Evaluation of the Object–Relational DBMS Postgres .I. Administrative Data

M.J.Kim, D.A.Nelson & B.N.Rossiter
Computing Science
Newcastle University NE1 7RU

October 1994

Abstract

This report reviews the object–relational model in the context of object–oriented and relational databases. The object–relational model is described in detail with particular emphasis on the Postgres data model as produced in work at Berkeley. There are many interesting application areas which can be looked at using Postgres, including those involving complex objects and active databases. The main work reported here is on the use of Postgres for running an administrative database for student records with extensive derived functions to calculate a variety of summary data. A preliminary evaluation of the Postgres system is given.

About the author

Moonja Kim is a senior engineer of ETRI (Electronics Telecommunication Research Institute), South Korea, [1] working on main memory database management systems. She spent one year at Newcastle University from August 1993–August 1994 on secondment from the ETRI.

David Nelson is currently a Ph.D. student in the Department of Computing Science at Newcastle University. His research interests are in databases and category theory.

Nick Rossiter is lecturer in the Department of Computing Science with particular interests in databases and systems analysis.

Suggested Keywords

Object–relational databases, object–oriented constructions, relational model, Postgres, administrative data, evaluation.

[1] Current address: Database Section, ETRI, P.O.Box 106, Daeduk Science Town, Daejeon, 305-606, S. Korea

1 Introduction

The relational model, first formulated by Codd [1970], has established itself as the paramount data model for administrative data processing in the 1990s. The reasons for this are its simplicity, its relative ease of use and the natural manner in which tabular structures can be searched, viewed and updated. Further, the sound mathematical basis of functions, relations and set operations, has enabled standard formulations (as manifested in SQL-1 and SQL-2 [ANSI 1986]) to be achieved clearly and unambiguously in a relatively painless manner.

Yet from the early 1980s, it was recognized that the relational model had limitations in dealing with complex objects, that is data structures with considerable internal nesting. Weaknesses were exposed for the standard relational model in applications such as engineering data [Lorie 1985] and office systems and textual applications [Heather & Rossiter 1987; Stonebraker 1986]. Another weakness was the lack of object identifiers, acknowledged in a further paper by Codd on an enhanced relational model RM/T [Codd 1979]. These were vital in some applications where value-oriented keys would require the concatenation of a considerable number of attributes to give a unique set which even then could not be guaranteed to give uniqueness for all future additions to the database. In RM/T, Codd also described how the relational model could be extended to handle abstractions such as inheritance and aggregation through the creation of additional relational structures representing generalization-specialization directed graphs and composition routes respectively.

¿From the perspective of complex object handling, the major limitation imposed by Codd's 1970 model is that of first normal form (1NF): all attribute values must be atomic. There is no fundamental reason for this rule. Mathematical relations can be nested to any degree but the 1NF rule does make it easier to formulate relational languages in algebra and calculus to search the tables. However, the mathematical basis for handling nested relations is now fully developed [Roth, Korth & Silberschatz 1988] and it is mainly implementation problems that have delayed their introduction. The use of nested relations does actually make some operations inherently easier: for instance, referential integrity can be more readily guaranteed in update tasks as if there is nowhere to attach a new object, it is clear that the presumption under which the data is being added needs to be revised. The use of nested relations also removes the need for multivalued dependencies to be handled as a special case of dependency. The codomain (range) of a functional dependency can be typed as single- or multiple-valued enabling fourth normal form (4NF) to be forgotten.

Work through the 1980s on persistent data systems by Atkinson et al [Atkinson, Chisholm & Cockshott 1981] highlighted another weakness in the conventional database approach: that variables defined in a host language accessing the database were from a different world to those within the database. Moreover, the type systems within, say, a relational database system of the 1980s were very limited being restricted to basic types such as integer, real, string and date. These factors give rise to the impedance mismatch in which managing the incompatible type systems in the database and the programming language requires considerable effort by the programmer in designing mapping rules and techniques.

A major influence on all software in recent years has been the acceptance by much of the programming community of the object–oriented paradigm, developed from work on Simula and Smalltalk. This provides many of the facilities identified above as lacking in the relational data model of Codd: complex objects, object identity, abstractions such as inheritance and composition and a unified data-typing system in the database and the programming language.

The database community [2] has reacted with some caution to the object–oriented paradigm in its present form and the early object–oriented databases have not yet convinced the community that this is the only way forward. The basic problem is that database workers are used to a formal framework for their activities which is certainly lacking at present at the global level in the object–oriented paradigm. Yet, as outlined above, database workers do see the need for enhanced functionality of their systems to meet new application requirements and to avoid some of the cumbersome procedures inherent in the current approaches. We therefore see a diversity of approaches at the present time to database systems for the late 1990s:

1. The relational model of Codd as presented in the first SQL standard SQL-1; this model is adequate for applications which can be represented in un-nested tabular structures and where manipulation is mainly updating, searching or simple reporting.

2. The object-relational model as presented in the next SQL standard SQL-3, and in research systems such as Postgres [Stonebraker, Anton & Hanson 1987] and in the commercial derivative from Postgres of Montage; this model expands the market from SQL-1 to handle complex objects and abstractions such as inheritance and aggregation.

3. The semantic approaches such as the Entity–Relationship and functional models; these approaches differ from the first two in not being based on relations as sets of data but on directed graphs explicitly defining access paths and relationships.

4. The object-oriented approach based on extending object–oriented programming language concepts to persistent data systems; the main strength of this approach is the unification of data structure and behaviour and the avoidance of any impedance mismatch.

[2] as portrayed through the panel session "Object–oriented databases: the worst of both worlds?" at the 12th BNCOD at Guildford in July 1994

There is very great debate at present on which of the last three approaches is the most viable. Clearly the object-relational method has the advantage of building upon a large relational user community. However, there are a number of areas where more work needs to be done or where further information is required:

- A universally accepted formal basis has yet to be developed.
- Performance needs to be assessed. A nested relational approach could actually be faster in searching for information than a 1NF system because clustering is an inherent part of the model.
- The ease of use, functionality and naturalness of object–relational systems needs to be determined relative to relational and object–oriented systems.

The formal basis is being investigated in other work at Newcastle [Nelson, Rossiter & Heather 1994]. The work described here investigates mainly the third point – the functionality and usability of an object–relational system, Postgres, with administrative data. The version of Postgres used is 4.1. The application is the storage and retrieval of data for first–year computing science students at Newcastle University. Manipulation requirements include the need to compose final marks and performance indicators from base data. The intention is that all of this should be done within the database system rather than by external host language routines.

Another student (A.R. Davidson) has just completed analysis, design and implementation of the same task using the standard relational system Ingres, so there is good scope for a comparison of the two approaches. Although an analysis was performed by Davidson [1993] using standard systems analysis techniques (data flows, entity-life histories, entity-relationships), the current work will revise some of the earlier models in the light of changes in the manner in which the course is run in the 1993/94 academic year.

First we review the development of object–relational systems in general before giving specific information on the Postgres system.

2 The Object-Relational Database Model

Object-relational database systems have evolved as a practical solution to the numerous problems that have been associated with the new-generation object-oriented database systems. Aspects such as closure, views and query languages have led to developers returning to relational ideas, which encompass these features in a natural and formal manner, but also incorporating features that have grown out of object-oriented database systems, such as inheritance, aggregation and better handling of complex objects.

This section will look in particular at the object-relational database system Postgres [Rowe & Stonebraker 1988], a recent research development from Berkeley which has added object-oriented features to the relational model, while still maintaining the look and feel of an Ingres type database. After discussing Postgres, we will then briefly describe the features of other object-relational systems, such as Montage [Olson 1994], which is a commercial development of the Postgres system, Matisse [Matisse 1994], UniSQL [Kim 1994] and OpenODB.

2.1 The Postgres Database System

2.1.1 Introduction and Aims

The Postgres system is a research development from Berkeley, and is intended to supercede the Ingres [Ingres 1994] relational database management system. The intention is to integrate object-oriented features into a database system, while still maintaining its relational database background.

The main design goals of Postgres [Stonebraker & Rowe 1986] 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;
6. make as few changes as possible (preferably none) to the relational model.

Postgres is still relationally based, but with the added ability to store complex objects. Complex objects are supported by the ability to define abstract types, and to use these as a type in a column of a relation. Querying is provided via a new query language, known as PostQUEL [Kemnitz 1993; Rhein et al 1990a; Rhein et al 1990b]. This is heavily based on the QUEL [Ingres 1994] relational calculus, with extensions to deal with the object-oriented concepts integrated into Postgres.

