is available. Here the log record for a conventional WAL system must be forced to disk at commit time. The other writes can remain in the buffer pool and

	Insert	Update	Delete	Abort
WAL-force	0	0	0	0
WAL-no-force	3.1	1.1	3.1	0.1 or 1.1
POSTGRES-force	0	0	0	0
POSTGRES-non-force	3	1	1	0 or 1

I/O Counts for the Primitive Operations
large-SM Configuration
Table 5

	Insert	Update	Delete	Abort
WAL-force	0	0	0	0
WAL-no-force	3.1	1.1	3.1	0.1 or 1.1
POSTGRES-force	3	1	1	0 or 1
POSTGRES-non-force	0	0	0	0

I/O Counts for the Primitive Operations
small-SM Configuration
Table 6

be written at a later time. In POSTGRES the LOG bit must be forced out to disk along with the insert to the TIME relation. Moreover, the data pages must be forced as in Table 6. In this case POSTGRES is marginally poorer in the total number of operations; and again the synchronous nature of these updates will be a significant disadvantage.

In summary, the POSTGRES solution is preferred in the large-SM configuration since all operations require less I/Os. In Table 6 the total number of I/Os is less for POSTGRES; however, synchronous I/O is required. Table 7 shows a situation where POSTGRES is typically more expensive. However, group commits [DEWI84] could be used to effectively convert the results for either type of system into the ones in Table 6. Consequently, POSTGRES should be thought of as fairly competitive with current storage architectures. Moreover, it has a considerable

	Insert	Update	Delete	Abort
WAL-force	1	1	1	1
WAL-no-force	3	1	3	0 or 1
POSTGRES-force	5	3	3	1
POSTGRES-non-force	0	0	0	0 or 1

I/O Counts for the Primitive Operations
no-SM Configuration
Table 7

advantage over WAL systems in that recovery time will be instantaneous while requiring a substantial amount of time in a WAL architecture.

6. CONCLUSIONS

This paper has described the storage manager that is being constructed for POSTGRES. The main points guiding the design of the system were:

1) instantaneous recovery from crashes

2) ability to keep archival records on an archival medium

3) housekeeping chores should be done asynchronously

4) concurrency control based on conventional locking

The first point should be contrasted with the standard write-ahead log (WAL) storage managers in widespread use today.

In engineering application one often requires the past history of the data base. Moreover, even in business applications this feature is sometimes needed, and the now famous TP1 benchmark assumes that the application will maintain an archive. It makes more sense for the data manager to do this task internally for applications that require the service.

The third design point has been motivated by the desire to run multiple concurrent processes if there happen to be extra processors. Hence storage management functions can occur in parallel on multiple processors. Alternatively, some functions can be saved for idle time on a single processor. Lastly, it allows POSTGRES code to be a collection of asynchronous processes and not a single large monolithic body of code.

The final design point reflects our intuitive belief, confirmed by simulations, that standard locking is the most desirable concurrency control strategy.

Moreover, it should be noted that read-only transactions can be optionally coded to run as of some point in the recent past. Since historical commands set no locks, then read-only transactions will never interfere with transactions performing updates or be required to wait. Consequently, the level of contention in a POSTGRES data base may be a great deal lower than that found in conventional storage managers.

The design of the POSTGRES storage manager has been sketched and a brief analysis of its expected performance relative to a conventional one has been performed. If the analysis is confirmed in practice, then POSTGRES will give similar performance compared to other storage managers while providing the extra service of historical access to the data base. This should prove attractive in some environments.

At the moment, the magnetic disk storage manager is operational, and work is proceeding on the vacuum cleaner and the layout of the archive. POSTGRES is designed to support extendible access methods, and we have implemented the B-tree code and will provide R-trees in the near future. Additional access methods can be constructed by other parties to suit their special needs. When the remaining pieces of the storage manager are complete, we plan a performance "bakeoff" both against conventional storage managers as well as against other storage managers (such as [CARE86, COPE84]) with interesting properties.

REFERENCES

[AGRA85] Agrawal, R. et. al., "Models for Studying Concurrency Control Performance Alternatives and Implications," Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.

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

[BART81] Bartlett, J., "A Non-STOP Kernel," Proc. Eighth Symposium on Operating System Principles," Pacific Grove, Ca., Dec. 1981.

[BERN80] Bernstein, P. at. al., "Concurrency Control in a System for Distributed Databases (SDD-1)," ACM-TODS, March 1980.

[BHID87] Bhide, A., "Query Processing in Shared Memory Multiprocessor Systems," (in preparation).

[CARE86] Carey, M. et. al., "Object and File Management in the EXODUS Database System," Proc. 1986 VLDB Conference, Kyoto, Japan, August 1986.

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

[DEC86] Digital Equipment Corp., "VAX/VMS V4.0 Reference Manual," Digital Equipment Corp., Maynard, Mass., June 1986.

[DEWI84] Dewitt, D. et. al., "Implementation Techniques for Main Memory Database Systems," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," IBM Research, San Jose, Ca., RJ1879, June 1978.

[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.

[HILL86] Hill, M., et al. "Design Decisions in SPUR," Computer Magazine, vol.19, no.11, November 1986.

[ROUS85] Roussoupoulis, N. and Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.

[SEQU85] Sequent Computer Co., "The SEQUENT Balance Reference Manual," Sequent Computers, Portland, Ore., 1985.

[SEVR76] Severence, D., and Lohman, G., "Differential Files: Their Application to the Maintenance of large Databases," ACM-TODS, June 1976.

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

[STON86] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86a] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[VITT85] Vitter, J., "An Efficient I/O Interface for Optical Disks," ACM-TODS, June 1985.

A Shared Object Hierarchy†

Lawrence A. Rowe

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Abstract

This paper describes the design and proposed implementation of a shared object hierarchy. The object hierarchy is stored in a relational database and objects referenced by an application program are cached in the program's address space. The paper describes the database representation for the object hierarchy and the use of POSTGRES, a next-generation relational database management system, to implement object referencing efficiently. The shared object hierarchy system will be used to implement OBJFADS, an object-oriented programming environment for interactive multimedia database applications, that will be the programming interface to POSTGRES.

1. Introduction

Object-oriented programming has received much attention recently as a new way to develop and structure programs [12,30]. This new programming paradigm, when coupled with a sophisticated interactive programming environment executing on a workstation with a bit-mapped display and mouse, improves programmer productivity and the quality of programs they produce.

