A Data Model and Query Language for EXODUS

by
Michael J. Carey
David J. DeWitt
Scott L. Vandenberg

Computer Sciences Technical Report #734
December 1987

A Data Model and Query Language for EXODUS

Michael J. Carey
David J. DeWitt
Scott L. Vandenberg

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

In this paper, we present the design of the EXTRA data model and the EXCESS query language for the EXODUS extensible database system. The EXTRA data model includes support for complex objects with shared subobjects, a novel mix of object- and value-oriented semantics for data, support for persistent objects of any type in the EXTRA type lattice, and user-defined abstract data types (ADTs). The EXCESS query language provides facilities for querying and updating complex object structures, and it can be extended through the addition of ADT functions and operators, procedures and functions for manipulating EXTRA schema types, and generic set functions. EXTRA and EXCESS are intended to serve as a test vehicle for tools developed under the EXODUS extensible database system project.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant IRI-8657323, and by grants from the Microelectronics and Computer Technology Corporation and Digital Equipment Corporation.

1. INTRODUCTION

When we began the EXODUS project [Care86, Care87] over two years ago, we purposely avoided centering the project around a new data model and query language. There were several reasons for this decision. First, we strongly doubted that a single data model and query language would adequately serve the wide variety of applications that seem to need database technology (just as no single programming language is suitable for all uses). We therefore began by carefully reviewing the data structuring and processing requirements of a representative sample of emerging application areas, designing EXODUS (in a more or less bottom-up fashion) to be a toolkit for use as a basis in constructing a wide variety of target database facilities. Also, we felt that, should a new "standard" data model emerge and be embraced by the database community (as the relational model was in the 1970's), the EXODUS toolkit would enable us to rapidly develop a system based on this new data model.

While a number of new data models have been proposed in the past few years, there appears to be no consensus on the horizon. A number of database researchers seem to believe that object-oriented database systems are the future [Fish87, Khos87, Lecl87, Horn87, Andr87, Bane87, Maie87], and several flavors of object-oriented models have been identified [Ditt86]. However, there is little consensus as to what an object-oriented database system should be; such systems today range from object-oriented programming languages with persistence to full database systems based on data models that would have been called "semantic data models" two years ago. In fact, the former kind of object-oriented DBMS almost seems like a step back to the days of navigational data manipulation languages, as it is not obvious how one will support ad-hoc queries (or optimize accesses effectively) for such systems [Bloo87, Ullm87].

Another direction in data model evolution, one which has spawned such efforts as [Codd79, Dada86, Sche86, Schw86, Rowe87], is to extend the relational model in some way. A common goal of these efforts (and of our work as well) is to provide better support for complex objects and new data types than that offered by the relational model, while still retaining such important features as a powerful, user-friendly, data manipulation language. One approach to dealing with complex objects (also known as "structural object-orientation" [Ditt86]) is to provide procedures as a data type [Ston87a]. Another approach is to permit relation-valued attributes [Dada86, Sche86]. A third approach is to take a functional view of data [Ship81, Mano86, Bato87]. In addition, a common theme among many of these efforts is to extend the database system's data definition facilities with support for some form of type inheritance.

Now that a number of the components of EXODUS are nearing completion, we have turned our attention to the process of selecting a target data model to use as a demonstration vehicle for the system. Since no one data model seemed exactly right to us, and we have been unable to obtain details regarding several promising commercial next generation data model efforts, we (somewhat reluctantly) decided that we should design our own data model and query language for demonstrating the capabilities of the EXODUS system. This paper presents the EXTRA data model and the associated EXCESS query language, which are the results of our design effort.[1] Readers familiar with the data modeling literature will recognize the result as a synthesis and extension of ideas from other data models, including GEM [Zani83], POSTGRES [Rowe87], NF^2 models [Dada86, Sche86], DAPLEX [Ship81], ORION [Bane87], Trellis/Owl [Scha86], O_2 [Lecl87], STDM [Cope84], and STDM's descendent, GemStone [Maie86]. Our most important extensions include: support for complex objects based on a novel mixture of object and value semantics; a user-friendly, high-level query language reflecting this mixture; facilities to allow objects of any type to be made persistent and accessed via the query language; and an integration of user-defined types and operations with the query language at two levels, for both abstract data types and for conceptual schema-level types.

The remainder of this paper is organized as follows: Section 2 presents the data modeling facilities of EXTRA, including a number of examples. In Section 3, we present the EXCESS query language, including its facilities for querying complex objects, performing updates, and computing aggregates. Section 4 discusses the support provided in EXTRA and EXCESS for user-defined types and operations. Section 5 compares our proposal with a number of other recently proposed data models and query languages. Finally, Section 6 presents our conclusions and our plans regarding future research and implementation work.

2. THE EXTRA DATA MODEL

In the EXTRA data model, a database is a collection of named persistent objects. These objects can be as simple or as complex as desired; EXTRA does not constrain the type structure of the named (or "top level") objects in the database. EXTRA separates the definition of types from the declaration of their instances, and it provides a type system based on a type lattice with multiple inheritance. The type system includes tuple, set, and array as type

[1] EXTRA stands for EXtensible Types for Relations and Attributes; EXCESS is short for EXtensible Calculus with Entity and Set Support.

constructors that may be composed arbitrarily to form new types. EXTRA also provides support for user-defined abstract data types. We will elaborate on each of these features in this section, and we will illustrate them by iterative refinement of a simple example database.

2.1. The Basic EXTRA Type System

In EXTRA, the definition of a type and the declaration of instances of that type are completely separated from one another. This makes it possible for a database to include more than one collection of instances of a given type, which can be quite useful in scientific and engineering applications [Lohm83, Kemp87]. While this separation is common in programming languages, it is less common in the database world [Bloo87]. As an example, the commands in Figure 1 define a new schema type called Person, which is a tuple type. The ssnum attribute of a Person is specified as being a unique key for Person objects; key constraints are supported but optional in EXTRA. Two sets for storing Person instances are then created, the Students set and the Employees set. (We will explain the **own ref** syntax shortly; it can be ignored for the purposes of the current discussion. For now, the Students and Employees sets can be thought of as relations.)

EXTRA provides a variety of base types and type constructors for defining schema types, some of which are used in Figure 1. Predefined base types include integers of various sizes, single and double-precision floating point numbers, booleans, character strings, and enumerations. EXTRA also supports the addition of new base types through an abstract data type (ADT) facility similar to those of [Ston86, Ston87b] like the Date type in Figure 1; ADT support will be covered in a later section. The type constructors of EXTRA include tuple (e.g., Person is a tuple type), fixed length arrays, variable length arrays, sets, and references. We will have more to say regarding

```
define type Person:
(
        ssnum:      int4,
        name:       char[ ],
        street:     char[20],
        city:       char[10],
        zip:        int4,
        birthday:   Date
) key (ssnum)

create Students:    { own ref Person }
create Employees:   { own ref Person }
```

Figure 1: Creating schema types and instances.

reference, set, and array type constructors later on, when we describe EXTRA's support for complex objects.

The example of Figure 1 is unrealistic in the sense that we will probably want to associate more information with Student and Employee objects than that which is contained in a Person object. Figure 2 refines our example, illustrating type inheritance in EXTRA and a more realistic declaration of the Students and Employees sets. The Student tuple type inherits all of the attributes of the Person tuple type, and in addition it has a grade point average (gpa) attribute and a department (dept) attribute. The Employee tuple type adds the attributes jobtitle, dept, manager, and salary to those of the Person tuple type.