We will continue by describing the main concepts of the Postgres data model, and then examine data manipulation, describing both the PostQUEL and C forms of querying.

2.1.2 The Data Model

The Postgres data model is based on the relational data model with the addition of classes, inheritance, types and functions [Rowe & Stonebraker 1988]. Whereas in the relational model, the fundamental notion is the relation, or table, in Postgres it is the class: a named collection of instances of objects [Rhein et al 1990a] used for defining complex types. Tables in Postgres are handled in the same manner as in a relational database. Classes represent abstract data types (ADTs) in Postgres, a feature which has been incorporated to some extent in more recent relational systems, although with varying degrees of success. By treating them as a principle concept, Postgres handles ADTs in a more natural manner, incorporating such concepts as inheritance and user-defined operators and procedures.

The Postgres model allows an ADT class to define the type or intension [Ullman 1988] of a column in a relational table, thereby allowing complex data to be stored in a field of a table. Allowing ADTs to be incorporated in tables provides a method of simulating features from object-oriented and semantic database models, such as aggregation, generalization, complex objects with shared subobjects, and attributes that can reference tuples in other relations [Rowe & Stonebraker 1988]. In this way, attributes in a table can be user–defined types, operators, and programming language functions, or procedures.

A column in a table can be of type procedure, which means that a value in a tuple can be determined by some operation on other values in the tuple, relation or database. Functions in Postgres are very similar to derived functions in functional databases [Shipman 1981] or virtual fields in CODASYL databases.

There are two kinds of type in Postgres: atomic and structured [Rowe & Stonebraker 1988]. Atomic types are those defined as ADTs, whereas structured types define complex data, such as arrays and procedures. Arrays in Postgres can be of undefined size, but they can only store elements of the same type.

Tables can have a key, which may be a composition of attributes. The attributes of the key may include an ADT, so long as a comparison operator is defined for that ADT. Tables also have an unique system identifier, or object identifier (oid), which uniquely references a particular tuple in the database. These oids can be queried by the user, but may not be updated manually.

When defining a new type in Postgres, it is usually the case that its internal size needs to be defined for optimal storage and access. For large objects (BLOBS) this internal size is not known, but Postgres does support these, by allowing their size to be defined as a variable. Large objects can be supported by using the Unix file system to store each object as a file or by using the inversion file system. The inversion system breaks up a large object into smaller chunks which are accessed using a B-tree. This type of system can guarantee against crashes while providing better performance for accesses.

Postgres supports inheritance of tables. By allowing such inheritance, the semantics of statements such as the 'student is a person' can be handled through table definitions in a more object-oriented manner. When inheriting tables, the key is also inherited. Multiple inheritance is also supported, but if two tables have the same attributes, then any name conflicts are disallowed.

In [Rowe & Stonebraker 1988], the major aim of Postgres is that inheritance can 'easily' be added to a relational database; easily implying that as few changes as possible are made to the underlying relational model. This can be modelled naturally through the class, or ADT, method. They claim that this shows that the major concepts in an object-oriented database can be cleanly and efficiently supported in an extensible relational DBMS.

Another concept which is a 'usual' feature in relational databases, but causes implementation problems in object-oriented databases, is time travel. This is where all past revisions to the database are stored, so a user can see the state that the database was in at a certain time in the past, or even query on attributes between two dates, i.e. 'list all the students who were taking computing between 1 September 1984 and 31 October 1992'. This means that a database can support versioning and snapshots for multi-user use.

2.1.3 Data Manipulation in Postgres

Postgres data manipulation is provided through its own query language PostQUEL. This is an extension to the QUEL [Ingres 1994] relational calculus provided with Ingres, and has been extended to deal with the new features of Postgres such as abstract data types, inheritance, etc. There is also a facility whereby Postgres tables can be manipulated from within C, known as LibPQ, which provides the full functionality of PostQUEL to an external program. Abstract data types, or classes, can also be defined in C, as well as complex procedures, in cases where the PostQUEL language does not provide enough functionality for the task of the procedure.

To allow the Postgres system to be used from within C code, and eventually other programming languages, a feature known as portals [Stonebraker & Rowe 1986] is provided. These are like cursors, in the way that they allow a programming language to retrieve data from the database, but they also allow, through the LibPQ system, the whole functionality of the PostQUEL calculus to be used externally.

The main additions to data manipulation concepts in Postgres are in the provision for user defined operators and functions. Functions can be written to perform complex updates and retrievals on tables and on ADTs, while operators can be defined to provide comparisons on complex objects, such as equality.

PostQUEL supports most of the constructs provided in QUEL, although with some variations in syntax. As well as this, the query language must provide the ability to use the new concepts [Rhein et 1990a; Rhein et al 1990b] which have been added to the

Postgres system, such as being able to retrieve user defined types (or ADTs), which is done by supplying input and output procedures for that ADT; query language and programming language functions, so that the query can use derived data; user defined operators, which are useful for providing operations such as for testing equality of a user defined type; fixed and variable length arrays, whereby it would be useful to look for a particular element in an array, and return all tuples which satisfy the query; functions of an instance, where we want the query to apply to all tables which inherit from the table being queried; and a rules system, so that alerters and triggers can be supported.

PostQUEL functions can also handle iterative queries, where the result of the function can be transitively closed. This means, for example, that a query to return a child's ancestors will not only return the parents, but will also return the grandparents, great grandparents, etc., presuming the information can be derived from the table. This allows recursive, or iterative, querying - something that the relational and object-oriented models do not support very well.

The PostQUEL language also supports the notion of 'time travel' in queries. This provides a means of historical queries, as mentioned previously. By allowing time–varying data, versions and snapshots of data can be stored and queried, although updates to versions are not carried through to the underlying table. Any changes to the underlying table will result in changes to the version, unless that version was initially created from a snapshot of the underlying table.

Because inheritance of tables is supported, the PostQUEL system must be able to handle inherited procedures as well. A table which has inherited a procedure from some parent table must be able to use that procedure on itself, resolving any problems with attributes in the new table, i.e. an attribute may have changed its type, or may have been overridden. Multiple inheritance also has to be supported, including the case arising when a table inherits two procedures with the same name from two different parents. This is handled by a structure called an IPL (inheritance procedure list), which keeps a hierarchical list of procedure instances, so a query can decide which instance of a procedure should be used for a particular query.

Queries can also explicitly follow an inheritance hierarchy. This is different to procedure inheritance, what we want here is for a query to be able to explicitly apply itself to all tables which directly or indirectly inherit from the table that is being queried. For example, for a list of all people, the user could explicitly define the query so that it looked at all students as well. This is something which is not directly supported in relational databases, where if there was to be a student table, then the query would have to be used twice, once for the person table, and then again for the student table.

Rules can also be incorporated into the data manipulation, so that alerters and triggers [Stonebraker & Rowe 1986] can be supported. An alerter is the result of a rule applied to a retrieve query, whereas a trigger is the result of a rule applied to an update. There are actually two rules systems, instance level and query rewrite. An instance level rule is one that will only ever apply to a few instances in the table, whereas a query rewrite rule is one that usually applies to most instances. For performance reasons, it is always best to define the type of a rule.

Finally, transaction management is provided simply by the ability to enclose a range of queries within a *begin ... end* construct. This ensures that in the case of a crash, any changes that were made to the state of the database within the transaction will be aborted, otherwise the transaction commits.

2.1.4 Comparison with other models

The Postgres system, and other object-relational type database systems, attempt to converge the functionality of the current models; relational, object-oriented and in some sense, functional. The model is relationally based, but adds the best features of the object-oriented models, such as inheritance and support for complex objects. The main additions to the relational model are support for non–atomic values and table inheritance.

By incorporating procedures into the query language, the object-relational model can claim to be very close to the functional model. However, the object-relational model adds a feature which semantic and functional models cannot handle very well: the ability to represent data with uncertain structure. Object-oriented models cannot easily deal with objects which have a variety of shared sub-objects, Postgres claims to be better at this. Postgres also claims a performance improvement in representing object-oriented pointer chains as standard relations.

2.2 Other Object-Relational Database Systems

As previously mentioned, Postgres is a research-based object-relational database system, developed at Berkeley. Other object-relational and object-based database systems have recently evolved, some of which we will now briefly discuss.

2.2.1 Montage

Montage [3], is the commercial version of Postgres. The main differences from Postgres are that it uses SQL rather than PostQUEL, provides a more object-oriented view for class definitions and delivers *data blades*, which give sets of predefined data types and functions.

It is claimed [Stonebraker 1994] that Montage adds the following concepts to SQL:

1. unique identifiers

[3] Note that Montage is now known as Illustra.

