1988

Engineering the Object-Relation Database Model in O-Raid

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Prasun Dewan

Stephen Leung

Report Number:
88-781

Engineering the Object-Relation Database Model in O-Raid[1]

Bharat Bhargava
Prasun Dewan
Stephen Leung
Department of Computer Science
Purdue University
W. Lafayette, IN 47907

ABSTRACT

This paper presents the design and implementation of the O-Raid system which has been developed by extending an existing distributed database system called Raid. The system design encompasses the simplicity of the relational model, the extensibility of the object-oriented model, and the interactive aspects of language-oriented editors. The resulting system has several novel properties. Objects, classes, and inheritance are supported together with a predicate-based relational query language. A hierarchy of column protocols define the common properties of objects in a particular relation column. Relations may contain heterogeneous objects that can individually evolve by being reclassified. Special facilities are provided to reduce the data search in a relation containing complex objects. A structure-editing interface integrated with the query language allows the editing of complex objects. A special query supports continuous display of objects in the database.

[1]This research is supported by NASA and AIRMICS under grant number NAG-1-676, and UNISYS.

1. INTRODUCTION

To deal with complex database applications such as those requiring design databases used in manufacturing, and geometric, geographical and image databases, there is a need to extend the relational database model and its implementation. Several research efforts are working to develop an independent model based on the concept of objects. The object-oriented model does not support many of the time tested facilities offered by the relational model such as queries for projecting and joining relations. Since much research has been done on both the theory and implementation of the relational model, we extend this research to implement a hybrid object-relation model. Our design ideas draw upon the research in object-oriented systems and language oriented editors. This approach is of interest to venders of existing relational implementations.

We have engineered this model in the O-Raid system by extending an existing relational database system called Raid [3]. Raid is a distributed database system that provides complete support for transaction processing, including transparency to concurrent access, crash recovery, distribution of data, and atomicity. Database sites communicate over an Ethernet network. More details of Raid are given in section 3 under implementation of O-Raid. Other systems supporting this model include Postgres [24, 31], Exodus [7], and DSM [3]. In O-Raid, we take the Postgres approach of extending a relational model with object-oriented features. Unique features of our approach include a hierarchy of column protocols, facilities to reduce the data search for complex objects, relations containing heterogeneous objects that can individually evolve by being reclassified, an integrated query and structure editing language, and continuous display of objects.

In O-Raid, we support the object-relation model by allowing relation attributes to be arbitrary objects. Each object is associated with an instance protocol describing its individual behavior and a column protocol describing its collective behavior as part of a relation column. These behaviors are defined by classes and column classes respectively, which are arranged in parallel inheritance hierarchies.

O-Raid provides an SQL-like query language for creating, viewing, and modifying objects. A query language, however, is insufficient for manipulating complex objects, since new users of such object have to learn their elaborate protocol and structure. Therefore, O-Raid also provides a structure-editing language allowing the user to see both the structure and protocol of the objects being edited. Both languages are useful for manipulating objects: The query language provides a way to make a set of changes in "batch", modify an object without going through the overhead of displaying it, and select a set of objects for editing, while the structure-editing language provides an interactive and visual interface for making incremental changes. The query and structure-editing languages are "integrated" in the sense that queries can be used as structure-editing commands.

O-Raid allows a user to enter the editor in the "readonly" mode to display a set of objects. The visual representations of these objects are kept consistent with their values in the database. As a result a user can continuously monitor the values of objects without going through the overhead of "polling" the system.

In the rest of this introduction, we present our understanding of the limitations of the relational and object models. We motivate the reader towards the object-relation model. In section 2, we present

our perception of this model in the context of the O-Raid system. In section 3, we present the software changes that are necessary to engineer O-Raid from the existing Raid system. In appendix A, we explain how we accomplish the implementation of O-Raid. In section 4, we outline the directions for further research that will make the relational database systems more responsive to new applications.

1.1. Motivation for the Object-Relation Data Model

In the following discussion we motivate the object-relation model by discussing the limitations of the relational and object-oriented model.

Limitations of the Relational Model

The relational model [8] represents real-world entities and their relationships in sets or *relations*. A relation can be considered as a two-dimensional table where each row represents a different entity, and each column represents a common property of the entities in the relation. In relational model terminology, the rows and columns are called *tuples* and *attributes* respectively. Relational database provides query languages based on alebraic and predicate calculus languages.

While the relational model is simple, it has several limitations that are well documented in the database literature (recently in [4, 24, 26] and [29]). These limitations include lack of support for:

- *Complex Structures*: Typically, a relational database restricts attributes to integers, reals, and fixed-length strings. As a result, complex data structures such as nested records, unions, and sequences need to be flattened into these simpler values. This limitation results in both awkwardness of use and inefficiency, as several relations may have to be joined to retrieve the flattened representation of a complex entity.
- *Semantic Checks*: A relational database cannot ensure that only semantically correct modifications are made to the database. For instance, it cannot ensure that an attribute is readonly or that modifications to it are consistent with values of other related attributes. As a result, the database may be left in an inconsistent state unless a separate subsystem to enforce integrity assertions is invoked.
- *Semantic Actions*: Often a user may desire that an update to an entity result in certain *side effects* or *semantic actions*. For instance, a user viewing an entity may wish to display its new value whenever the entity is updated. Similarly, a system administrator may want a report printed whenever a new bug is reported by a user. In the absence of support for such actions, each user who updates the entity has to ensure that the appropriate semantic action occurs. As a result, the system is less automated and prone to missed actions.
- *Generalization*: A user cannot create new entities as special cases of existing entities. As a result, information has to be duplicated in all specializations of a generic structure. Changes to the definition of the generic structure require changes to all its specializations.

Several solutions have been proposed to overcome these limitations. These include the logical relational design methodology [32], alerters, triggers, and constraints [5, 6, 24] and the object-oriented data model [11, 15, 18, 26].

