The POSTGRES Data Model†

Lawrence A. Rowe
Michael R. Stonebraker

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

Abstract

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

1. Introduction

This paper describes the data model for POSTGRES, a next-generation extensible database management system being developed at the University of California [StR86]. The data model is based on the idea of extending the relational model developed by Codd [Cod70] with general mechanisms that can be used to simulate a variety of semantic data modeling constructs. The mechanisms include: 1) abstract data types (ADT's), 2) data of type procedure, and 3) rules. These mechanisms can be used to support complex objects or to implement a shared object hierarchy for an object-oriented programming language [Row86]. Most of these ideas have appeared elsewhere [Ste84, Sto85, Sto86a, Sto86b].

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

The major contribution of this paper is to show that inheritance can be added to a relational data model with only a modest number of changes to the model and the implementation of the system. The conclusion that we draw from this result is that the major concepts provided in an

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

object-oriented data model (e.g., structured attribute types, inheritance, union type attributes, and support for shared subobjects) can be cleanly and efficiently supported in an extensible relational database management system. The features used to support these mechanisms are abstract data types and attributes of type procedure.

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

2. Data Model

A database is composed of a collection of *relations* that contain tuples which represent real-world entities (e.g., documents and people) or relationships (e.g., authorship). A relation has attributes of fixed types that represent properties of the entities and relationships (e.g., the title of a document) and a primary key. Attribute types can be atomic (e.g., integer, floating point, or boolean) or structured (e.g., array or procedure). The primary key is a sequence of attributes of the relation, when taken together, uniquely identify each tuple.

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

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

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

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

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

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

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

Data inheritance is specified with an **inherits**-clause. Suppose, for example, that people in the university database are employees and/or students and that different attributes are to be defined for each category. The relation for each category includes the *PERSON* attributes and the attributes that are specific to the category. These relations can be defined by replicating the *PERSON* attributes in each relation definition or by inheriting them for the definition of

PERSON. Figure 1 shows the relations and an inheritance hierarchy that could be used to share the definition of the attributes. The commands that define the relations other than the *PERSON* relation defined above are:

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

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

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

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

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

Figure 1: Relation hierarchy.

lowed.[1]

The POSTGRES query language is a generalized version of QUEL [HSW75], called *POSTQUEL*. QUEL was extended in several directions. First, POSTQUEL has a **from**-clause to define tuple-variables rather than a **range** command. Second, arbitrary relation-valued expressions may appear any place that a relation name could appear in QUEL. Third, transitive closure and **execute** commands have been added to the language [Kue84]. And lastly, POSTGRES maintains historical data so POSTQUEL allows queries to be run on past database states or on any data that was in the database at any time. These extensions are described in the remainder of this section.

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

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

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

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

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

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

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

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

The ''*'' operator specifies the relation formed by taking the union of the named relation (i.e., *STUDENT*) and all relations that inherit attributes from it (i.e., *STUDEMP*). If the ''*'' operator

---

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

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

Relation expressions may include other set operators: union ($\cup$), intersection ($\cap$), and difference (–). For example, the following query retrieves the names of people who are students or employees but not student employees:

```
retrieve (S.name)
from S in (STUDENT ∪ EMPLOYEE)
```

Suppose a tuple does not have an attribute referenced elsewhere in the query. If the reference is in the target-list, the return tuple will not contain the attribute.[2] If the reference is in the qualification, the clause containing the qualification is ‘‘false’’.

POSTQUEL also provides set comparison operators and a relation-constructor that can be used to specify some difficult queries more easily than in a conventional query language. For example, suppose that students could have several majors. The natural representation for this data is to define a separate relation:

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

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

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

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

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

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

which specifies the same relation as the query

---

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

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

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

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

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

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

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

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

Deferred update semantics are used for all updates commands.