A tuple type may also inherit attributes from more than one type, as the general form of the inheritance clause is **inherits** *type1, type2, ...* If a type inherits two (or more) attributes with the same name, a conflict arises. If these attributes are inherited from a single common supertype, conflict resolution unites them into a single attribute for the new type — since the conflicting attributes have the same source, they also have the same meaning. However, if a name conflict arises between attributes from different sources, no attempt is made to handle the problem automatically. Instead, we require the definer of the type to resolve the conflict explicitly via renaming. For example, suppose we wish to create a new type called "WorkStudyStudent" as a subtype of both the Employee and Student types of Figure 2. This would be disallowed due to a name conflict under our rule since the dept attribute would be inherited from two sources with two different meanings: the Student type's dept attribute specifies a student's major, while the Employee type's dept attribute specifies the department where an employee works. Figure 3 shows how this conflict can be resolved via renaming. POSTGRES would ignore this conflict because the conflicting dept attributes are of the same data type [Rowe87], while TAXIS would simply disallow the conflict [Nixo87]. EXTRA is closest to ORION [Bane87] in its handling of conflicts, except that we provide no automatic resolution. In the

```
define type Student:
(
        gpa:    float4,
        dept:   char[30]
)
inherits Person

define type Employee:
(
        jobtitle:       char[20],
        dept:           char[30],
        manager:        char[ ],
        salary:         int4
)
inherits Person

create Students:        { own ref Student }
create Employees:       { own ref Employee }
```

Figure 2: Creating subtypes and their instances.

absence of explicit renaming, ORION would select one of the two dept definitions automatically, based on the order of entries in the inherits clause, dropping the other one.

```
define type WorkStudyStudent:
(
        weeklyHours:    int4
        majorDept:      Student.dept,
        jobDept:        Employee.dept
)
inherits Student, Employee
```

Figure 3: An example with multiple inheritance.

2.2. Modeling Complex Objects in EXTRA

A complex object is an object that is composed of a number of component objects, each of which may in turn be composed of other component objects. In general, an object can be a component of more than one object. EXTRA provides three type constructors to support complex object modeling, the **ref** type constructor, the **own** type constructor, and the set (or "{ }") type constructor.

In EXTRA, reference attributes are specified as follows:

```
attrname:       ref typename
```

As an example, Figure 4 shows the schema of Figure 2 redefined using reference attributes for the dept attribute of the Student type and for the dept and manager attributes of Employee. These are like reference attributes in GEM [Zani83], with each Student object containing (a reference to) its corresponding Department object. The referenced

```
define type Student
(
        gpa:            float4,
        dept:           ref Department
)
inherits Person

define type Employee
(
        jobtitle:       char[20],
        dept:           ref Department,
        manager:        ref Employee,
        salary:         int4
)
inherits Person

create Students:        { own ref Student }
create Employees:       { own ref Employee }
```

Figure 4: Reference attribute example.

department object is required to exist elsewhere in the database (or else the value of the reference must be null[2]). In addition, the referenced object must be of type Department or else some subtype of the Department type.

Another type constructor that EXTRA provides for modeling complex object types is the set constructor, whose use has already been illustrated for creating the equivalent of a relation in EXTRA. EXTRA allows sets of any data type to be defined/created, and such sets can then contain objects of the specified type and also any of its subtypes. Sets of base types, constructed types, and reference types are all possible in EXTRA. This leads to a very powerful facility for modeling complex objects, as nested relations (ala NF^2 data models) can be supported via sets of tuples, and sets with shared subobjects (ala [Bane87 and Lecl87]) can be supported via sets of references. As an example, Figure 5 shows the definition for the Department type and the creation of a persistent set (Departments) of objects of this type. The definition of the employees attribute of the Department type specifies it as being a set of references to objects of type Employee. Thus, for a given department, the employees attribute effectively contains all of the employees that work for the department.

While reference attributes and their interaction with the set type constructor permit the development of complex object (graph) structures, there are cases where the database designer wishes to treat an object and its components as a single object that simply happens to have a complex value structure. There are also related cases where the components of an object need to be "full-fledged objects," but where the object should still be treated as a whole, as in ORION's composite objects [Kim87]. To support these different cases, EXTRA provides three different kinds of attribute value semantics: **own** attributes, **ref** attributes, and **own ref** attributes. An **own** attribute is simply a value, not a first-class object; it lacks identity in the sense of [Khos86]. By default, all attributes are taken to be

```
define type Department
(
        name:           char[ ],
        floor:          int4,
        numemps:        int4,
        employees:      { ref Employee }
) key (name)

create Departments:     { own ref Department }
```

Figure 5: Combining sets and references.

[2] Space limitations prevent us from fully addressing the integrity implications of EXTRA, but referential integrity and null values will be handled in a manner similar to GEM [Zani83].

own attributes unless otherwise specified. An **own ref** attribute is a reference with the added constraint that the referenced object is *owned* by the referencing object[3]; thus, if the referencing object is deleted, the referenced **own ref** object is deleted as well. Finally, a **ref** attribute is simply a reference to another (independent) object, as described earlier, without cascaded deletion semantics. In addition to their semantic importance, **own** and **own ref** provide EXTRA with information that can potentially be exploited for performance reasons (e.g., for clustering).

The combination of set and array type constructors, together with **ref**, **own ref** and **own** modes for attributes, yields a flexible and powerful facility for modeling complex object structures. The database designer can tailor a design appropriately, rather than adapting it to one particular semantics for modeling complex objects. For example, we could extend the definition of our Employee type with an additional field for keeping track of employees' children:

kids: { **own** Person }

This defines the kids attribute of an Employee as a set of tuples of type Person.[4] Note that these Person instances are tuple *values*, and not tuple *objects*. They do not have *object identity* as objects do, and therefore they cannot be referenced from elsewhere in the database via a reference attribute. In addition, if an employee is deleted, so are his or her kids. This provides a capability very similar to that provided by NF^2 data models [Dada86, Sche86]. If the kids attribute were instead declared to be of type "{ **own ref** Person }", the deletion semantics would be the same, but children could then be referenced from elsewhere in the database (by **ref** attributes of other objects). As with composite objects in ORION [Kim87], however, a Person instance in the kids set of one Employee instance cannot be in the kids set of another Employee instance simultaneously. To overcome this limitation, the kids attribute could be defined simply as "{ **ref** Person }," in which case sharing is permitted and the deletion of an employee instance will not automatically delete the kids. As we will see in the next section, despite their semantic differences, **own**, **ref**, and **own ref** attributes are all treated uniformly in the EXCESS query language for query simplicity. Thus, casual users can ignore the distinction, viewing attributes simply as other objects, as in most object-oriented data models [Lecl87, Horn87, Andr87, Bane87, Maie87].

---

[3] Note: All objects must have a "home" as an **own ref** component of some other object in order to exist in the database. Named, persistent, top-level objects are automatically made **own ref** components of the database in which they are created. This ensures that automatic garbage collection is not needed.

[4] Since **own** is the default, this is equivalent to "kids: { Person }".

In addition to sets and references, fixed and variable-length arrays are provided and are useful for modeling complex objects. Variable-length arrays are "insertable", meaning that array elements can be inserted or deleted anywhere in the array; such an array can be viewed as a sequence or an indexable list. An array type constructor can be used anywhere a set constructor can be applied, which means that (for example) one can have variable-length arrays of objects. This is useful for modeling complex objects where order is important (e.g., documents).

At this point, the reader may be wondering how EXTRA's type constructors (set, array, **ref**, **own**, and **own ref**) fit into the type hierarchy. The answer is simple: They don't. The type hierarchy contains types created using the **define type** command, which are called *schema types*, plus the EXTRA base types (both built-in and ADTs). For tuple types, the supertype/subtype relationship between types is defined explicitly by the user via the **inherits** clause. For an aggregate type such as a set, the rule is that a set type S1 consisting of objects of type T1 is a subtype of a set type S2 consisting of objects of type T2 if T1 is a subtype of T2. Arrays are handled similarly, as are the **own**, **ref**, and **own ref** type constructors.