The Object-Oriented Model and Its Limitations

The object-oriented model was motivated initially by the Smalltalk programming language [13]. In Smalltalk, information is encapsulated in *objects*, which respond to *messages* from other objects. In response to a message, an object executes a *method* which can manipulate the state of the object stored in the *instance variables* of the object. Each object is an *instance* of a *class*, which describes the instance variables and methods of the object. A class can be a *subclass* of another class, in which case it inherits the instance variables and methods of its *superclass*.

While the object-oriented model uses a terminology distinct from the one used in the relational model, components of it have direct counterparts in the relational model. Classes corresponds to relations, objects corresponds to tuples, instance variables correspond to attributes, and messages correspond to relational queries. However, the notions of user-defined classes and subclassing have no counterparts in the relational model. These features allow the object-oriented model to overcome the limitations of the relational model, as shown below:

- *Complex Data Structures*: Smalltalk class declarations can be used to describe complex structures such as nested records, arrays, sequences (called *collections*), and unions (simulated by subclassing).
- *Semantic Checks and Actions*: An object cannot directly manipulate the instance variables of another object. Instead, it needs to execute the methods of the object, which in turn can change the state of the object. These methods can ensure that only semantically correct changes are made to the object. Moreover, they can send messages to other objects to perform semantic actions.
- *Generalization*: Subclassing allows new classes to be created as specializations of existing classes.

The Smalltalk version of the object-oriented model, however has several limitations of its own (see [17] for a related discussion of this topic):

- *Limited Object Space*: The original Smalltalk-80 system required that all objects fit in a single physical address space. Later implementations, such as LOOM [14], expanded the object space to a single virtual address space, which is still fairly small.
- *Lack of Sharing and Protection*: Smalltalk is a single user system and thus does not support sharing or protection.
- *Lack of Transactions*: Smalltalk does not support the notion of a transaction. As a result, data may be left in an inconsistent state if the system crashes while a method is executing.
- *Programmer Overhead*: For each class, a programmer has to explicitly provide methods for reading and writing its instance variables. Thus definition of a new class incurs more overhead than definition of a new relation, even when no special semantic checks or actions are necessary.
- *Lack of Predicate-Based Query Language*: Unlike a relation, a class can be queried only for all its instances. It does not support predicate-based selections, projections, and joins provided by relational databases.

Some of these limitations are not inherent to the general notion of objects, and have been overcome in subsequent object-oriented systems designed specifically to meet these needs. The limited object space problem has been reduced in GemStone [17], Eden [1], Clouds [30], and Argus [16], by letting objects occupy a large number of independent address spaces instead of a single contiguous address space. Sharing and protection are supported by Eden, Clouds and GemStone by allowing only objects with appropriate access rights to send messages to an object. (The access rights are stored in capability lists in Eden and Clouds and in access lists in GemStone). Transactions are supported in GemStone, Clouds, and Argus by ensuring that certain sequences of actions occur atomically.

The Orion [2] system illustrates how the problem of programmer overhead and lack of support for a predicate-based selection can be reduced. For each instance variable declared in a class, the system automatically provides a message (with the same name) that can be used to read that variable in each instance of the class. Orion also embellishes the class protocol with the message select which may be used to perform predicate-based selections on members of the class. However, Orion, like other object-oriented systems, does not provide messages supporting projections or joins. The reason for this limitation is that object-oriented systems require that messages return objects of existing classes. Therefore "projection" messages cannot be supported since no existing class may define the subset of the instance variables of an object projected out by the message. Similarly, "join" messages cannot be supported since no existing class may define (a subset) of the union of the instance variables of two objects. Perhaps appropriate classes can be dynamically created when such messages are sent. However, it is not clear what protocol they should define. Moreover, the overhead of creating a new class at query-resolution time may be unacceptible. Therefore, the inability to support projections and joins seems a fundamental property of the object-oriented model.

Steps Towards the Object-Relation Model

The above discussion shows that both the relational and object-oriented models have inherent limitations. The former does not support complex data, semantic checks and actions, and generalization, while the latter does not support projections and joins. We explore the notion of a hybrid object-relation model. This model supports objects, classes, and inheritance (object-oriented features, as defined in [33]) and predicate-based selections, joins, and projections. Our definition is consistent with the usage of this term in [25]. In section 2, we present our interpretation of this model in detail.

Several systems currently support this model using different approaches. Postgres [24, 31], supports it by extending the Ingres relational model with object-like entities. Like Ingres, Postgres supports relations with attributes. However, each tuple in a relation essentially forms an object consisting of data attributes (instance variables) and procedures (methods). A new relation can be specified as a "sub-relation" of an existing relation, in which case it inherits the attributes of the "super-relation". Postgres extends the QUEL query language to allow execution of procedure attributes and specify inheritance among relations.

Data attributes in Postgres are restricted to simple values and arrays. Thus tuples cannot store complex structures such as hierarchical records and unions. Postgres requires that these and other complex structures be accessed via procedures responsible for extracting the flattened representation of these

structures stored in multiple relations. The inefficiency of joining these relations is reduced by precomputing or caching the results of a procedure attribute in the field itself (for small answers) or in a separate relation (for large answers).

Exodus [7], and DSM use the converse approach of extending an object-oriented system with relation-like entities. Exodus provides *sets* of object values and references, and supports queries for predicate-based selections, joins, and projections. However, the results of projections and joins cannot in general be stored in new sets since sets need to contain objects of existing types.

DSM provides a special class called *relation*, which is a subclass of *unordered collection*, and provides the functionality of binary and qualified relations, which are stored as hash tables in virtual memory. Currently, it does not support joins among these relations, or relations of degree greater than three. Projection messages are provided that can dynamically create relations of degree one, two or three. An extension of DSM supporting relations of arbitrary degree will be reported soon.