A program written in an object-oriented language is composed of a collection of objects that contain data and procedures. These objects are organized into an *object hierarchy*. Previous implementations of object-oriented languages have required each user to have his or her own private object

† This research was supported by the National Science Foundation under Grant DCR-8507256 and the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

hierarchy. In other words, the object hierarchy is not shared. Moreover, the object hierarchy is usually restricted to main memory. The LOOM system stored object hierarchies in secondary memory [14], but it did not allow object sharing. These restrictions limit the applications to which this new programming technology can be applied.

There are two approaches to building a shared object hierarchy capable of storing a large number of objects. The first approach is to build an object data manager [2,9-11,17,20,35]. In this approach, the data manager stores objects that a program can fetch and store. The disadvantage of this approach is that a complete database management system (DBMS) must be written. A query optimizer is needed to support object queries (e.g., "fetch all *foo* objects where field *bar* is *bas*"). Moreover, the optimizer must support the equivalent of relational joins because objects can include references to other objects. A transaction management system is needed to support shared access and to maintain data integrity should the software or hardware crash. Finally, protection and integrity systems are required to control access to objects and to maintain data consistency. These modules taken together account for a large fraction of the code in a DBMS. Proponents of this approach argue that some of this functionality can be avoided. However, we believe that eventually all of this functionality will be required for the same reasons that it is required in a conventional database management system.

The second approach, and the one we are taking, is to store the object hierarchy in a relational database. The advantage of this approach is that we do not have to write a DBMS. A beneficial side-effect is that programs written in a conventional programming language can simultaneously access the data stored in the object hierarchy. The main objection to this approach has been that the performance of existing relational DBMS's has been inadequate. We believe this problem will be solved by using POSTGRES as the DBMS on which to implement the shared hierarchy. POSTGRES is a next-generation DBMS currently being implemented at the University of California, Berkeley [31]. It has a number of features, including data of type procedure, alerters, precomputed procedures and rules, that can be used to implement the shared object hierarchy efficiently.

Figure 1 shows the architecture of the proposed system. Each application process is connected to a database process that manages the shared database. The application program is presented a conventional view of the object hierarchy. As objects are referenced by the program, a run-time system retrieves them from the database. Objects retrieved from the database

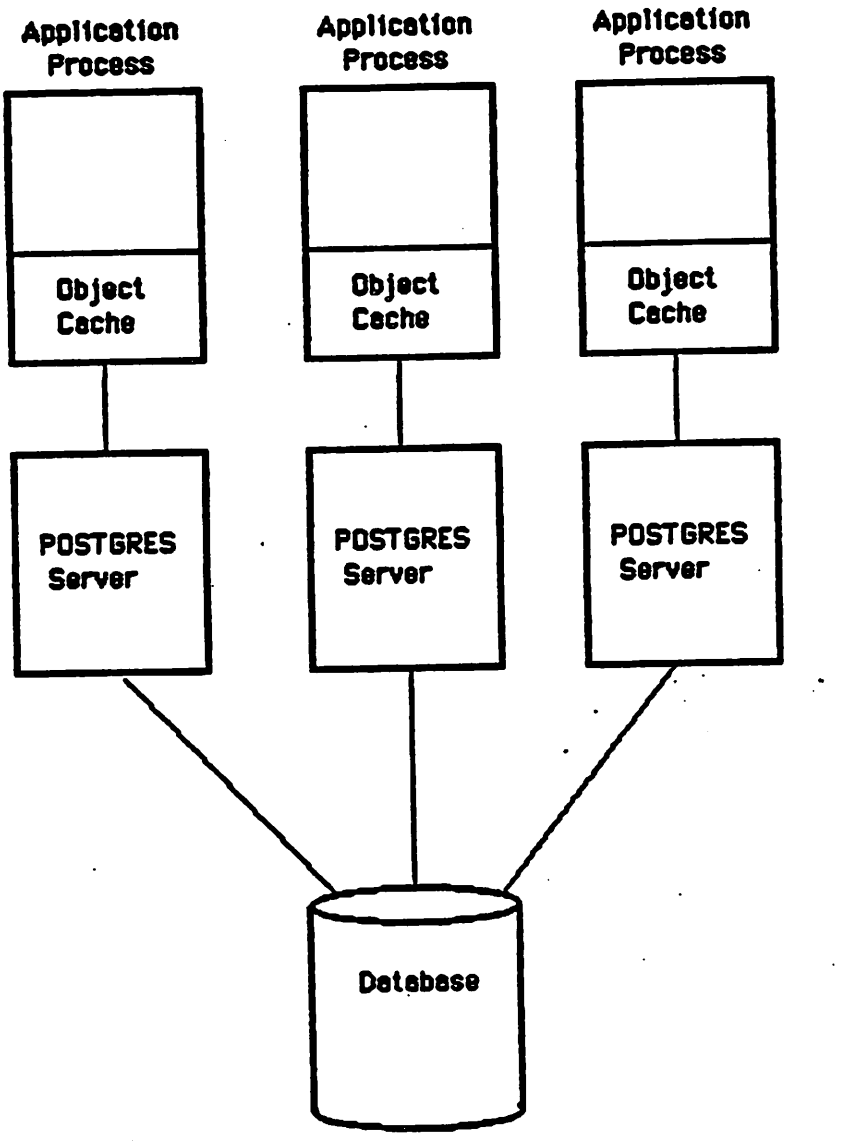

Figure 1. Process architecture.

are stored in an object cache in the application process so that subsequent references to the object will not require another database retrieval. Object updates by the application are propagated to the database and to other processes that have cached the object.

Other research groups are also investigating this approach [1,5,16,21,22,28]. The main difference between our work and the work of these other groups is the object cache in the application process. They have not addressed the problem of maintaining cache consistency when more than one application process is using an object. Research groups that are addressing the object cache problem are using different implementation strategies that will have different performance characteristics [17,18,20].

This paper describes how the OBJFADS shared object hierarchy will be implemented using POSTGRES. The remainder of this paper is organized as follows. Section 2 presents the object model. Section 3 describes the database representation for the shared object hierarchy. Section 4 describes the design of the object cache including strategies for improving the performance of fetching objects from the database. Section 5 discusses object updating and transactions. Section 6 describes the support for selecting and executing methods. And lastly, section 7 summarizes the paper.

2. Object Hierarchy Model

This section describes the object hierarchy model. The model is based on the Common Lisp Object System (CLOS) [7] because OBJFADS is being implemented in Common Lisp [29].

An *object* can be thought of as a record with named *slots*. Each slot has a data type and a default value. The data type can be a primitive type (e.g., *Integer*) or a reference to another object.[1] The type of an object is called the *class* of the object. Class information (e.g., slot definitions) is represented by another object called the *class object*.[2] A particular object is also called an *instance* and object slots are also called *instance variables*.