POSTQUEL supports the transitive closure commands developed in QUEL* [Kue84]. A ‘‘*’’ command continues to execute until no tuples are retrieved (e.g., **retrieve***) or updated (e.g., **append***, **delete***, or **replace***). For example, the following query creates a relation that contains all employees who work for Smith:

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

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

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

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

The date specified in the brackets following the relation name specifies the relation at the designated time. The date can be specified in many different formats and optionally may include a time of day. The query above only examines students who are not student employees. To search the set of all students, the **from**-clause would be

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

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

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

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

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

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

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

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

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

```
create version MYPEOPLE from PERSON
```

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

If the user does not want updates to the base relation to propagate to the version, he can create a version of a snapshot. A snapshot is a copy of the current contents of a relation [AdL80]. A version of a snapshot is created by the following command:

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

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

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

```
merge YOURPEOPLE into PERSON
```

that will merge the changes made to *YOURPEOPLE* back into *PERSON*. The **merge** command uses a semi-automatic procedure to resolve updates to the underlying relation and the version that conflict [Gae84].

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

3. Data Types

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

All atomic data types are defined to the system as ADT's. An ADT is defined by specifying the type name, the length of the internal representation in bytes, procedures for converting from an external to internal representation for a value and from an internal to external representation, and a default value. The command

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

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

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

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

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

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

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

Figure 2: Predefined operators precedence.

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

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

```
‘‘20,50:10,70’’
```

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

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

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

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

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

and the query

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

was executed. The *Sort* property of the *AE* operator specifies the procedure to be used to sort the relation if a merge-sort join strategy was selected to implement the query. It also specifies the procedure to use when building an ordered index (e.g., B-Tree) on an attribute of type *box*. The *Hashes* property indicates that this operator can be used to build a hash index on a *box* attribute. Note that either type of index can be used to optimize the query above. The *Restrict* and *Join* properties specify the procedure that is to be called by the query optimizer to compute the restrict and join selectivities, respectively, of a clause involving the operator. These selectivity properties specify procedures that will return a floating point value between 0.0 and 1.0 that indicate the attribute selectivity given the operator. Lastly, the *Negator* property specifies the procedure that is to be used to compare two values when a query predicate requires the operator to be negated as in

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

The **define operator** command also may specify a procedure that can be used if the query predicate includes an operator that is not commutative. For example, the commutator procedure for ‘‘area less than’’ (*ALT*) is the procedure that implements ‘‘area greater than or equal’’ (*AGE*). More details on the use of these properties is given elsewhere [Sto86b].

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

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

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

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

The second type-constructor allows values of type procedure to be stored in an attribute. Procedure values are represented by a sequence of POSTQUEL commands. The value of an attribute of type procedure is a relation because that is what a **retrieve** command returns. Moreover, the value may include tuples from different relations (i.e., of different types) because a procedure composed of two **retrieve** commands returns the union of both commands. We call a relation with different tuple types a *multirelation*. The POSTGRES programming language interface provides a cursor-like mechanism, called a *portal*, to fetch values from multirelations [StR86]. However, they are not stored by the system (i.e., only relations are stored).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bill's class list can be retrieved by the query

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

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

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

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

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

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

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

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

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

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

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

The relation name can be used for the type name because relations, types, and procedures have separate name spaces. The query in type *DEPARTMENT* will retrieve a specific department tuple given a unique object identifier (*oid*) of the tuple. Each relation has an implicitly defined attribute named *oid* that contains the tuple's unique identifier. The *oid* attribute can be accessed but not updated by user queries. *Oid* values are created and maintained by the POSTGRES storage system [Sto87]. The formal argument to this procedure-type is the type of an object identifier. The parameter is referenced inside the definition by ‘‘*$n*’’ where *n* is the parameter number.