2.3. Other Attribute Types

In addition to the facilities described thus far, EXTRA provides support for two other kinds of attributes: attributes of various types defined by users (i.e., user-defined ADTs), and attributes defined in terms of other values in the database, or derived attributes.

As an example of an ADT attribute, if we wished to include a picture of each employee in the database, we could extend our definition of the Employee type with a new attribute for storing this information:

face: Picture

Including this in the definition of the Employee object type says that each Employee will have a face attribute of type Picture, which is a user-defined ADT for storing bit-map images. We will describe EXTRA's ADT facilities (and distinguish them from schema types) in more detail in Section 4; basically, though, an ADT can be used wherever any of EXTRA's built-in types can appear.

As for derived attributes, EXTRA allows object attributes to be defined via queries in the EXCESS query language. This facility allows objects to appear to contain certain information which, rather than being stored, can be computed on demand from the current database state. As we will discuss in Section 4, derived attributes in EXTRA are similar to the derived or "procedural" data notions of other data models, and they are supported through

a facility for associating both EXCESS functions and procedures with EXCESS types. For example, we could extend our Person type definition by defining a derived age attribute, which could be referenced just as if it were a real attribute, as follows:

```
define Person function age returns int4
(
        retrieve (Today - this.birthday)
)
```

In this example, *this* is a special range variable that is implicitly bound to the Person instance to which the function is applied, *Today* is a top-level database object of type Date containing today's date, and the minus operator is a Date operator that computes the difference in years between two Dates. We will say more about EXCESS functions and procedures in Section 4, after the EXCESS query language has been described.

```
define type Person
(
  ssnum:      int4,
  name:       char[ ],
  street:     char[20],
  city:       char[10],
  zip:        int4,
  birthday:   Date
) key (ssnum)
```

```
define type Employee
(
  jobtitle:   char[20],
  dept:       ref Department,
  manager:    ref Employee,
  salary:     int4
  kids:       { own Person }
)
inherits Person
```

```
define type Student
(
  gpa:        float4,
  dept:       ref Department
)
inherits Person
```

```
define type Department
(
  name:       char[ ],
  floor:      int4,
  manager:    ref Employee,
  employees:  { ref Employee }
) key (name)
```

```
create Students:      { own ref Student }
create Employees:     { own ref Employee }
create Departments:   { own ref Department }
create StarEmployee:  ref Employee
create TopTen:        array [1..10] of ref Employee
create Calendar:      array [1..12] of array [ ] of Day
create Today:         Date
```

Figure 6: A complete employee database example

2.4. Data Model Summary

To summarize, an EXTRA database is a collection of named persistent objects of any type. EXTRA separates the notions of type and instance; thus, users can collect related objects together in semantically meaningful sets and arrays, which can then be queried, rather than having to settle for queries over type extents as in many data models (e.g., [Ship81, Bane87, Lecl87, Mylo80, Rowe87]). EXTRA provides tuple, set, fixed-length array, and variable-length array as type constructors. In addition, there are three kinds of values, **own**, **ref**, and **own ref** (although casual users such as query writers need not be concerned with this distinction). Combined with the other type constructors, these provide a powerful set of facilities for modeling complex object types and their semantics. Finally, EXTRA provides support for user-defined ADTs and for derived attributes. Figure 6 illustrates some of the sort of database structures that can be defined in EXTRA, showing how one could create Students, Employees, and Departments "relations," a named object for accessing the best employee directly, an array for keeping track of the top ten employees, a calendar object (represented as a nested array of ADT values of type Day), and an object for storing the current date. The next section will explain how such a database can be queried and updated via the EXCESS query language.

3. THE EXCESS QUERY LANGUAGE

While some may question the need for a general-purpose query language in a database system designed to support emerging application areas such as CAD/CAM, we believe that the inclusion of such a language is indeed justified. Functionality like associative searching can be important in any application domain, even CAD/CAM. For example, if reuse of design components is to become a reality, designers will need to be able to query the database of design objects in order to see if an appropriate component already exists. In addition, a full-function query language makes it possible for the same database system to be used for both business and engineering data, supporting queries such as those needed to compute design costs or to order parts for assembling a design object [Ston87c]. Lastly, associative query languages are important because they are amenable to query optimization techniques.

In this section we present the design of the EXCESS query language. While EXCESS is based on QUEL [Ston76], we have borrowed ideas from the QUEL extensions developed for GEM [Zani83] and POSTGRES [Rowe87, Ston87b] as well as work on SQL extensions for handling NF^2 data [Dada86, Sche86]. Salient features of the EXCESS language include: a uniform treatment of all kinds of sets and arrays, including nested sets; a type-

oriented treatment of range variables; a clean, consistent approach to aggregates and aggregate functions; and an update syntax that supports the construction of complex objects with shared subobjects. In the remainder of this section, we will describe each of the major features of the EXCESS query language. Our examples will be based on the employee database in Figure 6.

3.1. Set Query Basics

EXCESS provides a uniform syntax for formulating queries over sets of objects, sets of references, and sets of (own) values. For example, consider the following EXCESS query:

```
range of D is Departments
retrieve (E.name) from E in D.employees where D.floor = 2
```

This query finds the names of all employees who work in departments located on the second floor. The initial range statement specifies that the range variable D is to be bound to the set of department objects in the Departments set; this is the conventional (QUEL) use of a range variable. The phrase "**from** E **in** D.employees" in the query specifies that the range variable E is to be bound to the set of employees for each department that satisfies the selection predicate "D.floor=2". Since D.employees is a set of Employee references, this illustrates how sets of references can be easily manipulated via EXCESS queries.

As another example, the following query finds the names of the children of all employees who work for a department on the second floor:

```
range of E is Employees
retrieve (C.name) from C in E.kids where E.dept.floor = 2
```

Despite the fact that E.kids is a set of values rather than a set of objects, this query looks the same as the previous example because all sets are treated alike in EXCESS queries. (Minor differences do arise for updates, though, as we will describe shortly.)

EXTRA also allows the creation of named persistent objects as single instances of any type (e.g., Today and StarEmployee in Figure 6). Such objects can be referenced directly in the EXCESS query language. For example:

```
retrieve (Today)
retrieve (StarEmployee.name, StarEmployee.salary)
retrieve (TopTen[1].name, TopTen[1].salary)
```

3.2. Range Variables and Their Types

EXCESS provides several different mechanisms for specifying the set of objects over which a variable is to range. Most are similar to the mechanisms of GEM [Zani83] and POSTQUEL [Rowe87], but EXCESS also provides support for universal quantification (to simplify certain kinds of set queries). The simplest form of range statement has the traditional QUEL syntax:

range of <Variable> **is** <Range_Specification>

For this form of range variable, the <Range_Specification> must identify either a named, persistent set (e.g., Employees), array (e.g., TopTen), or subrange of an array (e.g., TopTen[2:5]). Unless restricted by lower and upper bounds, arrays are treated as sets (except for duplicate semantics on updates). Thus, for example,

range of E **is** TopTen

results in E ranging over all objects in the TopTen array while the following range statement

range of E **is** TopTen[2:5]

restricts E to ranging over the 2nd through 5th objects. As far as the user is concerned, it is immaterial whether the set or array contains objects, references, or values.

Like GEM, an implicit range variable is provided for each set or array specified in the target list or in the qualification of a query. Thus, our earlier query involving children of second floor employees could have been written as:

retrieve (C.name) **from** C **in** Employees.kids
where Employees.dept.floor = 2

EXCESS also provides a path syntax in order to simplify the task of formulating queries over nested sets of objects [Ship81, Cope84]. As an example, the semantics of the statement

range of C **is** Employees.kids

are that for each employee object in the Employees set, C will iterate over all the children of the employee. If one of the elements of a path is a single object, it is treated as a singleton set. For example, the statement

range of C **is** Departments.manager.kids

results in C ranging over the manager's children for each department.