2. user–defined types
3. user–defined operators
4. user–defined access methods
5. complex objects
6. user–defined functions
7. overloading
8. dynamic extendibility
9. inheritance of data and functions
10. arrays

The Montage systems claims to improve Postgres by adding [Olson 1994] an improved library interface (via data blades), better user defined function support, more security, better support for inheritance, simpler rules by removing the need for an instance level rule system, backup in the case of media crashes, browsers, and a performance speed up of between 2 and 100 times compared to Postgres. Also, most of the features mentioned in the initial Postgres papers, but which were not implemented because of research constraints, have now been developed.

Montage now provides a more object-oriented support for objects, encapsulating attributes, types and functions in a class, making it look less 'table–like'. Composite types are supported. These are like *structs* in C and C++. The constructed types sets, arrays and pointers (refs) are also supported.

Probably the biggest improvement is the provision of data blades, which are libraries of predefined data types, with their associated functions, providing bolt–on units to the Montage system. The four data blades currently available are the foundation data blade, which provides the traditional data types, now over forty; the text data blade, for handling variable length text; the image data blade, for handling rasterized graphic data; and the spatial data blade, for the handling of spatial data.

2.2.2 Matisse

Matisse provides an object–based approach for application development while trying to incorporate some relational features, such as referential integrity, versioning, triggers and rules, etc. It is very object-oriented based, and was developed to model highly complex business applications. There is support for binary large objects. Facilities for C function calls to the API ensure that any language can use the API. The most notable aspect is that it supports automatic inverse relationships, and achieves a high level of schema consistency.

2.2.3 UniSQL/X

The UniSQL system, as its name implies, is based on the SQL language. Although a few object-oriented database systems make this claim, the UniSQL/X database can boast that its SQL language is an extension of the full ANSI SQL language.

The extensions [Kim 1994] that the UniSQL data model provide over relational databases are nested relations, where a column in a relation can be another relation; encapsulation; inheritance hierarchy on relations in a directed acyclic graph format; and provision of sets as attributes of columns, where a set of values can even be of more than one arbitrary data type. The UniSQL system also supports the core object model defined by the OMG and can handle persistency of objects, in that it treats memory resident and persistent objects equally. This is different to other object-oriented systems, such as ObjectStore [Lamb et al 1991] where a class has to be defined as either persistent or memory resident, and it is difficult to interchange between the two.

The main extensions that the UniSQL/X language provide over ANSI SQL are for path queries. This means that queries can be over nested classes, can include methods, can return nested objects and can handle sets.

The UniSQL/M system provides for a multidatabase level, i.e. a multi-user system. It provides the full functionality of UniSQL/X, but for more than one user, and is fully distributed. Databases in UniSQL/M are in fact views of relational UniSQL/X databases.

2.2.4 OpenODB

OpenODB is an object-based database system from Hewlett Packard. Whereas most database systems are developed from scratch, OpenODB is actually built on top of a relational database system Allbase/SQL. This means that the data and code is stored internally in a relational database, and OpenODB keeps all the functionality, consistency and security of the relational database. On top of this, all the required object-oriented features are developed, such as inheritance, complex objects, etc., and mapped onto relational storage.

The definition and manipulation language of OpenODB is known as OSQL, and is a functionally extended semantic superset of SQL. The OSQL language can be both interpreted and provided as an interface in any language that can make C function calls. Some of the features in the OSQL query language are being adopted in the new SQL3 standard [ISO-ANSI 1994] which is currently being produced.

3 The Test Application

The following account is a summary of a description of the application as produced by A.R.Davidson in his M.Sc. Dissertation [Davidson 1993].

3.1 Background Information

This project is concerned with the activities of the academic department of Computing Science at Newcastle University, in particular the administration of the Level I courses. These courses are offered to students in the faculties of science, engineering and arts. Generally under half the students in any one year are actually intending to proceed to a B.Sc. honours degree in computing science; others are from outside the department taking computing as a third subject to broaden their knowledge or combining within a single subject computing with either chemistry, mathematics or surveying.

A pressing reason for automating the administration of the course is the increased intake of undergraduates in recent years, not all of whom are as strongly motivated as students were, say, ten years ago. The department must therefore monitor their progress and attendance in order to identify those who are not making satisfactory progress. The problem has been compounded by the fact that the increased intake has also meant that the existing manual system is no longer adequate in its ability to handle the volume.

General details such as name, degree courses, tutor name, etc, must be held for each student as well as a record of their attendance at practical sessions, their project marks and test results. Staff need easy retrieval of this data to identify those students whose performance and/or attendance fall below acceptable standards. This will be increasingly important if grant authorities start asking for detailed information on student progress.

Currently much of the data is held on paper files and information on each part of the course tends to get scattered and this leads to difficulties in obtaining an integrated view of the progress of all the students on the course. The information that is held on computer is unwieldy to manipulate.

3.2 Detailed Requirements

At present in any one year there are about 220 students taking Level I Computing Science as a first year subject. As described earlier, they are studying a wide variety of degree subjects including Computing Science itself, Microelectronics & Software Engineering, Mathematics, Chemistry, Surveying, and the Combined Honours and General Degrees. Some special courses are available to students in mathematics (Maths with Computing), chemistry (Chemistry with Computing) and surveying (Surveying Science H), which have different requirements in terms of the numbers of projects and practical sessions and in examination regulations.

Information to be held includes:

General Details: student's name, registration number, degree subject, tutor's name, address and e–mail address, student's e–mail address.

Attendances: week numbers for which each student should attend; actual attendances by name and registration number of student. The demonstrators will take a register at each practical session and then enter the details onto the system.

Initiation: for first five weeks of each year, record progress in exercises which are set before sufficient skills are attained to tackle complete projects.

Christmas Test: student's name and registration number, breakdown of marks to give number of questions attempted, number correct, number incorrect and the total mark.

Warnings: student's registration number, nature of warning.

Personal Difficulties: student's registration number, nature of problems.

Projects: A number of projects are set during the year, typically these would be:

- Practice Project – held during first term, not counted in final assessment.
- Project A – set near end of first term for completion at start of second term;
- Project B – completed at end of first semester;
- Project C – held at start of second semester;
- Project D – set near end of second term for completion at start of third term;
- Project E – not always issued, set at start of third term for completion just before examinations period.

The following details will be held for each project:

Summary data: Project name, student name and registration number, total gross marks for each student, penalties for lateness in submission or poor presentation.

Detailed data: A breakdown for each individual on the marks allocated for each part of the project. The assessed components vary from project to project but the basic list contains user manual, maintenance manual, test data, further thoughts, presentation (including spelling), control abstractions, data abstractions, style, layout, functionality, correctness, error handling, user interface and problem statement. These components are generally weighted equally, each being scored out of five on an integer scale from $0 \ldots 5$. Penalties are given in the form of negative marks for late submission or submission in the incorrect manner and format.

4 Modelling

The data structures are represented by the Entity–Attribute–Relationship diagram based on the original model of Chen [1976] as shown in Figure 1. In this figure, the convention adopted is that entity–types are represented by rectangles and relationships by diamond-shapes. The functionality is indicated as 1:1, 1:N or N:M to show the cardinality of the mapping between the various entity–types in the relationships. The membership class is mandatory for entity–types in all relationships except that *Student* may not participate (if sufficiently idle) in the *Performs* and *Registers* relationships.

The table–types corresponding to the E–R diagram of Figure 1 are shown in Figure 2. These are derived using the standard rules:

- each entity–type becomes a table–type;
- each N:M relationship becomes a table–type;
- a 1:N relationship between two entity–types, say A and B respectively, becomes a table–type if the membership class of B is optional.

The overall behavioural aspects of the application are summarized in the overview data flow diagram shown in Figure 3. The convention used here is that the closed rectangles denote external entities, the open rectangles denote files, the circles denote processes and the arrows denote data flows. All components are uniquely named. The part of the application that we are considering in this report is process 0.3 concerned with marking projects. This process is considered to be a functional primitive so it is not decomposed further.

The target of the current work is to design structures and functions in Postgres which handle the following aspects of the application:

1. defining classes to represent the structures in Figure 2;
2. using rules to enforce referential integrity in all defined classes;
3. using functions to define views which allow several classes to be viewed as if they are one single class;
4. using functions to derive data values from basic data entered so that summary performances by students can be automatically generated and maintained as underlying data changes;
5. demonstrating the use of the query language for making routine or *ad hoc* queries.