The following section describes how we have engineered this model in O-Raid.

2. OBJECT-RELATION MODEL IN O-RAID

In O-Raid, we support the object-relation model by extending the Raid relational model. The main components of O-Raid include:

- Relations with objects as attribute values.
- Class declarations, used for defining instance and column protocols.
- Basic SQL-like queries for creating, viewing, and modifying objects.
- Heterogeneous objects in a relation.
- Structure-editing of objects selected by an EDIT query.
- Integration of structure-editing and query languages by allowing manipulation of window relations.
- Continuous display of objects selected by a DISPLAY query.

These components are discussed below.

2.1. Relations

O-Raid relations are like Raid relations except that attributes can be objects described by user-defined classes. These objects can be composite and have attributes of their own. A tuple, however, is not an object, that is, it is not described by an existing class. This feature allows us to dynamically create relations containing elements that do not belong to existing classes. A single-attribute tuple may, however, be considered an object since there is no distinction between the tuple and its unique attribute. A composite attribute may be used as a key if its class has defined comparison operators obeying transtivity and other rules described in [31] that are necessary for using that attribute as an index.

Each column of a relation is represented by a *column object* which is an instance of a *column class*. This object contains the common properties of the objects in that column such as font, indentation,

access list, title of the column, and common state shared by all objects in the column.

2.2. Class Declarations

An O-Raid class declaration defines the individual and collective behavior of instances of the class. These behaviors are defined by the *instance protocol* and *column protocol* parts respectively of the class declaration.

Instance Protocol

The instance protocol defines the instance variables, attributes, and methods of the object. The instance variables, like their Smalltalk counterparts, keep the private state of the object, while the instance attributes keep the public state of the object. An instance attribute may be declared as readonly, in which case it can be examined by an external object but may be modified only by the instance methods of the object. The instance methods define the set of messages to which instances of the class respond, and are compiled into machine code.

Dividing the state of an object into instance variables, and readonly and modifiable attributes serves the following purpose: It relieves the programmer from the overhead of defining trivial methods that simply read and write its state, while allowing him to support data abstraction, semantic checks, and semantic actions when necessary. Instance variables can be used to keep implementation-dependent data of the object. Readonly attributes can be used to keep public data associated with semantic checks and actions. Modifiable attributes can be used to keep public data that does not need such protection. Thus a user can use the power of the object-oriented model without sacrificing the simplicity of the relational model.

Column Protocol

The column protocol part of an O-Raid class declaration corresponds to the class protocol part of a Smalltalk class declaration. Like the latter, it defines the collective behavior of instances of the class. The main difference is that the class protocol defines a single class object representing *all* instances of the class while the column protocol defines several *column objects*, each one of representing a column in which objects of that class have been put.

A column object, is like an ordinary object, has a private, public readable, and public modifiable state (defined by the column variables, readonly attributes, and modifiable attributes respectively), and can respond to messages by executing column methods. The state of a column object is accessible to all members of the column it represents, just as the state of a Smalltalk class object is accessible to all members of the class.

The class of a column object is called a *column class*, and corresponds to a Smalltalk metaclass. Each (non-column) class, *C*, is associated with a unique column class, *C column*. These two classes are defined by the instance and column protocols respectively of the class declaration associated with *C*. Like Smalltalk metaclasses, column classes are placed in an inheritance hierarchy of their own. This hierarchy parallels the inheritance hierarchy of the member classes. That is, if class *A* is a subclass of class *B*, then *A column* is a subclass of *B column*.

The column class *object column*, is the superclass of all column classes, and defines properties common to all columns such as time of last modification, number of elements in the column, title of the column, window in which the objects are displayed, shared state, access list, etc. These properties can be overridden/augmented in each column class. (The class *object* is the superclass of all non-column classes.)

Note that column classes do not compete with the notion of Smalltalk metaclasses. In O-Raid metaclasses can also be supported by adding a class protocol part to a class description. This protocol would define the collective behavior of all instances of the class instead of the instances in a certain column of a relation. In a later version of O-Raid, we plan to explore support for Smalltalk-like metaclasses.

The following example illustrates class declarations in O-Raid for classes *shape*, *rectangle*, and *triangle*.

```
class shape
superclass object
column attributes
        int Max_x, Min_x, Max_y, Min_y;
instance attributes
        int x_coord, y_coord; /* coordinates of the centroid of the object */
instance methods
        int distance (other_shape)
                shape   other_shape
        { ... }

        bool "<" (other_shape)
                shape   other_shape
        { ... }
```

The instance attributes *x_coord*, and *y_coord* are kept in each instance of a shape and contain the coordinates of its centroid. The instance method *distance*, calculates the distance between an instance and the object *other_shape*. The instance method "<" compares the object with another object. The column attributes *Max_x*, *Min_x*, *Max_y*, and *Min_y* are kept in each relation column in which shapes are stored, and defines the area to which these objects are confined.

Class *rectangle*, defining rectangles, can be created as subclass of *shape:*

```
class rectangle
superclass shape
instance variables
        int lower_left_x, lower_left_y, upper_right_x, upper_right_y;
instance methods
        void init(...)
          ...
        int area()
          ...
        void magnify(...)
          ...
```

In a similar manner class *triangle* can be created to describe triangles:

```
class triangle
superclass shape
instance variables
        int x1, y1, x2, y2, x3, y3;
instance methods
        void init(...)
          ...
        int area()
          ...
        void magnify (...)
          ...
```

These classes embellish the instance protocol of *shape* with the messages *init*, *area*, and *magnify*. The descriptions of instances of these objects are stored in variables instead of attributes to allow the implementation of these objects to be changed without affecting their interface.

