4.3 POSTGRES

POSTGRES (Stonebraker, 1986a) is an extended relational model DBMS, originally developed at the University of California, Berkeley by Prof. Michael Stonebraker and his group, and was later commercialised under the name of "ILLUSTRA". The prototype POSTGRES is available in the public domain.

POSTGRES illustrates how a relational DBMS (INGRES) could be successfully extended to support the requirements of a new range of database applications which called for features of object orientation. The main intent is to achieve the objectives of object orientation by making as few changes as possible in the original relational model of Codd while, at the same time, incorporating numerous desirable features that have been identified in the database literature and many of them being prototyped in the previous relational DBMS, INGRES (whose new extended version is POSTGRES). This approach has the obvious advantages of providing continuity with the conventional relational model and yet providing object-oriented features. Thus, relational applications could naturally be made to migrate to an object-oriented environment.

4.3.1 The Design and Architecture of POSTGRES

The design goals of POSTGRES as proclaimed by its developers (Stonebraker, 1986a) are:

1. To provide better support for complex objects.
2. To provide user extendability for data types, operators and access methods.
3. To provide facilities for active databases as alerters and triggers, and inferencing with forward and backward chaining.

POSTGRES provides extensions to support definitions of abstract data types, instances of which can then participate in the values of columns in relations (or tables). The basic techniques for adding new data types was originally explored by Stonebraker (Stonebraker, 1986).

POSTQUEL and data definition in POSTGRES

POSTQUEL is a language supported by POSTGRES. The original relational model of Codd being preserved without change, most of QUEL of INGRES was also retained. In QUEL and in POSTQUEL the database is a collection of relations (or tables) that contain tuples with the same fields defined, and the values in a field have the same data type.

As most of the original QUEL is retained unchanged, the following commands of QUEL 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 specifies the storage structure for a relation has been omitted because all relations are stored in a particular structure defined to support historical data. The INDEX command is retained so that other access paths to the data can be defined (It is implicit, here, that the reader be familiar with QUEL of INGRES; for details on INGRES, see Stonebraker (1976)).

Even though the basic structure of POSTQUEL is similar to that of QUEL, numerous extensions have been made to support complex objects, user defined data types and access methods, time varying data (i.e. various snapshots and historical data), iteration queries, alerters, triggers and rules.

The following built-in data types are provided:

1. Integer
2. Floating Point
3. Fixed Length Character String
4. Unbounded varying length arrays of fixed types with an arbitrary number of dimensions
5. POSTQUEL
6. Procedure.

Scalar type fields (Integers, float pt, fixed length character strings) are referenced by conventional (INGRES) dot notation (e.g. EMP.name).

Variable length arrays are provided for applications that need to store large homogeneous sequences of data (for example, in signal processing, image or voice data) usually required in multimedia applications.

Fields of the type POSTQUEL contain a sequence of data manipulation commands. They are referenced by means of conventional dot notation. But 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.bathing-avg).

Fields of the type procedure contain procedures written in a general purpose programming language with embedded data manipulation commands such as EQUEL or Rigel language. Fields of the type procedure and POSTQUEL can be executed using EXECUTE command.

For example, for a relation EMP (name, age, salary, hobbies, dept) in

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

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

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

Fields of the type POSTQUEL and "Procedure" can be used to represent complex objects with shared sub-objects and to support multiple representations of data.

In addition to the foregoing in-built data types, POSTQUEL also permits user defined data types. New data types and operators can be defined with the user defined data type facility.

Complex objects

In POSTQUEL, fields of type POSTQUEL can be utilised 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 sub-objects. For example, if a complex object comprising simple objects such as POLYGON, CIRCLE, LINE is required, it can be defined as follows, for the relations Circle (id,other fields), Line(id,Other fields)

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

The simple values of the relation are:

```
object(a) retrieve (POLYGON.all)
          where POLYGON.id = 10
          retrieve(CIRCLE.all)
          where CIRCLE.id = 40

object(b) retrieve(LINE.all)
          where LINE.id = 17
          retrieve(POLYGON.all)
          where POLYGON.id = 10
```

Both object(a) and object(b) above share polygons with id10.