A class inherits data definitions (i.e., slots) from another class, called a *superclass*, unless a slot with the same name is defined in the class. Figure 2 shows a class hierarchy (i.e., type hierarchy) that defines equipment in an integrated circuit (IC) computer integrated manufacturing database. [26].

---

[1] An object reference is represented by an *object identifier* (*objid*) that uniquely identifies the object.

[2] The term *class* is used ambiguously in the literature to refer to the type of an object, the object that represents the type (i.e., the class object), and the set of objects of a specific type. We will indicate the desired meaning in the surrounding text.

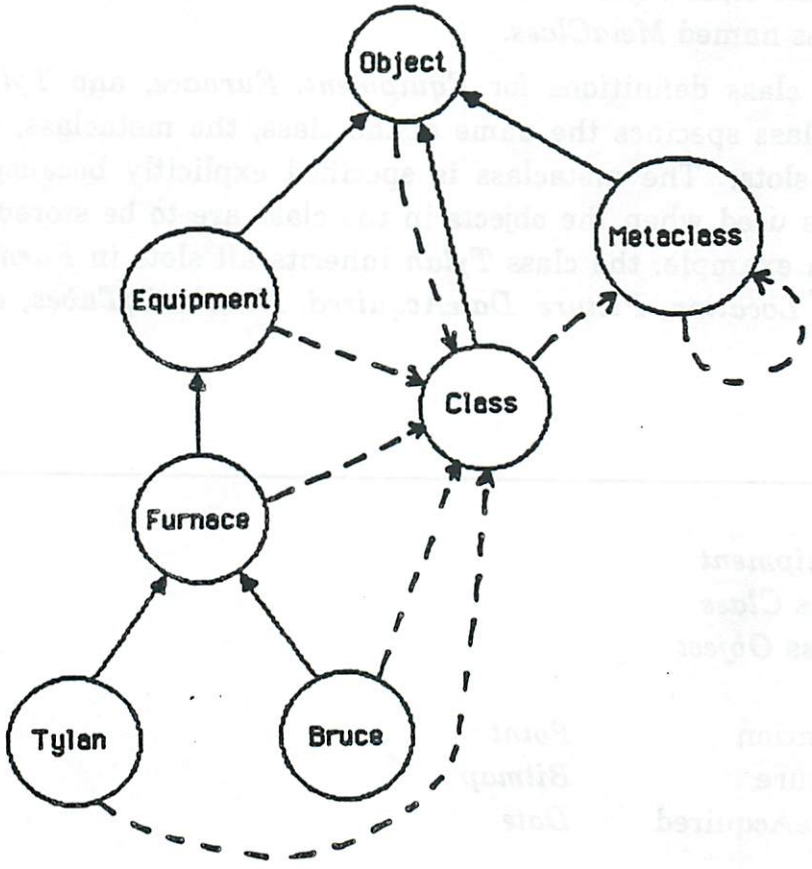

Figure 2: Equipment class hierarchy.

Each class is represented by a labelled node (e.g., *Object*, *Equipment*, *Furnace*, etc.). The superclass of each class is indicated by the solid line with an arrowhead. By convention, the top of the hierarchy is an object named *Object*. In this example, the class *Tylan*, which represents a furnace produced by a particular vendor, inherits slots from *Object*, *Equipment*, and *Furnace*.

As mentioned above, the class is represented by an object. The type of these class objects is represented by the class named *Class*. In other words, they are instances of the class *Class*. The *InstanceOf* relationship is represented by dashed lines in the figure. For example, the class object *Equipment* is an instance of the class *Class*. Given an object, it is possible to determine the class of which it is an instance. Consequently, slot

definitions and, as described below, procedures that operate on the object can be looked-up in the class object. For completeness, the type of the class named *Class* is a class named *MetaClass*.

Figure 3 shows class definitions for *Equipment*, *Furnace*, and *Tylan*. The definition of a class specifies the name of the class, the metaclass, the superclass, and the slots. The metaclass is specified explicitly because a different metaclass is used when the objects in the class are to be stored in the database. In the example, the class *Tylan* inherits all slots in *Furnace* and *Equipment* (i.e., *Location*, *Picture*, *DateAcquired*, *NumberOfTubes*, and *MaxTemperature*).

Class *Equipment*
MetaClass *Class*
Superclass *Object*
Slots

Location	*Point*
Picture	*Bitmap*
DateAcquired	*Date*

Class *Furnace*
MetaClass *Class*
Superclass *Equipment*
Slots

NumberOfTubes	*Integer*
MaxTemperature	*DegreesCelsius*

Class *Tylan*
MetaClass *Class*
Superclass *Furnace*
Slots

Figure 3: Class definitions for equipment.

Variables can be defined that are global to all instances of a class. These variables, called *class variables*, hold data that represents information about the entire class. For example, a class variable *NumberOfFurnaces* can be defined for the class *Furnace* to keep track of the number of furnaces. Class variables are inherited just like instance variables except that inherited class variables refer to the same memory location. For example, the slot named *NumberOfFurnaces* inherited by *Tylan* and *Bruce* refer to the same variable as the class variable in *Furnace*.

Procedures that manipulate objects, called *methods*, take arguments of a specific class (i.e., type). Methods with the same name can be defined for different classes. For example, two methods named *area* can be defined: one that computes the area of a *box* object and one that computes the area of a *circle* object. The method executed when a program makes a call on *area* is determined by the class of the argument object. For example,

```
area(x)
```

calls the *area* method for *box* if *x* is a *box* object or the *area* method for *circle* if it is a *circle* object. The selection of the method to execute is called *method determination.*

Methods are also inherited from the superclass of a class unless the method name is redefined. Given a function call "*f(x)*", the method invoked is determined by the following algorithm. Follow the *InstanceOf* relationship from *x* to determine the class of the argument. Invoke the method named *f* defined for the class, if it exists. Otherwise, look for the method in the superclass of the class object. This search up the superclass hierarchy continues until the method is found or the top of the hierarchy is reached in which case an error is reported.

Figure 4 shows some method definitions for *Furnace* and *Tylan*. Furnaces in an IC fabrication facility are potentially dangerous, so they are locked when they are not in use. The methods *Lock* and *UnLock* disable and enable the equipment. These methods are defined for the class *Furnace* so that all furnaces will have this behavior. The argument to these methods is an object representing a furnace.[3] The methods *CompileRecipe* and *LoadRecipe* compile and load into the furnace code that, when executed by

---

[3] The argument name *self* was chosen because it indicates which argument is the object.

```
method Lock(self: Furnace)
    ...
method UnLock(self: Furnace)
    ...
method CompileRecipe(self: Tylan, recipe: Text)
    ...
method LoadRecipe(self: Tylan, recipe: Code)
    ...
```

Figure 4: Example method definitions.

the furnace, will process the semiconductor wafers as specified by the recipe text. These methods are defined on the *Tylan* class because they are different for each vendor's furnace. With these definitions, the class *Tylan* has four methods because it inherits the methods from *Furnace*.