Figure 1: Entity–Relationship Diagram for Level I Administration

Student(Registration_id, Student_name, Course_name, Tutor_name, Email)
Course(Name, #_of_Weeks, #_of_Projects)
Xmas-test-marks(Student_id, No_question, No_attempted, No_correct, No_incorrect, Total_mark)
Attendance(Week_no, Student_id, Session)
Overall-performance(Student_id, Overall_attendance, Overall_project_marks)
Project_marks(Student_id, Project_name, user_man, maintenance_man, test_data, further_thought, spelling, control_abs, data_abs, style, layout, functionality, correctness, error_handling, user_interface, any_penalty, problem_stat, mark)
Tutor(Name, Room_no, Email, Department)

Figure 2: Table–Types for Level I Administration

5 Implementation in Postgres

5.1 Data Structure

5.1.1 Classes

The first step in the implementation was to create the database on the Newcastle University's Postgres system held on the Peepy machine. Next, the following classes identified during E-R modeling and data design were created using the Postgres *create* command.

The fundamental notion in Postgres is that of *class*, which is a named collection of instances of objects. Each *instance* has the same collection of named attributes and each *attribute* is of a specific type. The following commands are issued to create our classes:

```
create student(registration_id = text, student_name = text,
                 course_name = text, tutor_name = text, email = text)

create course(name = text, num_weeks = float4, num_projects = float4)

create tutor(name = text, room_no = text, email = text, dept = text)

create attendance(week_no = int2, student_id = text, session = char[4])

create project_marks(project_name = text, marker_name = text,
                     student_id = text, problem_stat = float4,
                     user_man = float4, maintenance_man = float4,
```

Figure 3: Overview Data Flow Diagram for Level I Administration

```
                     test_data = float4, further_thought = float4,
                     spelling = float4, control_abs = float4,
                     data_abs = float4, style = float4, layout = float4,
                     functionality = float4, correctness = float4,
                     error_handling = float4, user_interface = float4,
                     any_penalty = float4, mark = float4)

create xmas_test_mark(student_id = text, no_question = int2,
                      no_attempted = int2, no_correct = int2,
                      no_incorrect = int2, mark = int2)

create perform(sid = text, weeks = float4, projects = float4,
sum_projects = float4)
```

Student, course, tutor, attendance, project_marks, and *xmas_test_mark* classes are real (or base) classes whose instances are stored in the database. Each attribute of the *perform* class is derived from other attributes of other classes. This class corresponds to the *Overall–Performance* table–type shown in Figure 2.

5.1.2 Integrity

Postgres provides some automatic validation checks, e.g. integers cannot be put into character fields. The Level 1 Administration system uses the Postgres rule system to provide two further integrity constraints: uniqueness of attribute values and referential integrity.

The *registration_id* attribute values are unique in the *student* class, the following rule defining the uniqueness property of *registration_id* attribute in *student* class. When a record is appended into the *student* class, the record is discarded if the input value of *registration_id* attribute already exists or is NULL.

```
define rule student_unique is
       on append to student
       where student.registration_id = new.registration_id
          or new.registration_id ISNULL
       do instead nothing
```

Now, let us consider the referential integrity [Date 1981]. Here, we have to guarantee that the values occurring for the attribute *course_name* of student are a subset of the values that occur in the attribute *name* of the *course* class. We also need to assure that the values occurring for the attribute *tutor_name* of student are a subset of the values that occur in the attribute *name* of the *tutor* class.

The following rule is used to support the referential integrity amongst *student, course* and *tutor* classes:

```
define rule student_int is
        on append to student
        where course.name !~ new.course_name
        or tutor.name !~ new.tutor_name
        do instead nothing
```

The following rules show the integrity checking to ensure that the values for the *student_id* attribute of the *attendance, project_marks* and *xmas_test_mark* classes must be a subset of the values of the *registration_id* attribute in the *student* class:

```
define rule project_int is
       on append to project_marks
       where student.registration_id !~ new.student_id
       do instead nothing

define rule xmas_integrity is
       on append to xmas_test_mark
       where student.registration_id !~ new.student_id
       do instead nothing

define rule attend_integrity is
       on append to attendance
       where student.registration_id !~ new.student_id
       do instead nothing
```

5.1.3 Functions and Rules

Postgres allows a user to define functions (methods) to the DBMS. There are three different kinds of functions known to Postgres: POSTQUEL functions, C functions, and operators. The Level 1 Administration System uses POSTQUEL functions and C functions.

```
define function tutor_info(language="postquel", returntype=tutor)
       arg is (student)
       as "retrieve (tutor.all)
           where $1.tutor_name = tutor.name"

define function course_info(language="postquel", returntype=course)
       arg is (student)
       as "retrieve (course.all)
           where $1.course_name = course.name"

define function all_attendance(language = "postquel", returntype = float4)
       arg is (student)
       as "retrieve (attended = float4count{attendance.student_id
           where $1.registration_id = attendance.student_id})"

define function sum_project_mark(language = "postquel",
returntype = float4)
       arg is (student)
      as "retrieve (sum_mark = float4sum{project_marks.mark
         where $1.registration_id = project_marks.student_id})"

define function ave_project_mark(language = "postquel",
returntype = float4)
       arg is (student)
       as "retrieve (mark = (perform.sum_projects / perform.projects))
           where $1.registration_id = perform.sid"

define function attend_ratio(language = "postquel", returntype = float4)
       arg is (student)
       as "retrieve (ratio = (perform.all_attends / perform.weeks) *
100::float4)
           where $1.registration_id = perform.sid"

define function xmas_mark(language = "postquel", returntype = int2)
       arg is (student)
       as "retrieve (mark = xmas_test_mark.mark)
           where $1.registration_id = xmas_test_mark.student_id"
```

Figure 4: POSTQUEL functions

Figure 4 depicts the POSTQUEL functions used in the Level 1 Administration System. According to the degree subject of each student, the *course_info* function retrieves some course information by joining the *student* class and the *course* class over *course_name*. It is quite troublesome to do join operation for each retrieve command using a clause such as "where student.tutor_name = tutor.name". So, this join clause can be defined using a POSTQUEL function. As a result, the Postgres system provides an illusion to the user as if a query take place within one class. For example, we show below the command to find the course information.

```
retrieve (Student=s.student_name, ID=s.registration_id,
          Course=s.course_name, Weeks=s.course_info.num_weeks,
          Projects=s.course_info.num_projects)
from s in student
```

The *all_attendance* function is defined to count the number of attendances for each student at the practical sessions. This function has a *student* class as an argument and uses the *float4count* user-defined aggregate for counting a student's attendance. This function can be used to assign values to the *all_attends* attribute in the *perform* class. Below we define the *float4count* aggregate.

```
define aggregate float4count (sfunc2 = float4inc, initcond2 = "0")
```

Because the number of attendances at practical sessions depends on the subject degree for each student, the attendance percentage cannot be derived from an aggregate function. Therefore, *attend_ratio* is defined to get the percentage of attendance for each student. In Figure 4, *100::float4* is used for casting the number *100* to Postgres's *float4* type.

The *sum_project_mark* function sums all project marks for each student. It uses a system built–in aggregate function *float4sum*. This function can be used to assign values to the *sum_projects* attribute in the *perform* class.

In the case of the attendance percentage, the average of project marks cannot be obtained by an aggregate function. The *ave_project_mark* is a function to get the average of all project marks for each student.

The function *xmas_mark* returns a value of *mark* attribute of *xmas_test_mark* class.

```
define function cfunc_project ( language="c",
                               returntype = int2 )
        arg is (project_marks)
        as "/home/peepy/n302544/job/work/cfunc.o"

define function cfunc_xmas_1 ( language="c",
                              returntype = int2 )
        arg is (xmas_test_mark)
        as "/home/peepy/n302544/job/work/cfunc.o"

define function cfunc_xmas_2 ( language="c",
                              returntype = int2 )
        arg is (xmas_test_mark)
        as "/home/peepy/n302544/job/work/cfunc.o"
```

Figure 5: C functions

Figure 5 shows the C functions, which are actually arbitrary C procedures. The value of *mark* attribute in the *project_marks* class is computed using other attributes in the class. The *cfunc_project* can be used to assign values to the *mark* attribute in the *project_marks* class as follows:

```
replace project_marks (mark = cfunc_project (project_marks))
```

In the *xmas_test_mark* class, *no_incorrect* and *mark* attributes are also computed using other attributes in the class. The function *cfunc_xmas_1* is used to compute the value of the *no_incorrect* attribute. The *cfunc_xmas_1* can be used to assign values to the *no_incorrect* attribute in the *xmas_test_mark* class:

```
replace xmas_test_mark( no_incorrect = cfunc_xmas_1(xmas_test_mark))
```

After this, *cfunc_xmas_2* computes the value of mark attribute. The function *cfunc_xmas_2* can be used to assign values to the *mark* attribute in the *xmas_test_mark* class:

```
replace xmas_test_mark( mark = cfunc_xmas_2(xmas_test_mark) )
```

These functions can be defined to Postgres while the system is running. They are dynamically loaded, if needed, during query execution through:

```
load "/home/peepy/n302544/job/work/cfunc.o"
```

Appendix A shows the C functions: *cfunc_project(), cfunc_xmas_1()* and *cfunc_xmas_2().*

The rule has another useful facility, beside integrity checking, in the Postgres system. The rule has an analagous functionality to the function. Let us see again the three C functions in Figure 5. Here, we could directly get the value of the *mark* attribute using rules. Figure 6 shows the rules which serve the same purpose as the C functions in Figure 5.