While the first form of range statement is used strictly for associating a range variable with a persistent, named set or array, the remaining forms are used for associating a range variable with any set or array. One such form of range statement has the following syntax:

from <Variable> **in** <Set_Specification>[:<Type>]

Here, <Set_Specification> may be any one of the following: a persistent, named set or array (e.g., Employees, Top-Ten, or TopTen[2:5]); an attribute defined using a set or array type constructor (e.g., E.kids or Departments.employees); or a set formed by taking the union (+), intersection (*), or difference (-) of two or more sets (e.g., "TopTen + StarEmployee" or "Students - Employees").

In an EXCESS query, each range variable has an associated type, and the query may only refer to attributes associated with this type. In many cases, the appropriate type can be inferred easily from the query. For example, in our queries involving children of second floor employees, the type of the variable ranging over the Employees set is Employee, the type of the elements in the set. If the set over which a variable ranges is the union, intersection, or difference of two or more sets (e.g., "**from** P **in** Students - Employees"), it is required that the sets share a common supertype. If this supertype is unique, then the type of the range variable is inferred to be the most specific common supertype of the sets' types. If no unique supertype exists, then the normally optional <Type> specification in the range syntax must be used to identify the particular supertype relevant to the query.

The last form of range statement in EXCESS is as follows:

forall <Variable> **in** <Set_Specification>[: <Type>]

This form of range statement is used to specify a universally quantified range variable. Its use is illustrated in the following query, which retrieves the names and ages of all highly paid employees whose children are all under 5 years old:

retrieve (E.name, E.age) **from** E **in** Employees
where E.salary > 100000 **and** (**forall** C **in** E.kids C.age < 5)

It is useful to contrast this query with the following query, which retrieves the names and ages of the highly paid employees with at least one child under 5 years old:

retrieve (E.name, E.age) **from** E **in** Employees, C **in** E.kids
where E.salary > 100000 **and** C.age < 5

The inclusion of universal quantification simplifies the specification of queries where it is desired that all elements of a set or array satisfy some property. Such queries would otherwise have to be specified using an awkward combination of aggregates, and would also likely be more difficult to optimize. Without universal quantification, our "kids are all under 5" example would have to be written something like the following:

retrieve (E.name, E.age) **from** E **in** Employees, C **in** E.kids
where E.salary > 100000 **and** count(C) = count(C **where** C.age < 5)

For those cases in which a range variable ranges over an array or a subrange of an array, it may be useful to retrieve the index of each object that satisfies the query (as well as the object itself). To provide this capability, the <Variable> specification for each form of range statement can be augmented with an index variable for each dimension desired. As an example, the query

retrieve (i, E.name) **from** E[i] **in** TopTen **where** E.city = "Berkeley"

will retrieve the name of each TopTen employee living in Berkeley along with the index position of the employee in the TopTen array.

3.3. Objects as Results

In the preceding section, the result of each example query was a set of tuples for which the type of each attribute was **own**. In general, however, the result of a query can be a set of objects or a set of tuples in which one or more attributes is an object (**ref** or **own ref**). As an example, consider the following query which, for each employee making over $100,000, returns the name of the employee and his or her department:

range of E **is** Employees
retrieve (E.name, E.dept) **where** E.salary > 100000

If this query were embedded in a programming language, then the object identifier for each department could be bound to a host variable for subsequent manipulation. However, in the case of an ad-hoc query (where **retrieve** really means *print*), printing the object identifier of each department is not acceptable. Since there does not seem to be a single "right" answer to the question of what to print, the solution we have adopted is that when a retrieve query returns an object as the result of an ad-hoc query, the system will print each **own** attribute of the object (recursively, if it is a structured attribute). Thus, the above query is equivalent to:

```
range of E is Employees
retrieve (E.name, E.dept.name, E.dept.floor) where E.salary > 100000
```

To facilitate the specification of "deep" retrieval operations, fields in the target list of a retrieve query can be tagged with a "*" operator to indicate that the object plus all of its component[5] objects are to be recursively retrieved. For example,

```
retrieve (E.name, E.kids*)
```

will retrieve the name of the employee and, for each of his/her children, the child's ssnum, name, street, etc. This capability will be especially useful for those applications (e.g., CAD/CAM) that need to extract an object and all of its component objects from the database with a single retrieve statement.

3.4. EXCESS Join Queries

Like GEM, EXCESS provides three types of joins: *functional joins* (or implicit joins, using the dot notation); *explicit identity joins*, where entities are directly compared, and traditional relational *value-based joins*. A number of our examples have involved functional joins. As a further example, consider the following query:

```
retrieve (Employees.dept.name) where Employees.city="Madison"
```

This query selects the employees living in Madison and then finds the name of their departments using the Department reference attribute of the Employee type as an implicit join attribute. The same query can also be expressed using an explicit identity join:

```
retrieve (D.name) from D in Departments, E in Employees
where E.dept is D and E.city = "Madison"
```

In this form, the value of the reference attribute E.dept is directly compared with the range variable D using the **is** operator. The **is** operator is useful for comparing references, returning true if two references refer to the same object. Thus, **is** is a test for object equality rather than (recursive) value equality in the sense of [Banc86]. An **isnot** operator is also provided for convenience in testing that two references do not refer to the same object. As in GEM, these are the only comparison operators applicable to references.

---

[5] By *component object*, we mean those attributes whose type is **own** or **own ref**.

It is expected that most joins in EXCESS will be functional joins, as the important relationships between entities can be expressed directly through reference attributes; most other joins are expected to be explicit identity joins. However, traditional value-based joins are also supported in the EXCESS query language, as in GEM. For example, the following query finds the names of all persons living in the same city as Jones:

retrieve (P1.name) **from** P1,P2 **in** (Students + Employees)
where P1.city = P2.city **and** P2.name = "Jones"

3.5. Aggregates and Aggregate Functions

Since aggregates add important computational power to a query language, we felt that it was important to address them carefully in EXCESS. In deciding how to integrate aggregates into the EXCESS query language, we have tried to provide an intuitive semantics for aggregates and range variable binding. We have also tried to make a clear distinction between the attributes used for partitioning, the attributes being aggregated, and other query attributes. The following syntax is used for expressing EXCESS aggregates and aggregate functions:

agg-op(X [**over** Y] [**by** Z] [**from** <Variable> **in** <Set_Specification>] [**where** Q])

The execution of an aggregate can be viewed logically as follows: First, the qualification Q is applied to each element ranged over by the range variable (i.e., a selection is performed). The resulting set of values is then projected on the attribute(s) specified in Y, with duplicates (if any) being eliminated in the process; if no **over** Y clause is specified, duplicate elimination is not performed.[6] If a **by** Z clause has been specified, the set then is partitioned using the Z attribute(s) as a key. Note that if Z=Y, then each tuple forms its own partition; if no **by** clause is specified, then there is just one partition (in which case the aggregate is being used as a scalar aggregate). Finally, the operation agg-op is applied to the attribute(s) specified by X, yielding one result value for each partition. Both X and Z must be subsets of Y. For agg-op, EXCESS provides all of the usual built-in aggregates (e.g., sum, count, avg, max, and min).

As a first example, the following EXCESS query will find the average salary of all employees:

retrieve (avgsal = avg(E.salary **from** E **in** Employees))

---

[6] Using **over** in this way facilitates certain queries that would otherwise be difficult to express using QUEL unique aggregates or aggregates with SQL-like **unique** clauses, and it also renders such uniqueness clauses unnecessary [Klau85].