Slot and method definitions can be inherited from more than one superclass. For example, the *Tylan* class can inherit slots and methods that indicate how to communicate with the equipment through a network connection by including the *NetworkMixin* class in the list of superclasses.[4] Figure 5 shows the definition of *NetworkMixin* and the modified definition of *Tylan*. With this definition, *Tylan* inherits the slots and methods from *NetworkMixin* and *Furnace*. A name conflict arises if two superclasses define slots or methods with the same name (e.g., *Furnace* and *NetworkMixin* might both have a slot named *Status*). A name conflict is resolved by inheriting the definition from the first class that has a definition for the name in the superclass list. Inheriting definitions from multiple classes is called *multiple inheritance*.

[4] The use of the suffix *Mixin* indicates that this object defines behavior that is added to or mixed into other objects. This suffix is used by convention to make it easier to read and understand an object hierarchy.

Class *NetworkMixin*
MetaClass *Class*
Superclass *Object*
Instance Variables
HostName *Text*
Device *Text*
Methods
SendMessage(self: *NetworkMixin*; msg: *Message*)
ReceiveMessage (self: *NetworkMixin*) **returns** *Message*

Class *Tylan*
MetaClass *Class*
Superclass *Furnace NetworkMixin*
...

Figure 5: Multiple inheritance example.

3. Shared Object Hierarchy Database Design

The view of the object hierarchy presented to an application program is one consistent hierarchy. However, a portion of the hierarchy is actually shared among all concurrent users of the database. This section describes how the shared portion of the hierarchy will be stored in the database.

Shared objects are created by defining a class with metaclass *DBClass*. All instances of these classes, called *shared classes*, are stored in the database. A predefined shared class, named *DBObject*, is created at the top of the shared object hierarchy. The relationship between this class and the other predefined classes is shown in figure 6. All superclasses of a shared object class must be shared classes except *DBObject*. This restriction is required so that all definitions inherited by a shared class will be stored in the database.

The POSTGRES data model supports attribute inheritance, user-defined data types, data of type procedure, and rules [25,31] which are used by OBJFADS to create the database representation for shared objects. System catalogs are defined that maintain information about shared classes. In addition, a relation is defined for each class that contains a tuple that

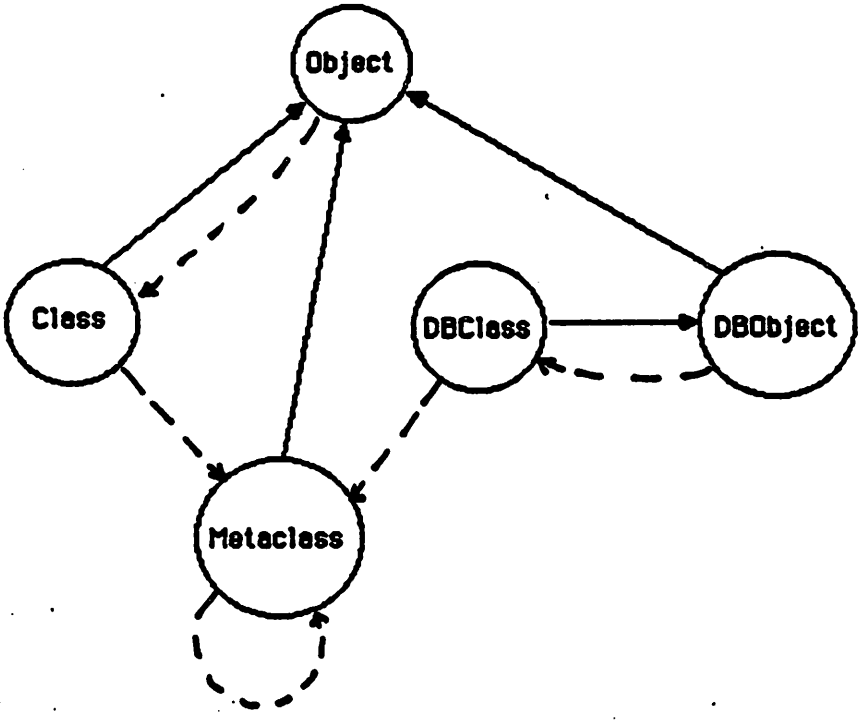

Figure 6: Predefined classes.

represents each class instance. This relation is called the *instance relation.*

OBJFADS maintains four system catalogs to represent shared class information: *DBObject*, *DBClass*, *SUPERCLASS*, and *METHODS*. The *DBObject* relation identifies objects in the database:

CREATE DBObject(Instance, Class)

where

Instance is the *objid* of the object.

Class is the *objid* of the class object of this instance.

This catalog defines attributes that are inherited by all instance relations. No tuples are inserted into this relation (i.e., it represents an abstract class). However, all shared objects can be accessed through it by using transitive closure queries. For example, the following query retrieves the *objid* of all instances:

RETRIEVE (DBObject*.Instance)

The asterisk indicates closure over the relation *DBObject* and all other relations that inherit attributes from it.

POSTGRES maintains a unique identifier for every tuple in the database. Each relation has a predefined attribute that contains the unique identifier. While these identifiers are unique across all relations, the relation that contains the tuple cannot be determined from the identifier. Consequently, we created our own object identifier (i.e., an *objid*) that specifies the relation and tuple. A POSTGRES user-defined data type, named *objid*, that represents this object identifier will be implemented. *Objid* values are represented by an identifier for the instance relation (*relid*) and the tuple (*oid*). *Relid* is the unique identifier for the tuple in the POSTGRES catalog that stores information about database relations (i.e., the *RELATION* relation). Given an *objid*, the following query will fetch the specified tuple:

RETRIEVE (o.all)
FROM o IN *relid*
WHERE o.oid = *oid*

This query will be optimized so that fetching an object instance will be very efficient.

The *DBClass* relation contains a tuple for each shared class:

CREATE DBClass(Name, Owner) INHERITS (DBObject)

This relation has an attribute for the class name (*Name*) and the user that created the class (*Owner*). Notice that it inherits the attributes in *DBObject* (i.e., *Instance* and *Class*) because DBClass is itself a shared class.