Predicate Methods

O-Raid supports *predicate methods* to provide efficient search of complex structures. We motivate this concept through an example. Assume that a boolean method, *overlap*, is defined in class rectangle:

```
bool overlap (rect)
        rectangle rect;
{ return( (lower_left_x <= rect.upper_right_x) &  (lower_left_y <= rect.upper_right_y) & ...))}
```

This method determines if the current object overlaps with its argument. Now assume that the method is used to select from a set of rectangles all those that overlap a particular rectangle. O-Raid has no choice but to invoke the compiled method on *all* members of the relation and return those that satisfy the condition. The search could be made more, efficient, however, if the variables *lower_left_x*, *lower_left_y*, etc that are used in the expression had been defined as keys in the relation containing these rectangles. The system could then use a suitable physical representation of the relation such as a multi-index K-D-B tree [22] to reduce the search space at query resolution. However, the query processor would need to *interpret* this expression in order to perform the comparisons.

O-Raid supports interpretation of methods by allowing an instance method to be made a *predicate method*. Such a method contains a boolean expression that is interpreted by the query processor at run

time. The query processor uses the expression to determine its search path through a B-Tree or K-D-B tree created to store the relation. The user can indicate the keys of this relation at relation-creation time, as described in section 2.3, or at class creation time by using an **indices** declaration. For instance, in the above example, the user may specify *upper_right_x*, and *upper_left_y* as keys at class creation time by declaring:

indices upper_right_x, upper_left_y;

Any relation containing rectangles will then use *upper_right_x*, and *upper_left_y* as keys.

2.3. Queries

O-Raid provides SQL-like queries for creating relations, inserting and deleting tuples in a relation, reading and writing attributes, and sending messages to objects. The syntax for the query language is given in appendix B. We illustrate its use by the following examples.

A user may create a new relation called *triangles*

```
CREATE INDEX ON (obj) triangles (name: char[20], obj: triangle)
```

with attributes *name* and *obj* and key *obj*, and then add a particular element to it

```
INSERT INTO triangles
        VALUES (name = "T1", obj = (x_coord: 0, y_coord: 0), obj init(...))
```

sending it the appropriate initializing message. In the above example, "(x_coord: 0, y_coord: 0)" is an aggregate describing a constant object. Notice that only the modifiable instance attributes *x_coord*, and *y_coord* can be initialized directly. The instance variables *x1*, *y1*, ..., *y3* have to be initialized by sending the *init* message. The relation can be updated by either assigning new values to its objects or sending them messages as illustrated below:

```
UPDATE INTO triangles
        SET (obj =  (x_coord: 1, y_coord: 1), obj magnify(...))
        WHERE ((obj < (x_coord: 0, y_coord:0)) & ((obj area()) <= 9))
```

The predicates involved in a WHERE clause can use results of method invocations, as illustrated by the above example. Since tuples in O-Raid are not objects, they can be projected and stored in new relations, as shown below:

```
SELECT (name) INTO temp FROM triangles WHERE  ((obj.x_coord > 0) & (obj.y_coord > 0))
```

Queries are also provided to manipulate the column objects associated with the columns of the relation. For instance, a user may update the column attributes of the *obj* column of *triangles* by invoking

```
UPDATE INTO triangles
        SET (obj = (Max_x: 100, ..., Min_y: 0, title: "Object"))
```

The query updates the attributes *Max_x*, *Max_y*, *Min_x*, *Min_y*, which are declared in class *shape column*, and *title*, which is inherited from class *object column*. The column name *obj* can be used to specify both an individual object and the relation object since names in the column protocol do not conflict with names in the instance protocol.

2.4. Support for Heterogeneous Objects

O-Raid relation columns can contain heterogeneous objects, that is, instances of different classes. One way to create such a column is to specify a common superclass of the valid objects in the column. Any object that has that class in its superclass chain may be put in that column. Such a column is specified by putting a "*" after the name of the common superclass. For instance, the relation *shapes* containing any shape may be created as follows:

```
CREATE INDEX ON (obj.x_coord, obj.y_coord)
        shapes (name: char[20], obj: shape*)
```

Only attributes of the named superclass can be used as keys, even if the current objects in the column have a more specialized common superclass. This technique for supporting heterogeneous objects is derived from object-oriented systems such as GemStone that allow elements in a collection to belong to any subclass of a given class.

Another way to create a heterogeneous column is to enumerate the various classes to which objects in the column can belong. This feature may be useful if a user wants to prevent the relation from containing objects of *any* current or future subclass. For instance, a user may want to create a relation containing rectangles and triangles but not all their specializations such as windows. Such a relation may be created in O-Raid by executing the query

```
CREATE INDEX ON (obj.x_coord, obj.y_coord)
        triangles_or_rectangles (name: char[20], obj: (triangle + rectangle)
```

Only attributes defined in the common superclasses of the enumerated classes can be used as keys.

The system keeps with each object in a heterogeneous column a tag indicating its class so that it may be accessed in a type-safe manner. This class is specified when the object is inserted into the relation, as shown below:

```
INSERT INTO triangles_or_rectangles
        VALUES (name = "T1", (triangle) obj = (x_coord: 0, y_coord: 0), obj init(...))
```

Objects often evolve and change their characteristics. For instance, a *person* object may become a *student* object, and later an *employee* object. Similarly, the shape of an object may change from a *triangle* object to a *rectangle* object. O-Raid supports such evolution by allowing the class of an object in a heterogeneous column to be changed to one of the valid classes for that column. Appropriate fields are added and deleted in the tuple for that object while the values of common fields are left unchanged. A special *reclassify* message may be sent to an object to change its class. Thus the query

```
UPDATE INTO triangles_or_rectangles VALUES (obj reclassify (rectangle), obj init(...))
        WHERE (name == "T1")
```

changes a triangle to a rectangle while maintaining its position.

This technique of reclassifying *individual* objects complements the work done in GemStone [21] and Orion [15] to allow evolution of *all* objects in a class by supporting changes to the class definition. We plan to explore such evolution in the next version of O-Raid.