In order to fulfill our design goal of a clear, consistent rule for range variable binding, we use Pascal-like scoping rules for range variables.[7] First, a range variable declared in the outer query can be used within an aggregate; however, if the same variable is redeclared within the aggregate, the effect is to define a new range variable. Second, since aggregates can be viewed as self-contained queries, *all* range variables declared inside an aggregate are strictly local to that aggregate expression. These rules considerably improve the semantics of range variables in aggregates over QUEL. A consequence is that the result of an aggregate must be bound explicitly to the remainder of an aggregate query. For example, the following query finds the name of each employee plus the average salary of all employees who work for his or her department:

range of EMP **is** Employees
retrieve (EMP.name, avg(E.salary **by** E.dept **from** E **in** Employees **where** E.dept **is** EMP.dept))

In this example, the variable E ranges over the Employees set within the aggregate function. In order to bind the results of the aggregate function to the outer query, the clause "E.dept **is** EMP.dept" must be included. In QUEL, this binding could be accomplished implicitly by using a single range variable for the entire query.

A final example will illustrate the separation of aggregation and partitioning attributes; it will also show how one can operate on attributes from one level of a complex object while partitioning on attributes from other levels. The following query retrieves the name of each employee; for each employee, it also retrieves the age of the youngest child among the children of all employees working in a department on the same floor as the employee's department:

range of EMP **is** Employees
retrieve (EMP.name, min(E.kids.age **by** E.dept.floor **from** E **in** Employees
where E.dept.floor = EMP.dept.floor))

In this example, the variable E ranges over Employees within the scope of the min aggregate, and E within the aggregate is connected to EMP declared in the range statement via a join on Employee.dept.floor. The query aggregates over Employee.kids, a set-valued attribute, and partitions the data using an attribute of Employee.dept, which is a single-valued reference attribute.

---

[7] Our scoping rules are similar to those of POSTQUEL [Rowe87, Ston87c].

3.6. Updates in EXCESS

EXTRA provides four commands for updating a database: **insert**, **copy**, **replace**, and **delete**. The **insert** command takes an object and adds a reference to it to a collection (set or array). The **copy** command is similar to the **insert** command except that it creates a new object, either by making a copy of an existing object or by using values supplied as parameters to the command. To make a copy of an existing object, the **own** components of the object are copied recursively, new objects are created recursively for all components of type **own ref**, and references are simply copied for components of type **ref**. Thus, the original object and the copy will share component objects referred to by attributes of type **ref**. For both **insert** and **copy**, if the target collection does not already exist, it must first be explicitly created using a **create** command.

The following example illustrates the use of the **copy** command to create a new employee object and add it to Employees set. The effect of this update is to add Smith to the Employees set, and to have him work for Jones in the Computer Science Department.

```
copy to Employees(name = "Smith", street = "1210 W. Dayton St.", city = "Madison",
birthday = "1/1/64", dept = D, manager = M, jobtitle = "programmer", salary = 50000)
from D in Departments, M in Employees
where D.name = "Computer Sciences" and M.name = "Jones"
```

The following insertion query completes the hiring process by adding a reference to Smith to the employees attribute of the Computer Science object in the Departments set:[8]

```
insert into D.employees (E) from D in Departments, E in Employees
where E.name="Smith" and D.name = "Computer Sciences"
```

To add a child of Smith's to the database, the following query would be used:

```
copy to E.kids(name="Anne", street = E.street, city = E.city, zip = E.city,
birthday = "10/10/79") from E in Employees where E.name = "Smith"
```

Since any object can be owned by only one **own ref** referencing object, the following **insert** command will fail:

```
create overpaid (own ref Employee)
range of E is Employees
insert into overpaid (E) where E.salary > 100000
```

[8] It would be nice to also ensure that the employees attribute of the appropriate department object is updated whenever an employee is hired or fired. Using the extensibility features described in Section 4, it is possible to define a *HireEmployee* procedure that encapsulates the appropriate pair of update queries as a single command.

The problem here is that employee objects are already owned by the Employees set object. This would have worked if overpaid had been created as "{**ref** Employee}". The following query will succeed since the **copy** command makes a copy of each employee object:

```
create overpaid {own ref Employee}
range of E is Employees
copy to overpaid (E) where E.salary > 100000
```

Support is also provided for updates to arrays. For example, the following query will put the employee with the highest salary in the first position of the TopTen array (declared in Figure 6).

```
range of E is Employees
insert into TopTen [1] (E) where E.salary >= max (X.salary from X in Employees)
```

An **insert** or **copy** operation on a fixed length array is performed by replacement beginning at the specified index position. If there is more than one employee with the maximum salary, those employees will be inserted starting at TopTen[2]. Updates on variable length arrays are performed by inserting (or beginning) at the specified position (with the keyword **last** being provided to indicate the end of the array).

The **replace** command is used to modify objects. For example,

```
replace E (salary = E.salary * 1.1) from E in Employees where E.name = "Jones"
```

will give Jones a 10% raise. Lastly, to illustrate the use of the **delete** operator, the following pair of EXTRA update queries would be used to fire Smith:

```
delete E from D in Departments, E in D.employees
where D.name = "Computer Sciences" and E.name = "Smith"

delete E from E in Employees where E.name = "Smith"
```

The deletion semantics for **own** values and **own ref** objects are that (i) they are deleted when their owner is deleted, and (ii) deleting them causes them to be removed from the database as well as from the containing object. Thus, when "Smith" is deleted from the Employees set, his object is deleted from the database and his children are recursively deleted. (Deleting a **ref** object deletes only the reference itself, and not the object.)

4. EXTENSIBILITY IN EXTRA AND EXCESS

A goal of the EXTRA data model and EXCESS query language was to provide users with the ability to define new types, along with new functions and operators for manipulating them. EXTRA supports two kinds of type

extensions: schema types and abstract data types (ADTs). Schema types were the subject of Section 2, where we described how new schema types can be constructed from base types and other schema types using the EXTRA type constructors. In this section, we describe how the set of base types can be extended by adding ADTs. We then consider schema types again, describing how functions and procedures written in EXCESS can be associated with schema types and how data abstraction can be achieved. We distinguish between these two kinds of type extensions because of the different facilities that are provided for implementing them: ADTs are written in the E programming language [Rich87], using the type system and general-purpose programming facilities provided by E. Schema types are created using the type system provided by the EXTRA data model and the higher-level but more restrictive programming facilities offered by the EXCESS query language. Finally, we discuss how EXCESS may be extended with new set functions (e.g., new aggregates).

4.1. Abstract Data Types

New base types can be added to the EXTRA data model via the EXTRA abstract data type facility. To add a new ADT, the person responsible for adding the type begins by writing (and debugging) the code for the type in the E programming language. E is an extension of C++ [Stro86] that has been designed and is currently being implemented as part of the EXODUS project. E will serve as the implementation language for access methods and operators for EXTRA/EXCESS, the target language for the query compiler, and (most importantly for our purposes here) the language in which base type extensions will be defined. E extends C++ with a number of features to aid programmers in database system programming, including "dbclasses" for persistent storage, class generators for implementing "generic" classes and functions, iterators for use as a control abstraction in writing set operations, and

```
dbclass Complex {
        float           realPart, imagPart;
public:
                        Complex(float&, float&);
                        Complex( );
        Complex         Add(Complex&);
        Complex         Multiply(Complex&);
        float           Real( );
        float           Imaginary( );
        void            Print( );
}
```

Figure 7: Contents of /usr/exodus/lib/adts/complex.h.

built-in class generators for typed files and variable-length arrays [Rich87].