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

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

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

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

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

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

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

can be written as

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

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

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

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

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

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

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

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

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

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

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

can be written

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

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

4. User-Defined Procedures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
GetValue(''Birthdate'')
```

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

The model for procedure inheritance is nearly identical to method inheritance in object-oriented programming languages [StB86]. Procedure inheritance uses the data inheritance hierarchy and similar inheritance rules except that a rule is provided to select a procedure when an inheritance conflict arises. For example, suppose that a *Comp* procedure was defined for *STUDENT* as well as for *EMPLOYEE*. The definition of the second procedure might be:

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

A conflict arises when the query on *STUDEMP* above is executed because the system does not know which *Comp* procedure to call (i.e., the one for *EMPLOYEE* or the one for *STUDENT*).

The procedure called is selected from among the procedures that take a tuple from the relation specified by the actual argument *STUDEMP* or any relation from which attributes in the actual argument are inherited (e.g., *PERSON*, *EMPLOYEE*, and *STUDENT*).

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

... **inherits** (STUDENT, EMPLOYEE)

and its IPL is

(STUDEMP, STUDENT,
EMPLOYEE, PERSON)

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

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

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

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

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

Comp(STUDEMP)

is[5]

---

[4] We are using a rule that is similar to the rule for the new Common Lisp object model [Boe86]. It is actually slightly more complicated than described here in order to eliminate some nasty cases that arise when there are cycles in the inheritance hierarchy.

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

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

This query can be precomputed to speed up procedure selection.

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

5. Other Data Models

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

Semantic and functional data models [Dae85, HaM81, Mye80, Shi81, SmS77, Zan83] do not provide the flexibility provided by the model described here. They cannot easily represent data with uncertain structure (e.g., objects with shared subobjects that have different types).

Modeling ideas oriented toward complex objects [HaL82, LoP83] cannot deal with objects that have a variety of shared subobjects. POSTGRES uses procedures to represent shared subobjects which does not have limitation on the types of subobjects that are shared. Moreover, the nested-dot notation allows convenient access to selected subobjects, a feature not present in these systems.

Several proposals have been made to support data models that contain non-first normal form relations [Bae86, Dae86, Dee86]. The POSTGRES data model can be used to support non-first normal form relations with procedure-types. Consequently, POSTGRES seems to contain a superset of the capabilities of these proposals.