2.5. Structure Editing

The query interface requires the user to know the names and types of the instance and column attributes of the objects in a relation and the messages that can be send to them. For instance, a user who enters the query

```
UPDATE INTO rel SET (obj = (f1: (f11: 3, f12: true), f2 = "foo", obj.f1 m1(3, true))
        WHERE (obj.f3 == 3.0)
```

needs to know that *f1* and *f2* are modifiable attributes of *obj*, *obj.f1* refers to a structured object consisting of modifiable attributes *f11* and *f12*, and responding to the message *m1*, and so on.

Relational databases supporting form-interfaces [23] have illustrated how the problem of remembering attribute names can be eliminated. These systems display tuples in *forms*, which contain slots for the names and values of the attributes. For instance, a tuple with attributes *f1* and *f2* may be displayed in the form

```
f1: 5
f2: "a string"
```

which may be edited to modify the attributes.

The form interface model cannot be directly applied to O-Raid since it assumes that the structure of a tuple is fixed, and that the attributes of a relation are simple values. However, in O-Raid, tuples can have a variable structure, and objects can be arbitrarily nested. Therefore, we have developed a more sophisticated model of interaction based on the notion of structure-editing [19] that is suitable for systems supporting complex objects.

We illustrate this interaction model with the following example:

```
class person
superclass object
instance attributes
        char[20] name;
        int age;
        char[20] address;
        char[20] e_mail;
        char [20] tel_no;

class student
superclass person
instance readonly attributes
        char[20] major;
        char[20] minor;
        float gpa;

class db_student
superclass student
column attributes
        real A_cutoff, B_cutoff, C_cutoff, D_cutoff;
instance readonly attributes
        enum {A, B, C, D}  grade;
instance attributes
        db_scores scores;

instance methods
        calc_new_db_grade () {...};
```

Assume that the following relation is defined

```
CREATE INDEX ON (std.name) std_rel (std: student*)
```

and populated with appropriate students. A user may now edit selected tuples from this relation by using a special EDIT query:

```
EDIT (std) INTO window1 FROM std_rel WHERE (std.gpa < 3.5)
```

This query acquires write locks on all the selected objects, retrieves them into a *window relation* (essentially an editor buffer) and displays the relation in a separate window supporting structure-editing. This window contains visual representations of the column attributes and methods of the selected columns and the instance attributes and methods of the selected objects. All information, except the key fields is initially elided, as shown below:

```
<std column ...>

name: Joe Doe
   <object...>
   <person...>
   <student..>
   <db_student..>

   ...

name: Henry Smith
    <object...>
    <person...>
    <student..>
```

In this display, the (non-key) attributes and methods declared in a class are grouped together. Thus the placeholder "<student...>" stands for all attributes and methods of the student that are declared in the class *student*.

A user may execute the *expand* editor command to display elided information. For instance, he may expand the placeholders "<std column...>", "<db_student...>" (in Joe Doe's entry), and "<person...>" (in Henry Smith's entry), to change the display to:

```
A_cutoff: 80  B_cutoff: 70  C_cutoff: 55  D_cutoff: 40

name: Joe Doe
    <object...>
    <person...>
    <student..>
    grade: B
    <scores...>
    calc_new_db_grade():

   ...

name: Henry Smith
    <object...>
    age: 21
    address: 101 Shaded Lane, Paradise, IN 47900
    e_mail: smith@paradise.cty
    tel_no: 743-1234
    <student..>
```

The user may now make Henry Smith a *db_student* by executing the *reclassify* command and choosing the new class from a menu of valid classes. The system responds with a template:

```
name: Henry Smith
  <object..>
  age: 21
  address: 101 Shaded Lane, Paradise, IN 47900
  e_mail: smith@paradise.edu
  tel_no: 743-1234
  <student..>
  grade: ?
      hw: <int>
      lab: <int>
      exam: <int>
  calc_new_db_grade():
```

containing placeholders for all the uninitialized modifiable attributes, which may be replaced with valid values.

O-Raid also provides commands to update the modifiable attributes of the displayed objects, invoke a method after filling its argument slots, delete an object to remove it from the database, fill a template to add a new object, commit the editing changes in the database, and erase the display and release write locks to the objects that were displayed.

We have illustrated above the default interface for editing objects. A user can customize this interface by specifying the values of *display attributes* of the objects displayed. These attributes are similar to Dost [10] attributes for displaying Mesa data structures, and determine the alignment, elided representation, prompt, etc of the displayed objects.

The notion of structure-editing has been mainly explored in the context of programming languages. Several researchers have argued, however, for using it as a general paradigm for interaction [9, 12, 20, 28]. We believe it is particularly suitable in our environment since we expect to support objects with complex structure and semantics.

The EDIT query for loading objects in a relation window has some similarity to a Postgres FETCH command loading a set of tuples in a portal. The difference between the two is that a relation window forms a buffer for a system-provided structure editor while a portal forms a buffer for some application program, which is responsible for providing a user interface for manipulating the data. We have decided to automatically provide such an interface because it is hard to generate manually, and can form, together with the query language, a standard default interface for interaction.

2.6. Integration of Structure Editing and Query Language

A user can modify an object in two ways: She can compose an appropriate SQL-like query or edit its visual representation. We support both interfaces because neither interface, individually, is suitable for all interaction. The query interface provides a way to (a) make a set of changes in "batch", (b) modify an object without going through the overhead of displaying it, and (c) select a set of objects for editing. On the other hand the editing interface provides an interactive and visual interface to make individual changes.

In O-Raid, we have "integrated" the two interfaces by allowing the query language to be used for manipulating window relations, which store the information buffered in an edit window. For instance, a

user can execute the query

```
UPDATE INTO window1 SET (std.scores.hw += 3, std calc_new_db_grade())
        WHERE (std.scores.hw <= 25);
```

to increment the *hw* fields of all the displayed students, and calculate and display their new grades. A this point, only the window relation is modified. The database is modified later when the *write* command is executed.