Suppose that we wanted to add complex number as a new ADT. First, we would need to implement the ADT as a dbclass (much like a C++ class) in E. Figure 7 gives a slightly simplified E interface definition for the Complex dbclass. The hidden internal representation of an object of this dbclass stores the real and imaginary components of complex numbers. The Complex dbclass also provides a number of public functions and procedures (known as member functions in C++ terms). In addition to their explicit arguments, all member functions have an implicit first argument, *this*, which is a reference to the object to which the function is being applied. The Complex dbclass has two "constructor" member functions for initializing newly created complex numbers, one that uses externally-supplied values[9] and another that assumes a default initial value (e.g., 0's). In addition, the dbclass has member functions to add and multiply pairs of Complex numbers, and member functions to extract the real and imaginary parts of a Complex number. Finally, the dbclass also has a Print member function. The EXCESS query language interface requires a Print member function for all ADTs.

Once a new data type has been implemented in E and fully debugged, it must be registered with the system so that the parser can recognize its functions and the query compiler can generate appropriate E code. To register the new ADT Complex, the implementor of the type would enter the following (assuming that "complex.h" contains the Complex interface definition and that "complex.e" contains the E code that implements it):

```
define adt Complex
(
        interface = /usr/exodus/lib/adts/complex.h,
        code = /usr/exodus/lib/adts/complex.e
)
```

This tells EXTRA/EXCESS where to find the definition and implementation of the type. The system uses the definition to extract the function name and argument type information needed for query parsing and type checking. Types included in the public portion of the definition must also be registered EXTRA base types, of course. The implementation file (or list of files, in general) will be compiled and stored by the system for use when queries that reference the Complex type are encountered.

---

[9] This constructor is invoked when a Complex object is created and arguments are specified, e.g., when the target list for a copy includes an entry of the form "... complexValue = (25.0, 10.0)".

After Complex has been registered as a new ADT, it can be used like any other base type in EXTRA/EXCESS schemas, queries, and updates. The default notation for member function invocation is similar to that of C++ and E (but "( )" may be omitted for functions with no explicit arguments). For example, if CnumPair is an object in the database with two complex-valued fields val1 and val2 (or a range variable bound to such an object), then the expression "CnumPair.val1.Real" will invoke the member function that extracts and returns the real portion of the val1 field. Similarly, the expression "CnumPair.val1.Add(CnumPair.val2)" will add the val1 and val2 fields together, producing a result of type Complex. Since some users may prefer a more symmetric function call syntax, EXCESS will also accept this expression in the form "Add(CnumPair.val1, CnumPair.val2)".[10]

In addition to supporting standard ADT function invocation, we follow the lead of [Ong84, Ston86, Ston87b] and support the registration of operators as an alternative function invocation syntax. For example, to define "+" as a infix operator synonym for the Add member function of the Complex dbclass, we would enter:

define adt-op "+" **infix for** Complex Add

With this definition, the previous example's addition can be rewritten more naturally as "CnumPair.val1 + CnumPair.val2". Existing EXCESS operators can be overloaded, as illustrated here. In addition, it is possible to introduce new operators (any legal EXCESS identifier or sequence of punctuation characters may be used). For new operators, we require the precedence and associativity of the operator to be specified, much as in [Ston87b]. Prefix and postfix operators can also be defined, and the number of arguments that an operator can have is not restricted. However, functions with three or more arguments cannot be defined as infix operators, and functions that are overloaded within a single dbclass may not be defined as operators.

In order to ensure that ADTs fit into the EXTRA query language in much the same way as built-in base types do, we restrict the form and functions of their dbclasses somewhat. In particular, we require that they be true abstract data types — their data portion must not be public. We also require function (and thus operator) results to be returned by value to ensure that expressions behave in the expected manner. And, while procedures are permitted to have side effects, side effects are currently forbidden for functions so that query results will not be dependent

[10] This would be the default syntax if Add were a "friend" function [Stro86] instead of a member function; a friend function is a non-member function that is permitted to access the private portion of a class.

upon query predicate evaluation order.[11] ADT functions and operators may be used anywhere that built-in base type functions and operators may appear in queries, but ADT procedure calls may only appear in update query target lists. By default, equality and assignment operators for ADTs are predefined with bit-wise comparison/copy semantics (as in C++ and E); if this is inappropriate for a given type, the implementor of the type can replace the default definitions by explicitly overriding them in the E dbclass definition.

Finally, it is important that the query optimizer be given sufficient information to recognize the applicability of alternative access methods and join methods when optimizing queries involving ADTs. We will follow an approach similar to that outlined in [Ston86, Ston87b] for addressing this issue, with a few differences. First, optimizer-specific information will not be specified via the EXCESS/EXTRA interface. Instead, it will be given in tabular form to a utility responsible for managing optimizer information. The EXCESS query optimizer, which will be constructed using the EXODUS optimizer generator [Grae87], will do table lookup to determine method applicability for ADTs (so that ADTs can be easily added dynamically). These tables will be similar to the access method templates of [Ston86], but expression-level optimizer information (e.g., associativity, commutativity, complementary function pairs, etc.) will also be represented in tabular form at this level. Second, ADT functions and operators will be treated uniformly for query optimization purposes. This is different than [Ston87b], where operators but not functions are optimized. Third, we will support the addition of both new access methods and new join methods (written by expert users, or "database implementors") to the DBMS through extensions of the rule-based optimization techniques detailed in [Grae87].

4.2. EXTRA Schema Types, Functions, and Procedures

Schema types, which were described in Section 2, are defined using the type system provided by the EXTRA data model. In addition, we provide facilities analogous to member functions of ADTs for defining EXCESS functions and procedures to operate on schema types. However, the fact that they are defined using the EXTRA type system and the EXCESS query language means that query optimization techniques can be applied to them. EXCESS functions and procedures are inherited through the type lattice in a manner similar to inheritance for attributes, giving EXTRA an object-oriented flavor. The goal of these facilities is to provide support for derived data, for writing "stored commands" (as in the IDM-500 query facility [IDM500]), and for data abstraction of the kind

[11] We may relax this assumption eventually. Also, this is a convention, not a constraint that the system will enforce.

described by Weber [Webe78].

4.2.1. EXCESS Functions

As illustrated in an example in Section 2.3, associating functions with schema types provides a mechanism for defining derived attributes. Such functions can be defined via a single EXCESS query of arbitrary complexity, involving other portions (possibly derived) of the object being operated on and/or other named database objects. For example, we could associate a function with the Employee type to return the floor that an employee works on as follows:

```
define Employee function floor returns int4
(
        retrieve (this.dept.floor)
)
```

Such functions can then be used in queries just like regular attributes, e.g.:

```
retrieve (Employees.floor) where Employees.name = "Smith"
retrieve (Employees.name) where Employees.floor > 10
```

EXCESS functions can also have arguments other than their implicit *this* argument. For example, we could define the following function and query:

```
define Employee function youngerKids(maxAge: int4) returns int4
(
        retrieve (count(C from C in this.kids where C.age < maxAge))
)

retrieve (Employees.name, Employees.youngerKids(10))
```

This example, involving a function that counts the number of children under a certain age that an employee has, illustrates several points. First, it shows how function arguments may be defined and used. Second, it shows that a function can be defined in terms of other functions. (Recall that age was defined as a Person function in Section 2.3.) EXCESS function and procedure arguments are passed by reference.

A function may return a result of any type, including a schema type, a set of some schema type, etc. Updates through functions are not permitted. By way of comparison, EXCESS functions are similar to functions in DAPLEX [Ship81] and IRIS [Fish87]. They can also be viewed as a simplified form of POSTGRES procedure attributes [Ston87a]; in particular, they are like parameterized procedure attributes. We do not support true pro-

cedure attributes, or "EXCESS as a data type," because procedure attributes are not needed for representing complex object structures in EXTRA as they are in POSTGRES. Also, since such attributes are known only to be of type POSTQUEL in POSTGRES, and may contain an arbitrary series of POSTQUEL queries, they are problematic with respect to type-safety and knowing what to expect when writing application programs involving such attributes.