Multiple representations of data are useful for caching data in a data structure that is better suited to a particular use, while retaining the ease of access via a relational representation. Multiple representations can be supported by defining a procedure that translates one representation (e.g. relational representation) to another (e.g. display list for graphic display). The translation procedure is stored in a database.

For the example considered earlier on, OBJECT relation would have an additional field named "display" that would contain a procedure which creates a display list for an object stored in POLYGON, CIRCLE and LINE as:

```
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 sub-objects which make up the object and that creates the display list representation for the object.

Thus, the code is repeated for every OBJECT tuple and the C procedure replicates the queries stored in the object field to retrieve the sub-objects.

This problem can be solved by storing the procedure in a separate relation (i.e. normalising 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

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

This command executes the procedure to create an alternative representation and passes to it the name of the object.

Time varying data and versions

Historical data and versions of data can be handled in POSTGRES. While regular retrievals always access the current types in the relation (or table), historical data can be accessed by indicating the desired time when defining a tuple variable, e.g.

```
retrieve (E.all)
from E in EMP ["Jan 1, 1995"]
```

This retrieves all records of all employees who worked on Jan 1, 1995. Here the FROM clause is similar to the SQL mechanism for defining tuple variables and replaces the range command of QUEL. As the RANGE command defined the tuple variable only for the duration of the current user program, it was removed. Since, in POSTQUEL, queries can now be stored as the value of a field, the scope of tuple variable definitions must be constrained. The FROM clause defines the scope of the definition like that of the current query itself.

Snapshot or the value of data as of a particular time-stamp can be retrieved by using the square bracket [time-stamp] notation. The system shall search back, through the historical data, to find the appropriate tuples bearing the specified time-stamp. When the user gives the RETRIEVE-INTO command the snapshot is materialised and a copy of the data is made into another relation.

If the historical data is not required to be stored for any given application, the same can be achieved by specifying a cut-off point by using DISCARD command for a relation and the data older than the cut-off point

is deleted from the database, e.g. 'discard EMP before "one year"' deletes all the data in EMP relation older than one year.

The command 'discard EMP before "now"' and 'discard EMP' retain only current data in EMP and delete all past data.

The range of time stamps also can be specified:

relation name [date 1, date 2]

specifies the relation containing all tuples that were in the relation at the same time between date 1 and date 2. Either of these limits 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 ever there in the relation (i.e. relation-name [ ]), e.g.

```
retrieve (E.all)
from E in EMP [ ]
where E.name = 'Kumar'
```

Retrieves all data on employees named 'Kumar' whoever worked for the company.

The memory hierarchy of POSTQUEL comprises (a) main memory, (b) secondary memory (disk), (c) tertiary memory (optical disk, used for historical data processing). Current data is stored in the secondary memory and the historical data migrates to the tertiary memory. This architectural hierarchy of the memory system is, however, transparent to the user and the user need not be concerned about where some particular historical data are available.

Versions are supported by POSTGRES. The user can create a version from a relation or 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 NEW VERSION command: 'new version EMPTEST from EMP' creates a version named EMPTEST that is derived from EMP relation. If the user wants to create a version that is not changed by subsequent updates to the underlying relation, he can create a version of a snapshot. A MERGE command will merge changes made in a version back to the underlying relation. For example, 'merge EMPTEST into EMP' will merge EMPTEST into original EMP.

Iterative queries, alerters, triggers and rules in POSTGRES

Iterative queries are required to support transitive closure, and iteration can be specified by an asterisk (*) to a command that should be repetitively executed. For example, if it is required to construct a relation that includes all the people managed by someone, either directly or indirectly, a RETRIEVE* INTO command is used.

For a given employee relation with a name and manager field: 'create EMP(name = char [20], mgr = char [20]') the following query creates a relation that contains all employees who work for Rao:

```
Retrieve * into SUBORDINATES (E.name, E.mgr)
    from E in EMP,S in SUBORDINATES
where E.name = 'Rao' or E.mgr = S.name
```

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

The * modifier can be appended to any data manipulation command of POSTQUEL, as APPEND, DELETE, EXECUTE, REPLACE, RETRIEVE and RETRIEVE INTO.

Alerters and Triggers can be specified by adding the keyword 'ALWAYS' to a query, e.g.