The query language can also be used to invoke arbitrary editing commands such as *elide*, and *expand*. For instance, a user can invoke the query

```
EXPAND (std.scores) FROM window1
        WHERE (std.grade == A)
```

to expand the *score* attribute of all students who have received the A grade.

A query language component of a structure-editor corresponds essentially to pattern-based component of a text editor. For instance, the UPDATE query corresponds to a "substitute-pattern" text editing command and the SELECT query corresponds to a "find pattern" command. Unlike, the pattern-based text editing language, the query language understands the structure and semantics of the objects being edited. For instance, it allows a user expand the contents of structured objects and send messages to them, as illustrated by the above examples.

2.7. The Display Query

Often a user wishes to continuously view a set of objects. For instance, a system administrator may want to continuously monitor the current status of bugs, and a manager may wish to continuously monitor the status of various projects. In the absence of a facility to support this task, a user needs to "poll" the system by continuously querying the state of objects. Therefore, in O-Raid we provide a DISPLAY query for continuously displaying the state of a set of objects in a display window. For instance a user may execute the query

```
DISPLAY (proj) INTO window2 FROM proj_rel WHERE (proj.status == incomplete)
```

to display all projects whose status is *incomplete*. The window is updated whenever a displayed object is changed. All editor commands and queries that do not update values or send messages can be invoked on the objects displayed in the window. Thus the user essentially invokes the structure editor in a "readonly" mode.

The DISPLAY query corresponds to an alerter in Postgres. The difference between the two is that the former continuously informs the user about changes to objects while the latter informs some application program about these changes, which can then communicate them to the user. We believe that the usefulness and generality of this feature justifies its inclusion in the set of default facilities, which needs to also contain an alerter-like mechanism for building application-specific displays.

3. IMPLEMENTATION OF THE O-RAID MODEL

We are implementing O-Raid by extending the Raid implementation. The Raid system is built on top of UNIX, and it runs on Sun workstations and VAXen. The complete system is implemented in 20K

lines of C code.

Overview of Raid

Figure 1 depicts the six major subsystems in Raid that reside on each site: *User Interface* (UI), *Action Driver* (AD), *Atomicity Controller* (AC), *Concurrency Controller* (CC), *Access Manager* (AM) and *Replication Controller* (RC). Each subsystem has been implemented as a server, and provides a very general interface. UI is a front-end invoked by a user to process queries on a database. It parses the queries and passes them to AD, which executes them and puts the updated data into a differential file. The transaction history, composed of timestamps of different actions, is sent to AC for validation. AC ensures global atomicity among all sites using a two-phase commit protocol. It sends the transaction history to all other ACs and its local CC for timestamp validation. CC provides different kinds of concurrency control methods, e.g. simple locking, read/write locking, timestamping, and conflict graph cycle detection. After the transaction is globally committed, the originating AD sends the differential file to AM's in other sites, which merge the differential file to the databases in a recoverable manner. RC is responsible for replication control. It allows continuing processing on an operational site while other sites are failing or recovering, or when the site's connections to a subset of sites is lost or restored due to a network partitioning. It uses the *read-one/write-all-available* strategy [3] for updating replicated copies of data.

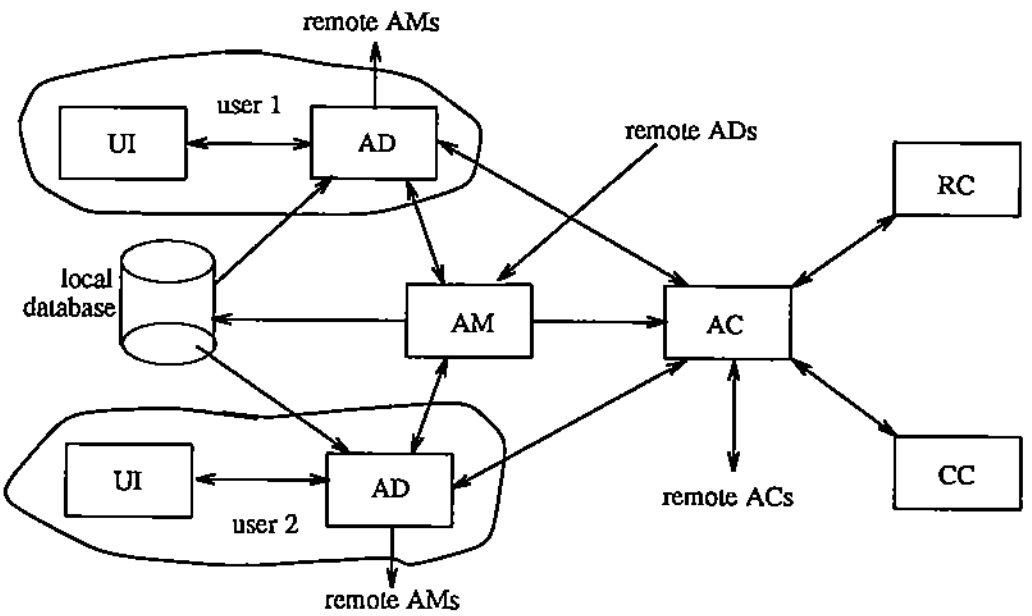

Figure 1: Raid Site Architecture

Implementation of O-Raid

To implement O-Raid, are only modifying UI, AD, and AM. Other subsystems are not affected at this stage and execute as in Raid. The granularity of concurrency control remains at the *tuple* level and the original method of assigning a unique *tuple identifier* based on *database identifier*, *relation identifier*, and *tuple logical address* is used. Transaction processing facilities of Raid remain the same; atomicity control, concurrency control, and replication control are also unchanged since the contents of the transaction history remain the same. The history contains read/write timestamps of tuples as before. Communication facilities also remain unchanged. Raid provides a *Long Datagram* (LDG) protocol which, can be used for communication of large objects. LDG is identical to the Arpanet UDP protocol except that it places no restriction on packet sizes. It has been built on top of UDP; each LDG packet is fragmented if necessary, and then sent using UDP. At the destination, fragments are collected and reassembled. The Raid message format is very general. It consists of a header composed of a message type, and sender address followed by a sequence of ASCII bytes describing the text of the message. Servers are free to interpret the message text in their own ways, and hence the message format is unchanged in O-Raid.