When someone tries to retrieve the records from *project_mark* class, the value of the *mark* attribute of the current record is first replaced with the computed result of the rules, then the user command *retrieve* is executed. The *xmas_rule_1* and *xmas_rule_2* rules are similarly explained.

5.2 Operation

5.2.1 Data Input

Postgres supports three different data input methods – *append* command, *copy* command and *PQputline()* function of LIBPQ library.

If the input data is small or the input event occurs irregularly, it is convenient to input the data with the *append* command or LIBPQ application program using the *PQputline()* function. However, when someone wants to input a large amount of data at one time, after making out a file for the input data, it is much more efficient to use the *copy* command.

When using the *copy* command, it is very important to point out that the user has the responsibility for the correctness of input data because there is no input error checking facility. Also, data input integrity using rules is possible only when the data is input through the *append* command.

```
define rule project_rule_1 is
       on retrieve to project_marks
       do replace current(mark = current.problem_stat + current.user_man
                    + current.maintenance_man  + current.test_data
                    + current.further_thought + current.spelling
                    + current.control_abs + current.data_abs
                    + current.style + current.layout
                    + current.functionality + current.correctness
                    + current.error_handling + current.user_interface
                    - current.any_penalty)

define rule xmas_rule_1 is
       on retrieve to xmas_test_mark
       do replace current(no_incorrect = current.no_question
                                        - current.no_correct)

define rule xmas_rule_2 is
       on retrieve to xmas_test_mark
       do replace current(mark = current.no_correct
                                - current.no_incorrect)
```

Figure 6: Rules

The Level 1 Administration System used the LIBPQ application program *load_course.c* to input the data for *course* class (see Appendix B). It also used the *copy* command to input the data for the *student, tutor, attendance, project_marks* and *xmas_test_mark* classes.

5.2.2 User Interface

In Postgres, there is no front–end processor for the user interface, but, two interfaces are supported to interact between the Postgres back–end and the user. One is to run the Postgres terminal monitor, which allows a user to interactively enter, edit and execute commands in the query language POSTQUEL. The other is to interact with Postgres from a C function by using a set of library routines, LIBPQ, which connects them through an IPC channel. Because our work has concentrated on the functional efficiency of the Postgres system, the Level 1 Administration System manipulates data using both interfaces – terminal monitor and LIBPQ. The main data manipulations are given below. It can be seen that the queries have an effect as if the result data is retrieved from one class, here *student* class, using user–defined functions. If we had not used these functions, joins would have been required in the query language. For example, the first query requires joins over four classes: *student, attendance, perform* and *xmas_test_mark*.

Manipulation (1) retrieves the overall performance result, allowing the user to display a report showing a student attendance record, project marks and Christmas test result.

```
retrieve (sid = s.registration_id, sname = s.student_name,
      degree = s.course_name, tutor = s.tutor_name,
      attendance = s.all_attendance, attend_ratio = s.attend_ratio,
      project_mark = s.ave_project_mark, xmas_mark = s.xmas_mark)
from s in student
```

(2) retrieves students whose attendances are below acceptable standard (in this case, 60%).

```
retrieve (sname = s.student_name, sid = s.registration_id,
      tutor = s.tutor_name, tutor_email = s.tutor_info.email,
      attendance = s.all_attendance, ratio = s.attend_ratio)
from s in student where s.attend_ratio < 60::float4
```

(3) retrieves students whose project marks are below a certain mark (in this case, 40%).

```
retrieve (sname = s.student_name, sid = s.registration_id,
      tutor = s.tutor_name, tutor_email = s.tutor_info.email,
      project_mark = s.ave_project_mark)
from s in student where s.ave_project_mark < 40::float4
```

(4) retrieves students whose Christmas test marks are below acceptable standard (in this case, 20%).

```
retrieve (sname = s.student_name, sid = s.registration_id,
      tutor = s.tutor_name, tutor_email = s.tutor_info.email,
      mark = s.xmas_mark)
from s in student where s.xmas_mark < 20
```

5.3 Testing

The implementation and testing of level 1 Administration System was made in the Sun4 workstation environment. We used POSTGRES Version 4.1 system and reference documentation dated August 1993 [Kemnitz 1993] and based upon earlier 1990 material [Rhein et al 1990a; Rhein et al 1990b]. A suite of testing programs was used to create the database and classes, input data and check the data integrity using rules. The suite also tested the correctness of data manipulation for the defined functions, aggregates and rules, including the return values of the computing fields. Appendix C shows the testing programs, which were written using the POSTQUEL query language. Appendix D lists a set of LIBPQ application programs.

6 Conclusions and Future Investigations

The functionality of the Postgres object–relational system has been explored in the context of an application involving the administration of student records. The work has shown that robust applications can be developed in this area. The use of POSTQUEL rules ensures referential integrity and provides triggers which check for uniqueness of data so that primary key and candidate key constraints can be checked. POSTQUEL functions can be used to represent joins giving an alternative and probably more efficient way to provide views of several classes as one class than in standard relational systems.

POSTQUEL functions can also be used to derive statistical information, such as sum and average, on attribute values from base values. More complex operations including those involving arithmetic are performed by C functions. Taken together POSTQUEL and C functions give a derived function or virtual element facility so that the overall student performance as produced by the system is always consistent with the latest information input into the system.

Considerable effort and motivation was required to learn the optimum way in which to approach coding in the Postgres environment. Not unexpectedly for a research system, the published papers employed a 'sugared' syntax which was not always available to us. The reference and user manuals are generally more reliable but deal mainly with simple unconnected examples. We should also point out that we are not using the latest version (4.2β) which has some additional facilities but is a test version.

Because of the effort involved in mastering the aspects of the system described in this report, we have included all the source code in the appendices to asist other workers in the future.

We intend to build on the work performed here by investigating the use of Postgres in other areas, such as complex objects (using ADTs i.e. Abstract Data Types), applications with generalization– specialization networks (inheritance facilities), performance aspects (fast path function) and active databases (triggers and alerts).

7 References

ANSI 1986, ANSI, The Database Lamguage SQL, Document ANSI X3.135.

Atkinson, Chisholm & Cockshott 1981, Atkinson, M P, Chisholm, K J, & Cockshott, W P, (July 1981), PS-Algol: an Algol with a persistent heap, ACM SIGPLAN Notices **17**(7).

Chen 1976, Chen, P P-S, (1976), The Entity-Relationship Model - towards a unified view of data, ACM TODS **1**(1) 9-36.

Codd 1970, Codd, E F, (1970), A Relational Model of Data for Large Shared Data Banks, CACM **13**(6) 377-387.

Codd 1979, Codd, E F, (1979), Extending the Database Relational Model to capture more meaning, ACM TODS **4** 397-434.

Date 1981, Date, C. Referential integrity. In Proceedings of the Seventh International VLDB Conference Cannes, France, Sept. 1981.

Davidson 1993, Davidson, A R, The Implementation of a Database System for the Administration of the Level I Computing Science Courses at the University of Newcastle upon Tyne, M.Sc. Dissertation, Computing Science, Newcastle University D574.

Heather & Rossiter 1987, Heather, M A, & Rossiter, B N, (1987), The Textual Environment and Database Management Systems, in: Empirical Foundations of Information and Software Science III, edd. Rasmussen, J, & Zunde, P, Plenum, New York 45-62.

Ingres 1994, Ingres Reference Manual, Relational Technology Inc.

ISO–ANSI 1994, ISO–ANSI working Draft, Database Language SQL/Foundation (SQL3), March.

Kemnitz 1993, Kemnitz, G., Ed. The POSTGRES Reference Manual, Version 4.1. Electronics Research Laboratory, University of California, Berkeley, Calif. Rep. M93/08, Aug. 1993.

Kim 1994, Kim W. On Object-Oriented Database Technology. ADB, Inc.