```
Retrieve always (EMP.all)
where EMP.name = 'Rao'
```

retrieves the data to the program that issued it whenever Rao's employee record is changed. A trigger is only an update query (i.e. Append, Replace, or Delete) with "always" the key word. For example, the command:

```
Delete always DEPT dname where count (tmp.name by DEPT.name) = 0
```

defines a trigger that will delete DEPT records for those departments which have no employees. While alerters and triggers run independently whenever the qualifying condition is satisfied, the iterative queries run for the specified number of times until they cease to have an effect. ALWAYS supports a forward chaining control structure in which an update wakes up a collection of alerters and triggers which wake up other commands in turn, and so on, until no new commands are awakened. Backward chaining is also supported in POSTGRES. Inferencing is conventionally supported by extending the view mechanism, or its equivalent, with the necessary additional capabilities.

The conventional canonical example of the definition of ANCESTOR relation based on stored relation PARENT:

```
PARENT(parentof, offspring)
```

ancestors can be defined as follows:

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

ANCESTOR view is defined by multiple commands that may involve recursion.A query

```
retrieve (ANCESTOR .parent-of)
where ANCESTOR .offspring = 'Kumar'
```

is processed by extensions to a standard algorithm to generate a recursive command or sequence of commands on stored relations.

Programming language interface and portals

In POSTGRES, programming language interface called HITCHINGPOST is provided to meet selected objectives such as the requirements of browsing style applications, so that all programs needing to access the database, including the ad hoc terminal monitor and any pre-processors for embedded query languages, could be written with HITCHINGPOST and finally, to provide all 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 'PORTAL' is provided so as to allow the program to retrieve data from the database. A portal is similar to a cursor 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.

A portal is defined by the RETRIEVE PORTAL or EXECUTE PORTAL command. The command

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

is passed to the back-end process which generates a query plan to fetch the data. The program can now issue new commands to fetch data from the back-end process to the front-end process or change to '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 back end into the buffer by executing a FETCH command, for example,