A new parser has been built to recognize the new query syntax. A precompiler is being developed to translate class declarations into C code and to produce tables describing the class schema. These tables are used by both the new UI and the new AD. The new UI supports the extended SQL query language and structure-editing interface. It maintains in memory hierarchical representations of objects displayed in the window, which, together with the class and relational schema descriptions, are used to support the structure-editing commands. As before, it passes the parsed query to AD, which is the heart of the system, and is organized as six components: *Query Processor* (QP), *Simple Query Processor* (SQP), *Database Access System* (DAS), *Object Manager* (OM), *Buffer System* (BS), and *UNIX File System* (FS).

OM is the major new component added to the existing AD. It is a collection of subroutines responsible for accessing objects and executing operations on them. It includes functions to return runtime addresses of methods, size of an object, offset of an object within a tuple, etc. QP and SQP process queries, which may invoke methods in objects. These methods are loaded if necessary from the database. A cache of most frequently methods is kept in virtual memory and methods defined in commonly used classes are preloaded. A *Class Control Block* table maps method names to their addresses.

DAS has been extended to provide more appropriate indexing methods including K-D-B-Trees. K-D-B-Tree indexing has been built to provide multi-key indexing which is useful for retrieving complex structures such as geometric objects For example, if we have a relation consisting of instances of *rectangles* we could use *lower_left_x*, *lower_left_y*, *upper_right_x*, and *upper_right_y* as indices to efficiently search for rectangles, as discussed in section 2.2. In this case, the K-D-B-Tree used to store the relation becomes a 4-D-B-Tree. All 4 attributes are utilized in the search algorithm and the tree is kept well balanced in all four dimensions.

In appendix A we give details and problems of extending AD to support the features of O-Raid.

4. FUTURE WORK

Planned Work in O-Raid

We plan to extend the design in several ways. Currently, we do not allow sharing of objects, or nested relations. We plan to support these features in the next version of O-Raid. We also plan to provide facilities to convert between database objects and the data structures of popular programming languages in order to reduce the "impedance mismatch" problem. Finally, we plan to pursue the concepts of class evolution, generating "friendly" textual displays for complex objects, supporting alerters and triggers, strategies for caching objects, classes, and method results, and support for Smalltalk-like metaclasses.

Our immediate goal is to complete the implementation of the current design. This will provide an estimate of the effort required to extend a relational model with object-oriented features. We will use the system to store geometric objects being developed as part of the CAPO (Computing About Physical Objects) project at Purdue, and Corporate Army databases containing geographical data. We also plan to use our implementation to test ideas about using the semantics of objects to increase concurrency during partitioning of a replicated database.

Future Directions for Research

Defining tuples and relations as special entities distinct from objects allows the use of a relational query language in an object-oriented world, but makes these non-objects "second-class citizens" that cannot be associated with user-defined protocols. This problem is reduced in O-Raid by making tuple fields and relation columns as first-class objects. A one-column relation is represented by its column object, and tuples in it are represented by their singleton fields. Nevertheless, it would be useful to provide relation and tuple protocols defining the behavior of multi-column relations. Providing this facility without sacrificing the relational query language requires a way of defining and efficiently creating relation classes at query-resolution time. The work being done in extending DSM is expected to shed some light on possible solutions to this problem.

The default query/structure-editing interface described here allows display and editing of only textual representations of objects. It would be useful if it could be extended to support graphical presentations of objects. One approach to support this facility is to let each object provide a description of its graphical/textual presentation, which is shown whenever the object is displayed in an edit or query window. A problem with this approach is that the editor cannot support modification of this presentation since it does not know the mapping between the values of an object and its presentations. For example, if an instance of *rectangle* displayed itself as a rectangle on the screen, then the editor would not know how to change the variables *lower_left_x*, *lower_left_y*, etc in response to editing of the rectangle. We are currently exploring a method that lets the object provide the system with a high-level description of the mapping between its presentations and its values, which is used by the editor to display and modify the object. An alternate approach is to let each object implement its editing interface, using perhaps a technique derived from the Smalltalk Model-View-Controller concept [27].

Finally, for the success of the object-relation model, it would be useful to build a programming environment around it that replaces or augments text files, hierarchical directories, command interpreters,

and text editors of current environments with relations of objects, hierarchical relations, query interpreters, and graphical/structure-editors respectively.

5. SUMMARY

An object-relation model offers benefits of object-oriented programming without sacrificing the facilities of the current relational model. In O-Raid we have engineered this model by allowing objects to be attributes of relations. Novel features of O-Raid include a hierarchy of column protocols, relations with heterogeneous objects that can be individually reclassified, facilities to reduce the data search for complex objects, a structure-editing interface integrated with a relational query language, and support for continuous display of objects. We are implementing O-Raid by extending the Raid implementation. The implementation will be used to support geometric and geographic databases, and to test algorithms that use semantics of objects to increase concurrency in a partitioned database. We plan to extend the design to support shared objects, integration with popular programming languages, class evolution, and other features that would increase its usefulness.

Further research is needed to define relation and tuple objects, a default structure-editing/graphical interface, and a programming environment based on the object-relation model.

REFERENCES

[1] G.T. Almes, A.P Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Overview," *IEEE Transactions on Software Engineering SE-11*, January 1985, pp. 43-59.