4.2.2. EXCESS Procedures

In addition to functions, we support the definition and invocation of EXCESS procedures. These differ from functions in that they have side effects, they can involve a sequence of EXCESS commands (instead of a single command), and they do not return results. As an example, suppose that we wished to write a procedure for the Employee schema type to move an employee from one department to another. We might define such a procedure as follows:

```
define Employee procedure ChangeJob(oldDept, newDept: Department)
(
        delete E from E in oldDept.employees where E is this
        insert into newDept.employees (this)
        replace (this.dept = newDept)
)
```

Then, if we wanted to move all of the employees in the Database Systems department into the Knowledge Base Systems department, we could use our new procedure to do the job as follows:

```
Emp.ChangeJob(Emp.dept, NewDept)
from Emp in Employees, NewDept in Departments
where Emp.dept.name = "Database Systems"
and NewDept.name = "Knowledge Base Systems"
```

As this example shows, procedures are called using a syntax similar to that of the EXCESS update commands.

Since not all procedures may be naturally associated with a single EXTRA schema type, and some might argue that the example above is a case in point, we also support procedures that are not bound to a particular type. (We support this option for functions as well.) For example, our employee update could be alternatively accomplished as follows:

```
define procedure ChangeJob(emp: Employee; oldDept, newDept: Department)
(
        delete E from E in oldDept.employees where E is emp
        insert into newDept.employees (emp)
        replace (emp.dept = newDept)
)

ChangeJob(Emp, Emp.dept, NewDept)
from Emp in Employees, NewDept in Departments
where Emp.dept.name = "Database Systems"
and NewDept.name = "Knowledge Base Systems"
```

This kind of procedure is similar in flavor to the stored commands of the IDM database machine [IDM500]. However, it is much more general, as we support the use of a **where** clause for binding procedure parameters and we invoke the procedure for all possible bindings (instead of just once, with constant parameters).

Aside from the obvious syntactic differences between this example and the previous one, the major difference between procedures that are associated with an EXTRA schema type and those that are not type-specific is a matter of inheritance semantics. In the type-specific case, we may choose to redefine the ChangeJob procedure for some subtypes of Employee, such as WorkStudyStudent, but not for others, like NightEmployee. Then, if ChangeJob is invoked on elements of a set containing instances of Employee and its subtypes, WorkStudyStudent's ChangeJob will be used for WorkStudyStudent instances, while Employee's ChangeJob will be used for Employee and NightEmployee instances. That is, procedures can be inherited via the inheritance mechanism outlined in Section 2. If ChangeJob is not associated with a particular type, while its definition may be overloaded, a single definition (i.e., the definition applicable to the Employee type) will be statically selected for such a query. This is similar to the distinction between virtual member functions and regular member functions in C++ [Stro86].

4.2.3. Achieving Data Abstraction

We plan to provide an authorization mechanism along the lines of the System R [Cham75] and IDM [IDM500] protection systems. Both individual users and user groups (including a special "all-users" group) will be recognized, and protection units will be specified via EXCESS queries. Grantable/revocable rights will include the ability to read and modify specified objects or individual attributes, the ability to invoke specified functions and procedures, the ability to define new functions and procedures for specified types, etc. While full details of this protection system still have to be specified, it will serve two purposes: First, it will act as an authorization mechanism for protecting database objects against unauthorized access and modification, as is typical. Second, it will provide a

means for achieving data abstraction (i.e., encapsulation) for EXTRA schema types. For example, one could choose to grant access to a given schema type only via its EXCESS functions and procedures, effectively making the schema type an abstract data type in its own right. (In fact, IDM stored commands are recommended for regulating database activity in a similar way [IDM500].) Features such as the modules of [Webe78] or the object semantics of an object-oriented data model can thus be captured via a single, more general mechanism.

4.3. Generic Set Operations

The final form of extension that we intend to support in the EXCESS query language is a facility for adding new, user-defined set functions to the language. At a logical level, such a function can be viewed as taking a set of elements of some type T1 as its argument and returning an element of type T2 as its result. T1 can be any type that satisfies certain constraints, while T2 must be either a specific type or the same type T1. Aggregate functions are a classic example here: The "min" function takes a set of values of any type on which a total order is defined, returning the smallest element of the set. The "count" function takes a set of values as its argument, returning the number of elements in the set (i.e., an integer result). One could imagine adding new functions of this sort, such as "median" or "std-deviation" for use in statistical applications.

Support for this form of extension has been included in POSTGRES, but in a limited form. In particular, one could introduce a "median" aggregate function for sets of integers, but not one that works for *any* totally ordered type [Ston87b]. The EXCESS approach to such extensions is based on features provided by the E programming language [Rich87]. E provides a facility for writing generic functions, and it supports the specification of constraints on the generic type (e.g., any type that has boolean "less_than" and "equals" member functions). E also provides a construct, called an iterator function, for returning sequences of values of a given type. Implementing a new aggregate will involve writing a generic E function with one argument, an iterator that yields elements of an appropriately constrained type T; the generic function should produce a result of either the same type T or else some base type. This function can then be registered with EXCESS and used with any type that meets its constraints.

5. COMPARISON WITH OTHER WORK

The EXTRA data model and EXCESS query language designs represent a synthesis and extension of ideas drawn from a number of other data models (both past and present). The data structuring facilities of the EXTRA

data model are probably closest to those of GemStone [Maie86], as GemStone provides both tuple and array constructors, and data is organized into user-maintained extents rather than system-maintained type extents. EXTRA's **ref** notion was based on GEM reference attributes [Zani83], and **own ref** is closely related to composite objects in ORION [Bane87]. EXTRA also goes beyond these systems in certain ways, however. In particular, to our knowledge, EXTRA's mix of **own**, **ref**, and **own ref** attributes yields a relatively unique mix of (structural) object- and value-orientation. Neither GemStone nor Orion have support for **own** attributes (except perhaps in implementing their small atomic types). And while GEM had both **own** and **ref** attributes, its type system was less rich and sets of references were not permitted.

The EXCESS query language is related to those of DAPLEX [Ship81], GEM [Zani83], NF^2 systems [Dada86, Sche86], and POSTGRES [Rowe87]. Implicit joins were taken directly from GEM, and originated in DAPLEX. The EXCESS treatment of queries over nested sets is similar in flavor to that of NF^2 query languages, although the path syntax for handling deeply nested queries was influenced by DAPLEX and the early STDM paper [Cope84]. Our handling of range variables was heavily influenced by that of POSTGRES [Rowe87]. In addition, EXCESS goes beyond its predecessors in several respects. Our mix of object- and value-oriented semantics, again, is unique among query languages that we have examined. Also, EXCESS provides a cleaner treatment of arrays than we have seen elsewhere. The only point for comparison here is POSTQUEL, which only operates on one-dimensional arrays of base types. Comparing EXCESS to POSTQUEL in general, we feel that omitting procedures as instance-level field values leads to cleaner and more consistent query language semantics.

Finally, ADT and access method extensibility in EXTRA/EXCESS were heavily influenced by the work of Stonebraker [Ston83, Ston86] and the resulting extension facilities in POSTGRES [Ston87b]. Our work here differs mostly in minor respects. Because ADTs in our system are written E, the system's internal language, adding ADTs is perhaps simpler here. Another difference is that we view ADT operators as synonyms for function calls rather than something to be handled differently for query optimization purposes. Lastly, our approach to user-defined set functions is more general than the corresponding POSTGRES approach to user-defined aggregates. With respect to schema types, our support for EXCESS functions is similar to the functions of DAPLEX [Ship81] and IRIS [Fish87], and also to the parameterized procedures of POSTGRES [Ston87a]. Our approach to user-defined procedures is rooted in the stored commands of the IDM database machine [IDM500], as is our approach to encapsulation through authorization, but EXCESS procedures are a much more general mechanism.