The superclass list for a class is represented in the *SUPERCLASS* relation:

CREATE SUPERCLASS(Class, Superclass, SeqNum)

where

Class	is the name of the class object.
Superclass	is the name of the parent class object.
SeqNum	is a sequence number that specifies the inheritance order in the case that a class has more than one superclass.

The superclass relationship is stored in a separate relation because a class can inherit variables and methods from more than one parent (i.e., multiple inheritance). The sequence number is required to implement the name conflict resolution rule.

Methods are represented in the METHODS relation:

CREATE METHODS(Class, Name, Source, Binary)

where

Class	is the *objid* of the class that defines the method.
Name	is the name of the method.
Source	is the source code for the method.
Binary	is the relocatable binary code for the method.

Method code is dynamically loaded into the application program as needed. Method determination and caching are discussed below.

Object instances are represented by tuples in the instance relation that has an attribute for each instance variable. For example, if the classes *Equipment*, *Furnace*, and *Tylan* shown in figure 3 were defined with metaclass *DBClass*, the relations shown in figure 7 would be created in the database. When an OBJFADS application creates an instance of one of these classes, a tuple is automatically appended to the appropriate instance relation. Notice that to create a shared class, the superclass of *Equipment* must be changed to *DBObject*.

The POSTGRES data model uses the same inheritance conflict rules for attributes that CLOS uses so attribute inheritance can be implemented in the database system. If the rules were different, OBJFADS would have to simulate data inheritance in the database or POSTGRES would have to be changed to allow user-defined inheritance rules as in CLOS.

```
CREATE Equipment(Location, Picture, DateAcquired)
INHERITS (DBObject)

CREATE Furnace(NumberOfTubes, MaxTemperature)
INHERITS (Equipment)

CREATE Tylan()
INHERITS (Furnace)
```

Figure 7: Shared object relations.

Thus far, we have not described how OBJFADS data types (i.e., Common Lisp data types) are mapped to POSTGRES data types. Data types will be mapped between the two environments as specified by type conversion catalogs. Most programming language interfaces to database systems do not store type mapping information in the database [3,4,6,23,24,27]. We are maintaining this information in catalogs so that user-defined data types in the database can be mapped to the appropriate Common Lisp data type.

The type mapping information is stored in three catalogs: *TYPEMAP*, *OFTOPG*, and *PGTOOF*. The *TYPEMAP* catalog specifies a type mapping and procedures to convert between the types:

CREATE TYPEMAP(OFType, PGType, ToPG, ToOF)

where

OFType	is an OBJFADS type.
PGType	is a POSTGRES type.
ToPG	is a procedure that converts from the OBJFADS type to the POSTGRES type.
ToOF	is a procedure that converts from the POSTGRES type to the OBJFADS type.

The table in figure 8 shows the mapping for selected Common Lisp types. Where possible, Common Lisp values are converted to equivalent POSTGRES types (e.g., *fixnum* to *int4*). In other cases, the values are converted to a print representation when they are stored in the database and recreated by evaluating the print representation when they are fetched into the program (e.g., symbols and functions). We expect over time to build-up a set of user-defined POSTGRES types that will represent the commonly used Common Lisp types (e.g., *list*, *random-state*, etc.). However, we also expect application data structures to be designed to take advantage of the natural database representation. For example, it makes more sense to store a list as a separate relation with a common attribute (e.g., a *PO#* that joins a purchase order with the line items it contains) than as an array of *objid*'s in the database.

Class variables are more difficult to represent than class information and instances variables. The straightforward approach is to define a relation *CVARS* that contains a tuple for each class variable:

CREATE CVARS(Class, Variable, Value)

where *Class* and *Variable* uniquely determine the class variable and *Value* represents the current value of the variable. This solution requires a union

Common Lisp	POSTGRES	Description
fixnum	int4	4 byte integer.
float	float	4 byte floating point number.
(simple-array string-char)	char[]	Variable length character string.
symbol	char[]	A string that represents the symbol (e.g., "'x" for the symbol *x*).
(local) object	char[]	A string that contains a function call that will recreate the object when executed.

Figure 8: Data type mapping examples.

type mechanism because the attribute values in different tuples may have different types. POSTGRES does not support union types because they violate the relational tenet that all attribute values must have the same type.

Two other representations for class variables are possible with POSTGRES. First, a separate relation can be defined for each class that contains a single tuple that holds the current values of all class variables. For example, the following relation could be defined for the *Furnace* class:

FurnaceCVARS(NumberOfFurnaces)

Unfortunately, this solution introduces representational overhead (the extra relation) and requires another join to fetch the slots in an object. Moreover, it does not take advantage of POSTGRES features that can be used to update the count automatically.

The second alternative uses POSTGRES rules. A rule can be used to define an attribute value that appears to the application as if it was stored [34]. For example, the following command defines a rule that computes the number of furnaces:

```
REPLACE ALWAYS Furnace*(
  NumberOfFurnaces = COUNT{Furnace*.Instance})
```

A reference to *Furnace.NumberOfFurnaces* will execute the COUNT aggregate to compute the current number of furnaces. The relation variable *Furnace** in the aggregate specifies that tuples in *Furnace* and all relations that inherit data from *Furnace* (e.g., *Tylan* and *Bruce*) are to be counted. With this representation, the database maintains the correct count. Notice that the command replaces this value in *Furnace** which causes the rule to be inherited by all relations that inherit data from *Furnace*. The disadvantage of this approach is that the COUNT aggregate is executed every time the class variable is referenced.

POSTGRES provides another mechanism that can be used to cache the answer to this query so that it does not have to be recomputed each time the variable is referenced. This mechanism allows the application designer to request that a rule be evaluated early (i.e., precomputed) and cached in the appropriate relation. In other words, the furnace count will be cached in the relations *Furnace*, *Tylan*, and *Bruce* so that references to the variable will avoid recomputation. Updates to *Furnace* or subclasses of *Furnace* will cause the precomputed value to be invalidated. POSTGRES will recompute the rule off-line or when the class variable is next referenced whichever comes first.

Class variables that are not computable from the database can be represented by a rule that is assigned the current value as illustrated in the following command:

```
REPLACE ALWAYS Furnace(x = current value)
```

Given this definition, a reference to *Furnace.x* in a query will return the current value of the class variable. The variable is updated by redefining the rule. We plan to experiment with both the single tuple relation and rule approaches to determine which provides better performance.

This section described the object hierarchy model and a database design for storing it in a relational database. The next section describes the application process object cache and optimizations to improve the time required to fetch an object from the database.