```
fetch 20 into P
```

fetches the first twenty records in the portal into its back-end program. These records can be accessed by subscript and field references on P such as P(i) refers to the ith record returned by that fetch command and P(i).name refers to 'name field' in the ith record. Subsequent fetches replace the previously fetched data in the front-end program buffer.

The concept of portal means 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. Portals differ from cursors in the methodology of data update. Once a cursor is partitioned on a record it can be modified or deleted (i.e. updated directly). Data in portal cannot be updated directly. It is updated by DELETE or REPLACE commands on the relations from which the portal data is taken.

In addition to RETRIEVE PORTAL command, portal can be defined by an EXECUTE command, for example, if the EMP relation had a field of type POSTQUEL named 'hobbies', then EMP (name, salary, age, hobbies), is the

relation that contained commands to retrieve persons hobbies from the relations

SOFTBALL (name, position) and
COMPUTERS (name, brand name).

An application program can define a portal that will range over the tuple's description of a person's hobbies as:

execute portal (EMP.hobbies)
where EMP.name = "Rao"

This command defines a portal 'H' that is bound to Rao's hobby records. Since a person can have several hobbies represented by more than one retrieve command in the 'hobbies' field, the records in the buffers may have different types.

Compilation and fast-path

Performance enhancement of query processing is achieved in POSTQUEL by two facilities: query compilation and fast-path. Any POSTQUEL command including portal commands can take advantage of these facilities. A system catalog is available for the purpose of storing queries that are to be compiled. The concerned application program can store these queries in the catalogue which has the structure

CODE (id, owner, command)

where 'id' and 'owner' fields form a unique identifier for each stored command. The program 'Command' field holds the command that is to be compiled. The program can execute the commands entered in the catalog. The real reason for higher speed performance is the 'compilation demon' which always examines the entries in CODE catalogue in every database and compiles the queries. Assuming the commands in CODE are already compiled, the query processing will be faster because the time to parse and optimise the query is avoided.

Compiled queries are faster than regular queries that are parsed and optimised at run time. Even this may not be fast for certain applications, since the EXECUTE command that involves the compiled queries must still be processed. Therefore, a fast path facility is available to avoid this overhead. Since the only variable information in the EXECUTE command is the argument list and the unique identifier for selecting the query to run, HITCHINGPOST has a run-time routine that allows this information to be passed to the back end in a binary format.

POSTGRES system architecture

POSTGRES is said to be built as 'process-per-user' architecture (instead of a server model). The process structure of POSTGRES is shown in Fig 4.1.

The POSTMASTER (box 1 in Fig. 4.1) contains the "demons" that will perform various database services (such as asynchronously compiling user

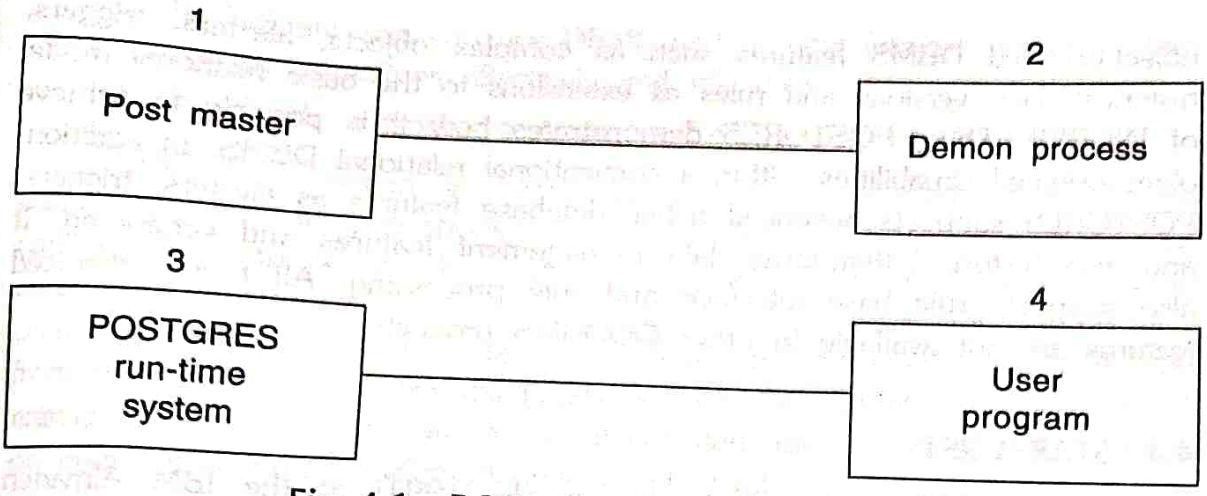

Fig. 4.1 POSTGRES process structure.

commands) and also contains the lock manager. There will be one POSTMASTER per machine and will be started at "sysgen" time.

The POSTGRES run-time system (box 3 in Fig 4.1) executes commands on behalf of one application program. Since a program can have several commands executing at the same time, the message protocol between the program and back end will use the single request-answer model. The request message will have a command designator and a sequence of bytes that contain arguments. The reply (answer) message format contains a response code and any other data requested by the command.

The access methods of POSTGRES will be both system inbuilt and also user defined based on the user defined data types. For procedural data computations, precompilation is used for (a) compiling an access plan for POSTQUEL commands, and (b) executing an access plan to provide an answer. Both these steps are performed when a collection of POSTQUEL commands is executed. The plan and the answer will be put in cache and for small answers the cache value will be placed in the field itself. For larger answers, the answer is put in a relation created for this purpose and its name is put in the field to serve as a pointer.

A 'demon' will run in the background mode and compile plans utilising 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 of the protection status of the command and whenever a procedural field is executed, the run-time system will ensure that the user is authorised for it. In the case of 'fast-path' the run-time system will require that the executing user and the defining user be the same and therefore, no run-time access to system catalogues is required. The same 'demon' will also precompute values of the answers. This concludes the description of POSTGRES system architecture, in brief.

Conclusion

In this section we have surveyed the features, capabilities and functionalities of POSTGRES, the first relational extension DBMS which provided advanced

object-oriented DBMS features such as complex objects, alerters, triggers, historical data, versions and rules as extensions to the basic relational model of INGRES. Thus, POSTGRES demonstrates how it is possible to achieve object-oriented capabilities within a conventional relational DBMS. In addition, POSTGRES supports advanced active database features as alerters, triggers, and also historical (temporal) data management features and versioning. It also supports rule base interface and rule processing. All these advanced features are not available in other OODBMS products.