Lamb et al 1991, Lamb C, et. al. The ObjectStore Database System. Communications of the ACM, Vol. 34, No. 10, October 1991.

Lorie et al 1985, Lorie, R, Kim, W, McNabb, D, Plouffe, W, & Meier, A, (1985), Supporting Complex Objects in a Relational System for Engineering Databases, in: Query Processing in Database Systems, Kim, W, Reiner, D S, & Batory, D S, (edd.)

145-155, Springer-Verlag, New York.

Matisse 1994, Matisse Sales Literature.

Nelson, Rossiter & Heather 1994, Nelson D A, Rossiter B N, & Heather M A. The Functorial Data Model: An Extension to Functional Databases. Computing Science, Technical Report no. 488, University of Newcastle upon Tyne.

Olson 1994, Olson M A. A Comparison of University Postgres and Montage. Montage Software, Inc.

Rhein et al 1990a, Rhein J, et. al. The Postgres User Manual. University of California, Berkeley.

Rhein et al 1990b, Rhein J, et. al. The Postgres Reference Manual. University of California, Berkeley.

Roth, Korth & Silberschatz 1988, Roth M A, Korth H F, & Silberschatz A, Extended Algebra and Calculus for Nested Relational Databases, ACM TODS **13**(4).

Rowe & Stonebraker 1988, Rowe, L A & Stonebraker, M R. The Postgres Data Model. VLDB, 1988.

Shipman 1981, Shipman, D W. The Functional Data Model and the Data Language DAPLEX. ACM TODS, Vol. 6, No. 1, 1981.

Stonebraker & Rowe 1986, Stonebraker, M, & Rowe L A. The Design of Postgres. Proceedings of 1986 ACM SIGMOD Conference, 1986.

Stonebraker 1986, Stonebraker, M, (1986), Document Processing in a Relational Data base System, The INGRES Papers, Addison-Wesley 357-375.

Stonebraker, Anton & Hanson 1987, Stonebraker, M, Anton, J, & Hanson, E, (1987), Extending a Database System with Procedures, ACM TODS **12**(3) 350-376.

Stonebraker 1994, Stonebraker, M. Object-Relational Database Systems. Montage Software, Inc.

Ullman 1988, Database and Knowledge-base Systems **I**, Computer Science Press, Stanford.

List of Appendices

Appendix A: cfunc.c program list

```
/*
 * cfunc.c - cfunc_project(), cfunc_xmas_1(), cfunc_xmas_2()
 */
#define PORTNAME sparc
#define PORTNAME_sparc

#include <utils/builtins.h>
#include <tmp/libpq-fe.h>

/* calculate value of MARK attribute in PROJECT_MARKS */
int2 cfunc_project(t)
TUPLE   t;
{
        int2    user_man, maintenance_man;
        int2    test_data, further_thought;
        int2    spelling, control_abs;
        int2    data_abs, style;
        int2    layout, functionality;
        int2    correctness, error_handling;
        int2    user_interface, any_penalty;
        int2    problem_stat;
        int2    mark = 0;
        bool    snull;

        /* get value by attribute name from the record */
        problem_stat = (int2)GetAttributeByName(t, "problem_stat",
&snull);
        user_man = (int2)GetAttributeByName(t, "user_man", &snull);
        maintenance_man = (int2)GetAttributeByName(t, "maintenance_man",
&snull);
        test_data = (int2)GetAttributeByName(t, "test_data", &snull);
        further_thought = (int2)GetAttributeByName(t, "further_thought",
```

```
&snull);
        spelling = (int2)GetAttributeByName(t, "spelling", &snull);
        control_abs = (int2)GetAttributeByName(t, "control_abs", &snull);
        data_abs = (int2)GetAttributeByName(t, "data_abs", &snull);
        style = (int2)GetAttributeByName(t, "style", &snull);
        layout = (int2)GetAttributeByName(t, "layout", &snull);
        functionality = (int2)GetAttributeByName(t, "functionality",
&snull);
        correctness = (int2)GetAttributeByName(t, "correctness", &snull);
        error_handling = (int2)GetAttributeByName(t, "error_handling",
&snull);
        user_interface = (int2)GetAttributeByName(t, "user_interface",
&snull);
        any_penalty = (int2)GetAttributeByName(t, "any_penalty", &snull);

        /* calculate value of mark attribute */
        mark = user_man + maintenance_man
                + test_data + further_thought
                + spelling + control_abs
                + data_abs + style
                + layout + functionality
                + correctness + error_handling
                + user_interface + problem_stat
                - any_penalty;

        return(mark);
}


/* calculate value of NO_INCORRECT attribute in XMAS_TEST_MARK */
int2 cfunc_xmas_1(t)
TUPLE   t;
{
        int2    question, correct;
        int2    incorrect = 0;
        bool    snull;

        /* get value by attribute name from the record */
        question = (int2)GetAttributeByName(t, "no_question", &snull);
        correct = (int2)GetAttributeByName(t, "no_correct", &snull);
        incorrect = question - correct;
        return(incorrect);
}
```

```
/* calculate value of MARK attribute in XMAS_TEST_MARK */
int2 cfunc_xmas_2(t)
TUPLE   t;
{
        int2    no_correct;
        int2    no_incorrect;
        int2    total_mark = 0;
        bool    snull;

        /* get value by attribute name from the record */
        no_correct = (int2)GetAttributeByName(t, "no_correct", &snull);
        no_incorrect = (int2)GetAttributeByName(t, "no_incorrect",
&snull);
        total_mark = no_correct - no_incorrect;
        return(total_mark);
}
```

Appendix B: load_course.c

```
/*
 * load_course.c - insert data into course class
 */
#define PORTNAME sparc
#define PORTNAME_sparc

#include <tmp/libpq.h>
#include <tmp/libpq-fe.h>

main(argc, argv)
int argc;
char **argv;
{
   if (argc != 2) {
      fprintf(stderr, "Usage: %s database_name\n", argv[0]);
      exit(1);
   }

   PQsetdb(argv[1]);
   PQexec("copy course from stdin");
   PQputline("cs1       48      4\n");
   PQputline("math1     36      3\n");
   PQputline("ee2       24      2\n");
   PQputline(".\n");
   PQendcopy();
   PQfinish();
   exit(0);
}
```