4. Object Cache Design

The object cache must support three functions: object fetching, object updating, and method determination. This section describes the design for efficiently accessing objects. The next section describes the support for

object updating and the section following that describes the support for method determination.

The major problem with implementing an object hierarchy on a relational database system is the time required to fetch an object. This problem arises because queries must be executed to fetch and update objects and because objects are decomposed and stored in several relations that must be joined to retrieve it from the database. Three strategies will be used to speed-up object fetch time: caching, precomputation, and prefetching. This section describes how these strategies will be implemented.

The application process will cache objects fetched from the database. The cache will be similar to a conventional Smalltalk run-time system [13]. An object index will be maintained in main memory to allow the run-time system to determine quickly if a referenced object is in the cache. Each index entry will contain an object identifier and the main memory address of the object. All object references, even instance variables that reference other objects, will use the object identifier assigned by the database (i.e., the *instance* attribute). These indirect pointers may slow the system down but they avoid the problem of mapping addresses when objects are moved between main memory and the database.[5] The object index will be hashed to speed-up object referencing.

Object caching can speed-up references to objects that have already been fetched from the database but it cannot speed-up the time required to fetch the object the first time it is referenced. The implementation strategy we will use to solve this problem is to precompute the memory representation of an object and to cache it in an OBJFADS catalog:

CREATE PRECOMPUTED(Objid, ObjRep)

where

Objid	is the object identifier.
ObjRep	is the main memory object representation.

Suppose we are given the function *RepObject* that takes an object identifier and returns the memory representation of the object. Notice that the memory representation includes class variables and data type conversions. An application process could execute *RepObject* and store the result back in

---

[5] Most Smalltalk implementations use a similar scheme and it does not appear to be a bottleneck.

the *PRECOMPUTED* relation. This approach does not work because the precomputed representation must be changed if another process updates the object either through an operation on the object or an operation on the relation that contains the object. For example, a user could run the following query to update the values of *MaxTemperature* in all *Furnace* objects:

REPLACE Furnace*(MaxTemperature = *newvalue*)

This update would cause all *Furnace* objects in *PRECOMPUTED* to be changed.[6]

A better approach is to have the DBMS process execute *RepObject* and invalidate the cached result when necessary. POSTGRES supports precomputed procedure values that can be used to implement this approach. Query language commands can be stored as the value of a relation attribute. A query that calls *RepObject* to compute the memory representation for the object can be stored in *PRECOMPUTED.Objrep*:

RETRIEVE (MemRep = RepObject($Objid))

$Objid refers to the object identifier of the tuple in which this query is stored (i.e., *PRECOMPUTED.Objid*). To retrieve the memory representation for the object with *objid* "Furnace-123," the following query is executed:

RETRIEVE (object = PRECOMPUTED.ObjRep.MemRep)
WHERE PRECOMPUTED.objid = "Furnace-123"

The nested dot notation (*PRECOMPUTED.ObjRep.MemRep*) accesses values from the result tuples of the query stored in *ObjRep* [36]. The constant "Furnace-123" is an external representation for the *objid* (i.e., the *Furnace* object with *oid* 123). Executing this query causes *RepObject* to be called which returns the main memory representation of the object.

This representation by itself does not alter the performance of fetching an object. The performance can be changed by instructing the DBMS to precompute the query in *ObjRep* (i.e., to cache the memory representation of the object in the *PRECOMPUTED* tuple). If this optimization is performed, fetching an object turns into a single relation, restriction query that can be efficiently implemented. POSTGRES supports precomputation of query language command values similar to the early evaluation of rules described

---

[6] *Furnace* objects cached in an application process must also be invalidated. Object updating, cache consistency, and update propagation are discussed in the next section.

above.[7] Database values retrieved by the commands will be marked so that if they are updated, the cached result can be invalidated. This mechanism is described in greater detail elsewhere [32,33].

The last implementation strategy to speed-up object referencing is prefetching. The basic idea is to fetch an object into the cache before it is referenced. The *HINTS* relation maintains a list of objects that should be prefetched when a particular object is fetched:

CREATE HINTS(FetchObject, HintObject, Application)

When an object is fetched from the database by an application (*Application*), all *HintObject*'s for the *FetchObject* will be fetched at the same time. For example, after fetching an object, the following query can be run to prefetch other objects:

RETRIEVE (obj = p.ObjRep.MemRep)
FROM p IN PRECOMPUTED, h IN HINTS
WHERE p.Objid = h.HintObject
AND h.FetchObject = *fetched-object-identifier*
AND h.Application = *application-name*

This query fetches objects one-at-a-time. We will also investigate precomputing collections of objects, so called *composite objects* [30]. The idea is to precompute a memory representation for a composite object (e.g., a form or procedure definition that is composed of several objects) and retrieve all objects into the cache in one request. This strategy may speed-up fetching large complex objects with many subobjects.

We believe that with these three strategies object retrieval from the database can be implemented efficiently. Our attention thus far has been focussed on speeding up object fetching from the database. We will also have to manage the limited memory space in the object cache. An LRU replacement algorithm will be used to select infrequently accessed objects to remove from the cache. We will also have to implement a mechanism to "pin down" objects that are not accessed frequently but which are critical to the execution of the system or are time consuming to retrieve.

---

[7] The POSTGRES server checks that the command does not update the database and that any procedures called in the command do not update the database so that precomputing the command will not introduce side-effects.

This section described strategies to speed-up object fetching. The next section discusses object updating.

5. Object Updating and Transactions

This section describes the run-time support for updating objects. Two aspects of object updating are discussed: how the database representation of an object is updated (database concurrency and transaction management) and how the update is propagated to other application processes that have cached the object.

The run-time system in the application process specifies the desired update mode for an object when it is fetched from the database into the object cache. The system supports four update modes: local-copy, direct-update, deferred-update, and object-update. Local-copy mode makes a copy of the object in the cache. Updates to the object are not propagated to the database and updates by other processes are not propagated to the local copy. This mode is provided so that changes are valid only for the current session.

Direct-update mode treats the object as though it were actually in the database. Each update to the object is propagated immediately to the database. In other words, updating an instance variable in an object causes an update query to be run on the relation that represents instances of the object. A conventional database transaction model is used for these updates. Write locks are acquired when the update query is executed and they are released when it finishes (i.e., the update is a single statement transaction). Note that read locks are not acquired when an object is fetched into the cache. Updates to the object made by other processes are propagated to the cached object when the run-time system is notified that an update has occurred. The notification mechanism is described below. Direct-update mode is provided so that the application can view "live data."