6. SUMMARY AND FUTURE WORK

In this paper we have presented the design of the EXTRA data model and the EXCESS query language. EXTRA and EXCESS represent a synthesis and extension of many ideas from other data models, including GEM, NF^2 models, DAPLEX, ORION, POSTGRES, GemStone, and O_2. Our extensions include: complex object support based on an interesting mix of object- and value-oriented semantics; a user-friendly, high-level query language that captures these semantics; support for the storage and manipulation of a database of persistent objects of any type, not just sets of tuples; and support for user-defined types, functions, and procedures, both at the abstract data type level and at the conceptual schema level. An implementation of this data model and query language will serve as a demonstration vehicle for the EXODUS extensible database system project.

Our plans for continued work on EXTRA and EXCESS include a number of interesting problems and issues. At the logical level, since we would like the system to be useful for CAD-type applications, we need to extend the model with some form of version support and a facility for check-out and check-in of objects. In addition, we need to address the problem of coupling the system with a general-purpose programming language (perhaps E). Finally, we will face type evolution issues at two levels — for ADTs, and for EXTRA schema types. At the next level down, we need to design a query algebra upon which to construct a query optimizer using the EXODUS optimizer generator [Grae87], and we need to carefully design the tables for linking ADTs to access methods and join methods. Finally, there are a number of interesting problems to be worked out in designing the physical level of the system, including object indexing, support for small versus large sets and arrays, object clustering, and the possibility of enhancing performance through explicit replication of shared objects.

ACKNOWLEDGEMENTS

During the process of designing EXTRA and EXCESS we were fortunate to have the opportunity to discuss our ideas with a number of knowledgeable visitors, including Francois Bancilhon, Carlo Zaniolo, Dave Maier, and Won Kim. These individuals helped us to clarify our design for the EXTRA type system and the semantics of our query language, and they provided us with a wealth of information and advice on object-oriented database systems. While these individuals should not be blamed for the results of our efforts, we think it is fair to say that the design would not be nearly as good were it not for their generous help. We would also like to thank Joel Richardson for helping us clarify the semantics of the EXTRA data model and for discussions regarding extensibility issues.

REFERENCES

[Andr87] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment," *Proc. of the 2nd ACM OOPSLA Conf.*, Orlando, FL, 1987.

[Banc86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," *Proc. of the ACM-PODS Conf.*, Cambridge, MA, March 1986.

[Bane87] J. Banerjee et al., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[Bato87] D. Batory, "Principles of Database Management System Extensibility," *Database Eng.*, June 1987.

[Bloo87] Bloom, T., and Zdonik, S., "Issues in the Design of Object-Oriented Database Programming Languages," *Proc. of the 2nd ACM OOPSLA Conf.*, Orlando, FL, 1987.

[Care86] M. Carey et al., "The Architecture of the EXODUS Extensible DBMS," *Proc. of the Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Care87] M. Carey and D. DeWitt, "An Overview of the EXODUS Project," *Database Eng.*, June 1987.

[Cham75] D. Chamberlin et al., "Views, Authorization, and Locking in a Relational Database System," *Proc. Natl. Computer Conf.*, Anaheim, CA, 1975.

[Codd79] E. Codd, "Extending the Relational Model to Capture More Meaning," *ACM Trans. on Database Sys.* 4(4), Dec. 1979.

[Cope84] G. Copeland and D. Maier, "Making Smalltalk a Database System," *Proc. ACM-SIGMOD Conf.*, Boston, MA, 1984.

[Dada86] P. Dadam et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," *Proc. of the ACM-SIGMOD Conf.*, Washington, DC, 1986.

[Ditt86] K. Dittrich, "Object-Oriented Database Systems: The Notion and the Issues," *Proc. of the Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Fish87] D. Fishman et al., "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[Grae87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. of the ACM-SIGMOD Conf.*, San Francisco, CA, May 1987.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[IDM500] IDM 500 Software Reference Manual, version 1.4, Britton-Lee Inc., Los Gatos, CA.

[Kemp87] A. Kemper and M. Wallrath, "An Analysis of Geometric Modeling in Database Systems," *ACM Comp. Surveys* 19(1), March 1987.

[Khos86] S. Khoshafian and G. Copeland, "Object Identity," *Proc. of the 1st ACM OOPSLA Conf.*, Portland, OR, Sept. 1986.

[Khos87] S. Khoshafian and P. Valduriez, "Sharing, Persistence, and Object Orientation: A Database Perspective," DB-106-87, MCC, Apr. 1987.

[Kim87] W. Kim et al., "Composite Object Support in an Object-Oriented Database System," *Proc. of the 2nd ACM OOPSLA Conf.*, Orlando, FL, 1987.

[Klau85] A. Klausner and N. Goodman, "Multirelations — Semantics and Languages," *Proc. of the 11th VLDB Conf.*, Stockholm, Sweden, 1985.

[Klug82] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *JACM* 29(3), July 1982.

[Lecl87] C. Lecluse et al., "O_2, an Object-Oriented Data Model," unpublished manuscript, Altair, Paris, France, Sept. 1987.

[Lohm83] G. Lohman et al., "Remotely-Sensed Geophysical Databases: Experience and Implications for Generalized DBMS," *Proc. of the ACM-SIGMOD Conf.*, San Jose, CA, 1983.

[Maie86] D. Maier et al., "Development of an Object-Oriented DBMS," *Proc. of the 1st ACM OOPSLA Conf.*, Portland, OR, 1986.

[Mano86] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Asilomar, CA, Sept. 1986.

[Mylo80] J. Mylopoulos et al., "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. on Database Sys.* 5(2), June 1980.

[Nixo87] B. Nixon et al., "Implementation of a Compiler for a Semantic Data Model: Experiences with TAXIS," *Proc. of the ACM-SIGMOD Conf.*, San Francisco, CA, 1987.

[Ong84] J. Ong et al., "Implementation of Data Abstraction in the Relational Database System INGRES," *ACM-SIGMOD Record* 14(1), March 1984.

[Rich87] J. Richardson and M. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proc. of the 1987 ACM-SIGMOD Conf.*, San Francisco, CA, May 1987.

[Rowe87] L. Rowe and M. Stonebraker, "The POSTGRES Data Model," *Proc. of the 13th VLDB Conf.*, Brighton, England, 1987.

[Scha86] C. Schaffert et al., "An Introduction to Trellis/Owl," *Proc. of the 1st ACM OOPSLA Conf.*, Portland, OR, Sept. 1986.

[Sche86] H.-J. Schek and M. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Sys.* 11(2), 1986.

[Schw86] P. Schwarz et al., "Extensibility in the Starburst Database System," *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, 1986.

[Ship81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Sys.* 6(1), March 1981.

[Ston76] M. Stonebraker et al., "The Design and Implementation of INGRES," *ACM Trans. on Database Sys.* 1(3), Sept. 1976.

[Ston83] M. Stonebraker et al., "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," *Proc. 1983 ACM-IEEE Data Base Wk.*, San Jose, CA, May 1983

[Ston86] M. Stonebraker, "Inclusion of New Types in Relational Database Systems," *Proc. of the 2nd Int'l. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1986.

[Ston87a] M. Stonebraker et al., "Extending a Database System with Procedures," *ACM Trans. on Database Sys.* 12(3), Sept. 1987.

[Ston87b] M. Stonebraker et al., "Extendability in POSTGRES," *Database Eng.*, June 1987.

[Ston87c] M. Stonebraker, personal communication, 1987.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[Ullm87] J. Ullman, "Database Theory — Past and Future," *Proc. of the ACM-PODS Conf.*, San Diego, CA, March 1987.

[Webe78] H. Weber, "A Software Engineering View of Database Systems," *Proc. of the 4th VLDB Conf.*, 1978.

[Zani83] C. Zaniolo, "The Database Language GEM," *Proc. of the ACM-SIGMOD Conf.*, San Jose, CA, 1983.