Appendix C: test program list

```
appendix C consists of
       test.sh : shell script
       CreateClass.q : query for creating classes
       DefIntegrity.q : queries for defining integrity rules
       DefFunRule.q : queries for defining functions and rules
       CopyData.q : queries for loading data into database
       query2 : queries for data manipulation
       query3 : queries using C functions


*******************************************************
                   test.sh
*******************************************************


#!/bin/sh
#

echo =============== destroying old database... =====================
destroydb admindb

echo =============== create admindb database... =====================
createdb admindb
if [ $? -ne 0 ]; then
     echo createdb failed
     exit 1
fi

echo =============== create Classes ... =============================
monitor admindb < CreateClass.q 2>&1 | tee CreateClass.result
if [ $? -ne 0 ]; then
     echo the CreateClass script has an error
     exit 1
fi

echo =============== Define Integrity Rules ... =====================
monitor admindb < DefIntegrity.q 2>&1 | tee DefIntegrity.result
if [ $? -ne 0 ]; then
     echo the DefIntegrity script has an error
     exit 1
fi

echo =============== Define Functions and Rules ... =================
```

```
monitor admindb < DefFunRule.q 2>&1 | tee DefFunRule.result
if [ $? -ne 0 ]; then
     echo the DefFunRule script has an error
     exit 1
fi

echo =============== load data into classes ... =====================
monitor admindb < CopyData.q 2>&1 | tee CopyData.result
if [ $? -ne 0 ]; then
     echo the CopyData script has an error
     exit 1
fi


echo =============== run queries ... ================================
monitor admindb < query2 2>&1 | tee query2.result
if [ $? -ne 0 ]; then
     echo the query2 script causes an error
     exit 1
fi

echo =============== test C Functions ... ===========================
monitor admindb < query3 2>&1 | tee query3.result
if [ $? -ne 0 ]; then
     echo the query3 script causes an error
     exit 1
fi


exit 0


*******************************************************
                   CreateClass.q
*******************************************************


/*
 * CreateClass.q : CLASS DEFINITIONS
 */
create student(registration_id = text,
               student_name = text,
               course_name = text,
               tutor_name = text,
               email = text)
\g
```

```
create course(name = text,
              num_weeks = float4,
              num_projects = float4)
\g
create tutor(name = text,
             room_no = text,
             email = text,
             dept = text)
\g
create attendance(week_no = int2,
                  student_id = text,
                  session = char[4])
\g
create project_marks(project_name = text,
                     marker_name = text,
                     student_id = text,
                     problem_stat = float4,
                     user_man = float4,
                     maintenance_man = float4,
                     test_data = float4,
                     further_thought = float4,
                     spelling = float4,
                     control_abs = float4,
                     data_abs = float4,
                     style = float4,
                     layout = float4,
                     functionality = float4,
                     correctness = float4,
                     error_handling = float4,
                     user_interface = float4,
                     any_penalty = float4,
                     mark = float4)
\g
create xmas_test_mark(student_id = text,
                      no_question = int2,
                      no_attempted = int2,
                      no_correct = int2,
                      no_incorrect = int2,
                      mark = int2)
\g
create perform (sid = text, weeks = float4, projects = float4, all_attends =
float4, sum_projects = float4)
\g
```

```
*********************************************************************
              DefIntegrity.q
**********************************************************************


/*
 *  DefIntegrity : DEFINE INTEGRITY RULES
 */
define rule student_unique is
       on append to student
       where student.registration_id = new.registration_id
          or new.registration_id ISNULL
       do instead nothing
\g
define rule student_int is
        on append to student
        where course.name !~ new.course_name
        or tutor.name !~ new.tutor_name
        do instead nothing
\g
define rule project_int is
       on append to project_marks
       where student.registration_id !~ new.student_id
       do instead nothing
\g
define rule xmas_integrity is
       on append to xmas_test_mark
       where student.registration_id !~ new.student_id
       do instead nothing
\g
define rule attend_integrity is
       on append to attendance
       where student.registration_id !~ new.student_id
       do instead nothing
\g



**********************************************************************
              DefFunRule.q
**********************************************************************


/*
 * DefFunRule.q : defines aggregate, functions and rules
 */
```

```
define aggregate float4count(sfunc2 = float4inc, initcond2 = "0")
\g
/*
 * FUNCTION DEFINITIONS
 */
define function course_info(language="postquel", returntype=course)
       arg is (student)
       as "retrieve (course.all)
           where $1.course_name = course.name"
\g
define function tutor_info(language="postquel", returntype=tutor)
       arg is (student)
       as "retrieve (tutor.all)
           where $1.tutor_name = tutor.name"
\g
define function all_attendance(language = "postquel", returntype = float4)
       arg is (student)
       as "retrieve (attended = float4count{attendance.student_id
           where $1.registration_id = attendance.student_id})"
\g
define function sum_project_mark(language = "postquel", returntype = float4)
       arg is (student)
      as "retrieve (sum_mark = float4sum{project_marks.mark
         where $1.registration_id = project_marks.student_id})"
\g
define function attend_ratio(language = "postquel", returntype = float4)
       arg is (student)
       as "retrieve (ratio = (perform.all_attends / perform.weeks) *
100::float4)
           where $1.registration_id = perform.sid"
\g
define function ave_project_mark(language = "postquel", returntype = float4)
       arg is (student)
       as "retrieve (mark = (perform.sum_projects / perform.projects))
           where $1.registration_id = perform.sid"
\g
define function xmas_mark(language = "postquel", returntype = int2)
       arg is (student)
       as "retrieve (mark = xmas_test_mark.mark)
           where $1.registration_id = xmas_test_mark.student_id"
\g
/*
 * RULE DEFINITIONS
 */
define rule project_rule_1 is
       on retrieve to project_marks
```

```
       do replace current(mark = current.problem_stat + current.user_man
                             + current.maintenance_man  + current.test_data
                             + current.further_thought + current.spelling
                             + current.control_abs + current.data_abs
                             + current.style + current.layout
                             + current.functionality + current.correctness
                             + current.error_handling + current.user_interface
                             - current.any_penalty)
\g
define rule xmas_rule_1 is
       on retrieve to xmas_test_mark
       do replace current(no_incorrect = current.no_question
                                         - current.no_correct)
\g
define rule xmas_rule_2 is
       on retrieve to xmas_test_mark
       do replace current(mark = current.no_correct
                                 - current.no_incorrect)
\g



***********************************************************************
               CopyData.q
***********************************************************************


/*
 * CopyData.q : CLASS POPULATION
 */
copy student from "/home/peepy/n302544/job/data/student.data"
\g
copy attendance from "/home/peepy/n302544/job/data/attend.data"
\g
copy tutor from "/home/peepy/n302544/job/data/tutor.data"
\g
copy course from "/home/peepy/n302544/job/data/course.data"
\g
copy project_marks from "/home/peepy/n302544/job/data/project.data"
\g
copy xmas_test_mark from "/home/peepy/n302544/job/data/xmas.data"
\g



***********************************************************************
```

```
               query2
***********************************************************************

/*
 * query2 : data manipulation
 */
retrieve (student.all)
\g
retrieve (course.all)
\g
retrieve (tutor.all)
\g
retrieve (attendance.all)
\g
retrieve (p.student_id, p.project_name, p.mark) from p in project_marks
\g
retrieve (xmas_test_mark.all)
\g
retrieve into tmp_perform (sid = s.registration_id,
                       weeks = s.course_info.num_weeks,
                       projects = s.course_info.num_projects,
                       all_attends = s.all_attendance,
                       sum_projects = s.sum_project_mark)
from s in student
\g
append perform (tmp_perform.all)
\g
retrieve (perform.all)
\g
/*
 * retrieve course information
 */
retrieve (Student=s.student_name, ID=s.registration_id,
          Course=s.course_name, Weeks=s.course_info.num_weeks,
          Projects=s.course_info.num_projects)
from s in student
\g
/*
 * retrieve tutor information
 */
retrieve (Student=s.student_name, ID=s.registration_id,
          Tutor=s.tutor_name, Tutor_Dept=s.tutor_info.dept,
          Tutor_email=s.tutor_info.email)
from s in student
\g
/*
```

```
 * retrieve overall performance for each student
 */
retrieve (sid = s.registration_id, sname = s.student_name,
          degree = s.course_name, tutor = s.tutor_name,
          attend_ratio = s.attend_ratio, project_mark = s.ave_project_mark,
xmas_mark = s.xmas_mark)
from s in student
\g
/*
 * retrieve students whose attendance is below acceptable standard
 */
retrieve (sname = s.student_name, sid = s.registration_id,
          tutor = s.tutor_name, tutor_email = s.tutor_info.email,
          attendance = s.all_attendance, ratio = s.attend_ratio)
from s in student where s.attend_ratio < 60::float4
\g
/*
 * retrieve students whose project_mark is below acceptable standard
 */
retrieve (sname = s.student_name, sid = s.registration_id,
          tutor = s.tutor_name, tutor_email = s.tutor_info.email,
          project_mark = s.ave_project_mark)
from s in student where s.ave_project_mark < 40::float4
\g
/*
 * retrieve students whose xmas_test_mark is below acceptable standard
 */
retrieve (sname = s.student_name, sid = s.registration_id,
          tutor = s.tutor_name, tutor_email = s.tutor_info.email,
          mark = s.xmas_mark)
from s in student where s.xmas_mark < 20
\g



**********************************************************
                 query3
*******************************************************



/*
 * query3 - DEFINE C-FUNCTIONS
 */
define function cfunc_project ( language="c",
                               returntype = int2 )
```

```
        arg is (project_marks)
        as "/home/peepy/n302544/job/work/cfunc.o"
\g
define function cfunc_xmas_1 ( language="c",
                              returntype = int2 )
        arg is (xmas_test_mark)
        as "/home/peepy/n302544/job/work/cfunc.o"
\g
define function cfunc_xmas_2 ( language="c",
                              returntype = int2 )
        arg is (xmas_test_mark)
        as "/home/peepy/n302544/job/work/cfunc.o"
\g
/*
 * FUNCTION DYNAMIC LOADING
 */
load "/home/peepy/n302544/job/work/cfunc.o"
\g
/*
 * DELETE OLD DATA
 */
delete project_marks
\g
delete xmas_test_mark
\g
/*
 * COPY DATA FROM FILE
 */
copy project_marks from "/home/peepy/n302544/job/data/project.data"
\g
copy xmas_test_mark from "/home/peepy/n302544/job/data/xmas.data"
\g
/*
 * CALCULATE VALUE BY C-FUNCTIONS
 */
replace project_marks( mark = cfunc_project(project_marks) )
\g
retrieve (p.project_name, p.student_id, p.mark) from p in project_marks
\g
replace xmas_test_mark( no_incorrect = cfunc_xmas_1(xmas_test_mark))
\g
replace xmas_test_mark( mark = cfunc_xmas_2(xmas_test_mark) )
\g
retrieve (xmas_test_mark.all)
\g
```