Deferred-update mode saves object updates until the application explicitly requests that they be propagated to the database. A conventional transaction model is used to specify the update boundaries. A begin transaction operation can be executed for a specific object. Subsequent variable accesses will set the appropriate read and write locks to ensure transaction atomicity and recoverability. The transaction is committed when an end transaction operation is executed on the object. Deferred-update mode is provided so that the application can make several updates atomic.

The last update mode supported by the system is object-update. This mode treats all accesses to the object as a single transaction. An intention-

to-write lock is acquired on the object when it is first retrieved from the database. Other processes can read the object, but they cannot update it. Object updates are propagated to the database when the object is released from the cache. This mode is provided so that transactions can be expressed in terms of the object, not the database representation. However, note that this mode may reduce concurrency because the entire object is locked while it is in the object cache.

Thus far, we have only addressed the issue of propagating updates to the database. The remainder of this section will describe how updates are propagated to other processes that have cached the updated object. The basic idea is to propagate updates through the shared database. When a process retrieves an object, a database alerter [8] is set on the object that will notify the process when it is updated by another process. When the alerter is trigger by another process, the process that set the alerter is notified. The value returned by the alerter to the process that set it is the updated value of the object. Note that the precomputed value of the object memory representation will be invalidated by the update so that it will have to be recomputed by the POSTGRES server. The advantage of this approach is that the process that updates an object does not have to know which processes want to be notified when a particular object is updated.

The disadvantages of this approach are that the database must be prepared to handle thousands of alerters and the time and resources required to propagate an update may be prohibitive. Thousands of alerters are required because each process will define an alerter for every object in its cache that uses direct-, deferred-, or object-update mode. An alerter is not required for local-copy mode because database updates by others are not propagated to the local copy. POSTGRES is being designed to support large databases of rules so this problem is being addressed.

The second disadvantage is the update propagation overhead. The remainder of this section describes two propagated update protocols, an alerter protocol and a distributed cache update protocol, and compares them. Figure 9 shows the process structure for the alerter approach. Each application process (AP) has a database process called its POSTGRES server (PS). The POSTMASTER process (PM) controls all POSTGRES servers. Suppose that AP_i updates an object in the database on which $M \leq N$ AP's have set an alerter. Figure 10 shows the protocol that is executed to propagate the updates to the other AP's. The cost of this propagated update is:

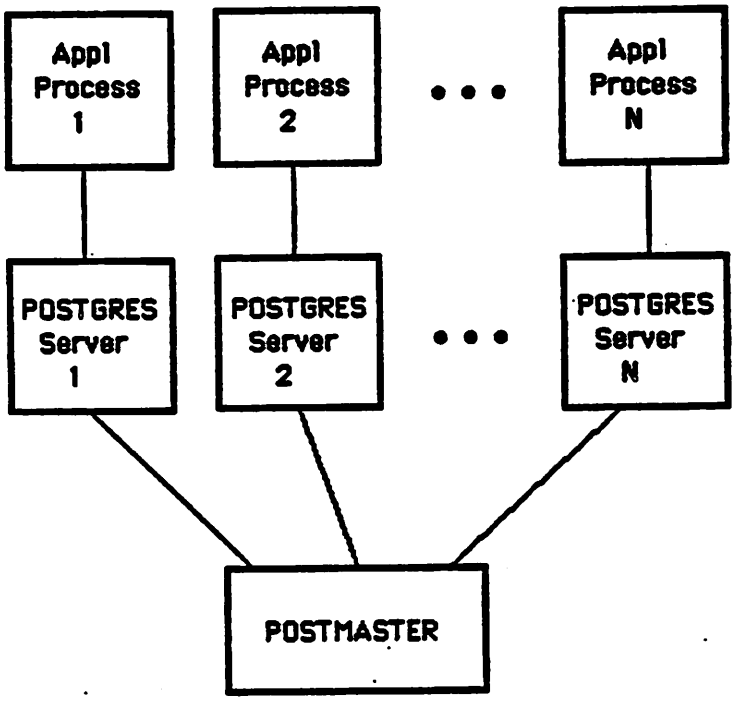

Figure 9. Process structure for the alerter approach.

2M+1	process-to-process messages
1	database update
1	catalog query
1	object fetch

The object fetch is avoidable if the alerter returns the changed value. This optimization works for small objects but may not be reasonable for large objects.

The alternative approach to propagate updates is to have the user processes signal each other that an update has occurred. We call this approach the *distributed cache update* approach. The process structure is similar to that shown in figure 9, except that each AP must be able to broadcast a message to all other AP's. Figure 11 shows the distributed cache update protocol. This protocol uses a primary site update protocol. If AP_i does not have the update token signifying that it is the primary site for

1. AP_i updates the database.
2. PS_i sends a message to PM indicating which alerters were tripped.
3. PM queries the alerter catalog to determine which PS's set the alerters.
4. PM sends a message to PS_j for each alerter.
5. Each PS_j sends a message to AP_j indicating that the alerter has been tripped.
6. Each PS_j refetches the object.

Figure 10. Propagated update protocol for the alerter approach.

1. AP_i acquires the update token for the object.
2. AP_i updates the database.
3. AP_i broadcasts to all AP's that the object has been updated.
4. Each AP_j that has the object in its cache refetches it.

Figure 11. Propagated update protocol for the distributed cache approach.

the object, it sends a broadcast message to all AP's requesting the token. The AP that has the token sends it to AP_i. Assuming that AP_i does not have the update token, the cost of this protocol is:

2	broadcast messages
1	process-to-process message
1	database update
1	object fetch

One broadcast message and the process-to-process message are eliminated if AP_i already has the update token. The advantage of this protocol is that a multicast protocol can be used to implement the broadcast messages in a way that is more efficient than sending N process-to-process messages. Of course, the disadvantage is that AP's have to examine all update signals to determine whether the updated object is in its cache.

Assume that the database update and object fetch take the same resources in both approaches and that the alerter catalog is cached in main memory so the catalog query does not have to read the disk in the alerter approach. With these assumptions, the comparison of these two approaches comes down to the cost of 2 broadcast messages versus 2M process-to-process messages. If objects are cached in relatively few AP's (i.e., $M << N$) and broadcast messages are efficient, the distributed cache update appears better. On the other hand, if M is larger, so the probability of doing 2 broadcasts goes up, and broadcasts are inefficient, the alerter approach appears better. We have chosen the alerter approach because an efficient multicast protocol does not exist but the alerter mechanism will exist in POSTGRES. If this approach is too slow, we will have to tune the alerter code or implement the multicast protocol.

This section described the mechanisms for updating shared objects. The last operation that the run-time system must support is method determination which is discussed in the next section.