Object-oriented data models [Ane86, CoM84] have modeling constructs to deal with uncertain structure. For example, GemStone supports union types which can be used to represent subobjects that have different types [CoM84]. Sharing of subobjects is represented by storing the subobjects as separate records and connecting them to a parent object with pointer-chains. Precomputed procedure values will, in our opinion, make POSTGRES performance competitive with pointer-chain proposals. The preformance problem with pointer-chains will be most obvious when an object is composed of a large number of subobjects. POSTGRES will avoid this problem because the pointer-chain is represented as a relation and the system can use all of the query processing and storage structure techniques available in the system to represent it. Consequently, POSTGRES uses a different approach that supports the same modeling capabilities and an implementation that may have better performance.

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

6. Summary

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

References

[AdL80] M. E. Adiba and B. G. Lindsay, ‘‘Database Snapshots’’, *Proc. 6th Int. Conf. on Very Large Databases*, Montreal, Canada, Oct. 1980, 86-91.

[Ane86] T. Anderson and et. al., ‘‘PROTEUS: Objectifying the DBMS User Interface’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Bae86] D. Batory and et.al., ‘‘GENESIS: A Reconfigurable Database Management System’’, Tech. Rep. 86-07, Dept. of Comp. Sci., Univ. of Texas at Austin, 1986.

[Boe86] D. B. Bobrow and et.al., ‘‘COMMONLOOPS: Merging Lisp and Object-Oriented Programming’’, *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986, 17-29.

[Cod70] E. F. Codd, ‘‘A Relational Model of Data for Large Shared Data Bases’’, *Comm. of the ACM*, JUNE 1970.

[CoM84] G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, *Proc. 1984 ACM-ACM-SIGMOD Conf. on Management of Data Int. Conf. on the Mgt. of Data*, June 1984.

[Dae86] P. Dadam and et.al., ‘‘A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies’’, *Proc. ACM-ACM-SIGMOD Conf. on Management of Data Conf. on Mgt. of Data*, Washington, DC, May 1986.

[Dae85] U. Dayal and et.al., ‘‘A Knowledge-Oriented Database Management System’’, *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

[Dee86] U. Deppisch and et.al., ‘‘A Storage System for Complex Objects’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Gae84] H. Garcia-Molina and et.al., ‘‘DataPatch: Integrating Inconsistent Copies of a Database after a Partition’’, Tech. Rep. Tech. Rep.# 304, Dept. Elec. Eng. and Comp. Sci., Princeton, NJ, 1984.

[HaM81] M. Hammer and D. McLeod, ‘‘Database Description with SDM’’, *ACM-Trans. Database Systems*, Sep. 1981.

[HaL82] R. Haskins and R. Lorie, ‘‘On Extending the Functions of a Relational Database System’’, *Proc. 1982 ACM-ACM-SIGMOD Conf. on Management of Data Conference on Management of Data*, Orlando, FL, JUNE 1982.

[HSW75] G. Held, M. R. Stonebraker and E. Wong, ‘‘INGRES -- A Relational Data Base System’’, *Proc. AFIPS AFIPS Nat. Computer Conf.*, 1975, 409-416.

[Kue84] R. Kung and et.al., ‘‘Heuristic Search in Database Systems’’, *Proc. 1st International Workshop on Expert Data Bases*, Kiowah, SC, Oct. 1984.

[LoP83] R. Lorie and W. Plouffee, ‘‘Complex Objects and Their Use in Design Transactions’’, *Proc. Engineering Design Applications Stream of ACM-IEEE Data Base Week*, San Jose, CA, May 1983.

[Mye80] J. Myloupoulis and et.al., ‘‘A Language Facility for Designing Database Intensive Applications’’, *ACM-Trans. Database Systems*, JUNE 1980.

[Row86] L. A. Rowe, ‘‘A Shared Object Hierarchy’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Shi81] D. Shipman, ‘‘The Functional Model and the Data Language Daplex’’, *ACM-Trans. Database Systems*, Mar. 1981.

[SmS77] J. Smith and D. Smith, ‘‘Database Abstractions: Aggregation and Generalization’’, *ACM Trans. Database Systems*, JUNE 1977.

[StB86] M. Stefik and D. G. Bobrow, ‘‘Object-Oriented Programming: Themes and Variations’’, *The AI Magazine 6*, 4 (Winter 1986), 40-62.

[Ste84] M. R. Stonebraker and et. al., ‘‘QUEL as a Data Type’’, *Proc. 1984 ACM-ACM-SIGMOD Conf. on Management of Data Conf. on the Mgt. of Data*, May 1984.

[Sto85] M. R. Stonebraker, ‘‘Triggers and Inference in Data Base Systems’’, *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

[StR86] M. R. Stonebraker and L. A. Rowe, ‘‘The Design of POSTGRES’’, *Proc. 1986 ACM-ACM-SIGMOD Conf. on Management of Data Int. Conf. on the Mgt. of Data*, June 1986.

[Sto86a] M. R. Stonebraker, ‘‘Object Management in POSTGRES Using Procedures’’, *Proc. Int. Wkshp on Object-Oriented Database Systems*, Asilomar, CA, Sep. 1986.

[Sto86b] M. R. Stonebraker, ‘‘Inclusion of New Types in Relational Data Base Systems’’, *Proc. Second Int. Conf. on Data Base Eng.*, Los Angeles, CA, Feb. 1986.

[Sto87] M. R. Stonebraker, ‘‘POSTGRES Storage System’’, Submitted for publication, 1987.

[Zan83] C. Zaniola, ‘‘The Database Language GEM’’, *Proc. 1983 ACM-ACM-SIGMOD Conf. on Management of Data Conference on Management of Data*, San Jose, CA.,

May 1983.