Appendix D : libpq program list

```
Subject: appendix D : libpq program list

Appendix D : Makefile
             classcreate.c
             definerule.c


*****************************************************************
             Makefile
*****************************************************************


POSTGRES=/home/peepy/postgres

LIB1=${POSTGRES}/include
LIB2=${POSTGRES}/src/back--end
LIB3=${LIB2}/port/sparc
LIB4=${LIB2}/obj
LIB5=${POSTGRES}/lib


cfunc.o: cfunc.c
        cc -I${LIB1} -I${LIB2} -I${LIB3} -I${LIB4} -c cfunc.c
        chmod 755 cfunc.o

classcreate: classcreate.c
        cc -I${LIB1} -I${LIB2} -I${LIB3} -I${LIB4} -o classcreate \
           classcreate.c -L${LIB5} -lpq

definerule: definerule.c
        cc -I${LIB1} -I${LIB2} -I${LIB3} -I${LIB4} -o definerule \
            definerule.c -L${LIB5} -lpq

load_course: load_course.c
        cc -I${LIB1} -I${LIB2} -I${LIB3} -I${LIB4} -o load_course \
           load_course.c -L${LIB5} -lpq

perform: perform.c
        cc -I${LIB1} -I${LIB2} -I${LIB3} -I${LIB4} -o perform \
          perform.c -L${LIB5} -lpq
```

```
*****************************************************************
             classcreate.c
*****************************************************************


/*
 * classcreate.c - creates classes
 */
#define PORTNAME sparc
#define PORTNAME_sparc

#include <tmp/libpq.h>
#include <tmp/libpq-fe.h>


main(argc, argv)
int  argc;
char **argv;
{
    char *res;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s database_name\n", argv[0]);
        exit(1);
    }

    PQsetdb(argv[1]);

    (void) PQexec("begin");
    res = (char *) PQexec("create student (registration_id = text, \
                      student_name = text, course_name = text, \
                      tutor_name = text,email = text)");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        PQexec("end");
        PQfinish();
        exit(1);
    }
    if (*res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        PQexec("end");
        PQfinish();
        exit(1);
    }

    res = (char *) PQexec("create course (name = text, \
```

```
                      num_weeks = float4, num_projects = float4)");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    PQexec("end");
    PQfinish();
    exit(1);
}
if (*res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    PQexec("end");
    PQfinish();
    exit(1);
}

res = (char *) PQexec("create tutor(name = text, \
                      room_no = text, email = text, dept = text)");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    PQexec("end");
    PQfinish();
    exit(1);
}
if (*res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    PQexec("end");
    PQfinish();
    exit(1);
}

res = (char *) PQexec("create attendance(week_no = int2, \
                      student_id = text, session = char[4])");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    PQexec("end");
    PQfinish();
    exit(1);
}
if (*res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    PQexec("end");
    PQfinish();
    exit(1);
}

res = (char *) PQexec("create project_marks(project_name = text,\
                      marker_name = text, student_id = text, \
```

```
                          problem_stat = float4, user_man = float4, \
                          maintenance_man = float4, test_data = float4, \
                          further_thought = float4, spelling = float4, \
                          control_abs = float4, data_abs = float4, style =
float4, \
                          layout = float4, functionality = float4,\
                          correctness = float4, error_handling = float4,\
                          user_interface = float4, any_penalty = float4,\
                          mark = float4)");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        PQexec("end");
        PQfinish();
        exit(1);
    }
    if (*res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        PQexec("end");
        PQfinish();
        exit(1);
    }

    res = (char *) PQexec("create xmas_test_mark(student_id = text,\
                          no_question = int2, no_attempted = int2,\
                          no_correct = int2, no_incorrect = int2, mark =
int2)");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        PQexec("end");
        PQfinish();
        exit(1);
    }
    if (*res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        PQexec("end");
        PQfinish();
        exit(1);
    }

    res = (char *) PQexec("create perform(sid = text, weeks = float4,\
                           projects = float4, all_attends = float4,\
                           sum_projects = float4)");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        PQexec("end");
        PQfinish();
```

```
        exit(1);
    }
    if (*res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        PQexec("end");
        PQfinish();
        exit(1);
    }

    PQexec("end");
    PQfinish();
    exit(0);

}

*****************************************************************
             definerule.c
*****************************************************************

/*
 * definerule.c - defines aggregations, functions, and rules
 */

#define PORTNAME sparc
#define PORTNAME_sparc

#include <tmp/libpq.h>
#include <tmp/libpq-fe.h>

void
exit_error()
{
    (void) PQexec("end");
    PQfinish();
    exit(1);
}

main(argc, argv)
int argc;
char **argv;
{
    char *res;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s database_name\n", argv[0]);
        exit(1);
```

```
}

PQsetdb(argv[1]);
(void) PQexec("begin");

/*
 * AGGREGATION DEFINITIONS
 */
res = (char *) PQexec("define aggregate float4count(sfunc2 = float4inc,\
                       initcond2 = \"0\")");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    exit_error();
}


/*
 * FUNCTION DEFINITIONS
 */
res = (char *) PQexec("define function course_info(language=\"postquel\",\
                       returntype=course) arg is (student)\
                       as \"retrieve (course.all)\
                       where $1.course_name = course.name\"");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    exit_error();
}

res = (char *) PQexec("define function tutor_info(language=\"postquel\",\
                       returntype=tutor) arg is (student)\
                       as \"retrieve (tutor.all)\
                       where $1.tutor_name = tutor.name\"");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
```

```
        exit_error();
    }

    res = (char *) PQexec("define function all_attendance\
                            (language = \"postquel\", returntype = float4)\
                            arg is (student)\
                            as \"retrieve (attended =
float4count{attendance.student_id \
                            where $1.registration_id =
attendance.student_id})\"");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }

    res = (char *) PQexec("define function sum_project_mark\
                             (language = \"postquel\", returntype = float4)\
                            arg is (student)\
                            as \"retrieve (sum_mark =
float4sum{project_marks.mark\
                            where $1.registration_id = project_marks.student_id}
)\"");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }


    res = (char *) PQexec("define function attend_ratio\
                             (language = \"postquel\", returntype = float4)\
                            arg is (student)\
                            as \"retrieve (ratio = (perform.all_attends \/
perform.weeks)\
                            * 100::float4) where $1.registration_id =
perform.sid\"");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
```

```
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }


    res = (char *) PQexec("define function ave_project_mark\
                            (language = \"postquel\", returntype = float4)\
                           arg is (student)\
                           as \"retrieve (mark = (perform.sum_projects \/
perform.projects))\
                           where $1.registration_id = perform.sid\" ");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }


    res = (char *) PQexec("define function xmas_mark\
                            (language = \"postquel\", returntype = float4)\
                           arg is (student)\
                           as \"retrieve (mark = xmas_test_mark.mark)\
                           where $1.registration_id = xmas_test_mark.student_
id\" ");
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }


    /*
     * RULE DEFINITIONS
     */
    res = (char *) PQexec("define rule project_rule_1 is\
                           on retrieve to project_marks\
                           do replace current(mark = current.problem_stat\
                           + current.user_man\
```

```
                        + current.maintenance_man  + current.test_data\
                        + current.further_thought + current.spelling\
                        + current.control_abs + current.data_abs\
                        + current.style + current.layout\
                        + current.functionality + current.correctness\
                        + current.error_handling + current.user_interface\
                        - current.any_penalty)");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    exit_error();
}

res = (char *) PQexec("define rule xmas_rule_1 is\
                        on retrieve to xmas_test_mark\
                        do replace current(no_incorrect\
                        = current.no_question - current.no_correct)");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    exit_error();
}

res = (char *) PQexec("define rule xmas_rule_2 is\
                        on retrieve to xmas_test_mark\
                        do replace current(mark = current.no_correct \
                        - current.no_incorrect)");
if (!res) {
    fprintf(stderr, "\nError: PQexec returned NULL\n");
    exit_error();
}
if (res == 'E') {
    fprintf(stderr, "\nError: \"%s\"\n", ++res);
    exit_error();
}

res = (char *) PQexec("define rule perform_rule_1 is\
                        on retrieve to perform\
                        do instead retrieve (student.registration_id, \
                        student.all_attendance, student.all_project_mark)");
```

```
    if (!res) {
        fprintf(stderr, "\nError: PQexec returned NULL\n");
        exit_error();
    }
    if (res == 'E') {
        fprintf(stderr, "\nError: \"%s\"\n", ++res);
        exit_error();
    }

    (void) PQexec("end");
    PQfinish();
    exit(0);
}
```