6. Method Determination

Method determination is the action taken to select the method to be executed when a procedure is called with an object as an argument. Conventional object-oriented systems implement a cache of recently called methods to speed-up method determination [12]. The cache is typically a hash table that maps an object identifier of the receiving object and a method name to the entry address of the method to be executed. If the desired object and method name is not in the table, the standard look-up algorithm is invoked. In memory resident Smalltalk systems, this strategy has proven to be very good because high hit ratios have been achieved with modest cache sizes (e.g., 95% with 2K entries in the cache) [19].

We will adapt the method cache idea to a database environment. A method index relation will be computed that indicates which method should be called for each object class and method name. The data will be stored in the *DM* relation defined as follows:

CREATE DM(Class, Name, DefClass)

where

Class	is the class of the argument object.
Name	is the name of the method called.
DefClass	is the class in which the method is defined.

Given this relation, the binary code for the method to be executed can be retrieved from the database by the following query:

```
RETRIEVE (m.Binary)
FROM m IN METHODS, d IN DM
WHERE m.Class = d.DefClass
  AND d.Class = argument-class-objid
  AND d.Name = method-name
```

The DM relation can be precomputed for all classes in the shared object hierarchy and incrementally updated as the hierarchy is modified.

Method code will be cached in the application process so that the database will not have to be queried for every procedure call. Procedures in the cache will have to be invalidated if another process modifies the method definition or the inheritance hierarchy. Database alerters will be used to signal object changes that require invalidating cache entries. We will also support a check-in/check-out protocol for objects so that production programs can isolate their object hierarchy from changes being made by application developers [15].

This section described a shared index that will be used for method determination.

7. Summary

This paper described a proposed implementation of a shared object hierarchy in a POSTGRES database. Objects accessed by an application program are cached in the application process. Precomputation and prefetching are used to reduce the time to retrieve objects from the database. Several update modes were defined that can be used to control concurrency. Database alerters are used to propagate updates to copies of objects in other caches. A number of features in POSTGRES will be exploited to implement the system, including: rules, POSTQUEL data types, precomputed queries

and rules, and database alerters.

References

1. R. M. Abarbanel and M. D. Williams, A Relational Representation for Knowledge Bases, Unpublished manuscript, Apr. 1986.
2. H. Afsarmanesh and et. al., "An Extensible, Object-Oriented Approach to Databases for VLSI/CAD", *Proc. 11th Int. Conf. on VLDB*, Aug. 1985.
3. A. Albano and et. al., "Galileo: A Strongly-Typed, Interactive Conceptual Language", *ACM Trans. Database Systems*, June 1985, 230-260.
4. E. Allman and et. al., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language", Proc. of a Conf. on Data: Abstraction, Definition, and Structure, *SIGPLAN Notices*,, Mar. 1978.
5. T. Anderson and et. al., "PROTEUS: Objectifying the DBMS User Interface", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.
6. M. P. Atkinson and et. al., "An Approach to Persistent Programming", *Computer Journal 26*, 4 (1983), 360-365.
7. D. Bobrow and G. Kiczales, "Common Lisp Object System Specification", Draft X3 Document 87-001, Am. Nat. Stand. Inst., February 1987.
8. O. P. Buneman and E. K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Trans. Database Systems*, Sep. 1979, 368-382.
9. G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proc. 1984 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1984.
10. U. Dayal and et.al., "A Knowledge-Oriented Database Management System", *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.
11. N. P. Derrett and et.al., "An Object-Oriented Approach to Data Management", *Proc. 1986 IEEE Spring Compcon*, 1986.
12. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, May 1983.

13. T. Kaehler, "Virtual Memory for an Object-Oriented Language", *Byte 6*, 8 (Aug. 1981).

14. T. Kaehler and G. Krasner, "LOOM – Large Object-Oriented Memory for Smalltalk-80 Systems", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, Reading, MA, May 1983.

15. R. Katz, "Managing the Chip Design Database", *Computer Magazine 16*, 12 (Dec. 1983).

16. J. Kempf and A. Snyder, "Persistent Objects on a Database", Report STL-86-12, Sftw. Tech. Lab., HP Labs, Sep. 1986.

17. S. Khoshanfian and P. Valduriez, "Sharing, Persistence, and Object Orientation: A Database Perspective", DB-106-87, MCC, Apr. 1987.

18. G. L. Krablin, "Building Flexible Multilevel Transactions in a Distributed Persistent Environment", Persistence and Data Types, Papers for the Appin Workshop, U. of Glasgow, Aug. 1985.

19. G. Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, Reading, MA, May 1983.

20. D. Maier and J. Stein, "Development of an Object-Oriented DBMS", *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986.

21. F. Maryanski and et.al., "The Data Model Compiler: a Tool for Generating Object-Oriented Database Systems", Unpublished manuscript, Elect. Eng. Comp. Sci. Dept., Univ. of Connecticut, 1987.

22. N. Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework", *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986, 186-201.

23. J. Mylopoulos and et. al., "A Language Facility for Designing Interactive Database-Intensive Systems", *ACM Trans. Database Systems 10*, 4 (Dec. 1985).

24. L. A. Rowe and K. A. Shoens, "Data Abstraction, Views, and Updates in Rigel", *Proc. 1979 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, Boston, MA, May 1979.

25. L. A. Rowe and M. R. Stonebraker, "The POSTGRES Data Model", to appear in *Proc. 13th VLDB Conf.*, Britton, England, Sep. 1987.

26. L. A. Rowe and C. B. Williams, "An Object-Oriented Database Design for Integrated Circuit Fabrication", submitted for publication, Apr. 1987.

27. J. Schmidt, "Some High Level Language Constructs for Data of Type Relation", *ACM Trans. Database Systems 2*, 3 (Sep. 1977), 247-261.

28. A. H. Skarra and et. al., "An Object Server for an Object-Oriented Database System", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

29. G. L. Steele, *Common Lisp - The Language*, Digital Press, 1984.

30. M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations", *The AI Magazine 6*, 4 (Winter 1986), 40-62.

31. M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1986.

32. M. R. Stonebraker, "Object Management in POSTGRES Using Procedures", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

33. M. R. Stonebraker, "Extending a Relational Data Base System with Procedures", to appear *ACM TOD*, 1987.

34. M. R. Stonebraker, E. Hanson and C. H. Hong, "The Design of the POSTGRES Rules System", *IEEE Conference on Data Engineering*, Los Angeles, CA, Feb. 1987.

35. S. M. Thatte, "Persistent memory: A Storage Architecture for Object-Oriented Database Systems", *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

36. C. Zaniola, "The Database Language GEM", *Proc. 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA., May 1983.