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 forsee 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.

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 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]. 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:

```
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

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

‘‘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. 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

Figure 3. POSTGRES process structure.

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.

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’’

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

Figure 4. Compatibility modes for I locks.

	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.

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 ‘‘&’’ 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.

	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.

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 $F * 1000 / X \ll 1$. In this case, checking the disposition of a transaction in LOG will cause a page fault with probability:

$$\text{FAULT (LOG)} = 1 - [B / Q]$$

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

$$P(\text{XACT}) = 1 - (B * X) / (Q * F * 1000)$$

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:

$$\text{FAULT (XACT)} = P(\text{XACT}) + X / 1000$$

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

$$\text{delta} = \text{FAULT (LOG)} - \text{FAULT (XACT)}$$

is maximized by choosing:

$$X = 1000 * \text{square-root} (F)$$

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

Figure 7. Expected number of faults versus buffer size.

space or when there is nearly enough to hold the whole log. Moreover

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

and if

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.