[2] Jay Banerjee, Won Kim, and Kyung-Chang Kim, "Queries in Object-Oriented Databases," *Proceedings of the Fourth International Conference on Data Engineering*, February 188, pp. 31-38.

[3] B. Bhargava and J. Riedl, "Implementation of Raid," *Seventh IEEE Symposium on Reliability in Distributed Systems, Columbus, Ohio*, October 1988 (to appear).

[4] Michael Blaha, William J. Premerlani, and James E. Rumbaugh, "Relational Database Design Using an Object-Oriented Methodology," *Comm. ACM* 31:4 (April 1988), pp. 414-427.

[5] Toby Bloom and Stanley B. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages," *OOPSLA '87 Proceedings*, October 1987, pp. 441-451.

[6] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Wolf, "Constraint Hierarchies," *OOPSLA '87 Proceedings*, October 1987, pp. 48-60.

[7] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg, "A Data Model and Query Language for EXODUS," *Proceedings of the SIGMOD International Conference on Management of Data*, June 1988, pp. 413-423.

[8] E. Codd, "A Relational Model for Large Shared Data Banks," *Comm. ACM* 13:6 (1970).

[9] Prasun Dewan and Marvin Solomon, "An Approach to Generalized Editing," *Proceedings of the IEEE Ist International Conference on Computer Workstations*, November 1985, pp. 52-60.

[10] Prasun Dewan and Marvin Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 (January 1987), pp. 150-159.

[11] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shain, "Iris: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems* 5:1 (January 1987), pp. 48-69.

[12] Christopher W. Fraser, "A Generalized Text Editor," *CACM* 23:3 (March 1980), pp. 154-158.

[13] Adele Goldberg and David Robinson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[14] Ted Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language," *OOPSLA '86 Proceedings*, September 1986, pp. 87-106.

[15] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrel Woelk, "Composite Object Support in an Object-Oriented Database System," *OOPSLA '87 Proceedings*, October 1987, pp. 118-125.

[16] Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the 9th POPL*, 1982, pp. 7-19.

[17] Davis Maier, Jacob Stein, Allen Otis, and Alan Purdy, "Development of an Object-Oriented DBMS," *OOPSLA '86 Proceedings*, September 1986, pp. 472-483.

[18] Thomas Merrow and Jane Laursen, "A Pragmatic System for Shared Persistent Objects," *OOPSLA '87 Proceedings*, October 1987, pp. 103-110.

[19] Norman Meyrowitz and Andries van Dam, "Interactive Editing Systems: Parts 1 and 2," *Computing Surveys* 14:3 (September 1982), pp. 321-416.

[20] David Notkin, "Sharing and Modularization in Structure Editing Environments," *Proceedings of the 19th Hawaii International Conference on Systems Sciences*, January 1986.

[21] D. Jason Penney and Jacob Stein, "Class Modification in GemStone Object-Oriented DBMS," *OOPSLA '87 Proceedings*, October 1987, pp. 111-117.

[22] R. Robinson, "The K-D-B-Tree: A Search Structure for Large Multi-Dimensional Indexes," *Proceedings of 1981 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, April 1981, pp. 10-18.

[23] Lawrence A. Rowe, "'Fill-in-the-Form' Programming," *Proceedings of VLDB*, 1985, pp. 394-404.

[24] Lawrence A. Rowe and Michael Stonebraker, "The POSTGRES Data Model," *Proc. 13th VLDB Conference*, 1987 .

[25] James Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language," *OOPSLA '87 Proceedings*, October 1987, pp. 466-481.

[26] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl, and R. Meyer, "OOPS - An Object Oriented Programming System with Integrated Data Management Facility," *Proceedings of the Fourth International Conference on Data Engineering*, February 1988, pp. 118-127.

[27] Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden, 1986.

[28] Jeffrey Scofield, "Editing as a Paradigm for User Interaction," Ph.D. Thesis and Technical Report No. 85-08-10, University of Washington, Department of Computer Science, August 1985.

[29] Karen E. Smith and Stanley B. Zdonik, "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems," *OOPSLA '87 Proceedings*, October 1987, pp. 452-465 .

[30] Eugene Spafford, "Architecture and Operation Invocation in the CLOUDS Kernel," Tech. Report No CSD-TR-730, Purdue University, December 1987.

[31] Michael Stonebraker and Lawrence A. Rowe, "The POSTGRES papers," Memorandom No. UCB/ERL M86/85, University of California, Berkeley, June 87.

[32] T. J. Teorey, D. Yang, and J.P. Fry, "A Logical Design Methodology for Relational Database Extended Entity-Relationship Model," *Acm Computing Surveys* 18:2 (June 1986).

[33] Peter Wegner, "Dimensions of Object-Based Language Design," *OOPSLA '87 Proceedings*, October 1987, pp. 168-182.

Appendix A: Implementation of O-Raid Action Driver (AD)

In this appendix we give details and problems of extending AD to support the object-relation model of O-Raid. *Action driver* (AD of Raid) has been modified to adapt the object/relation model. It contains six components - *Query Processor* (QP), *Simple Query Processor* (SQP), *Database Access System* (DAS), *Object Manager* (OM), *Buffer System* (BS), and *UNIX File System* (FS). OM is the only major new component added to the system, as discussed in section 4.

QP is responsible for global optimization of the query execution. Its strategies are: (1) perform selection and projection as early as possible (2) preprocess file appropriately (3) evaluate options before computing (4) look for common subexpression in an expression. After the preprocessing and substitution, QP calls SQP to retrieve/update tuples involving only *single relation*. Qualification part of a query is stored as a parsed tree format for efficient access. Target list of a query is stored as a linked list. Each query has a *query structure* which has pointers to target list and qualification part.

SQP receives the single-relation query from QP; then it looks up the qualification part of the query and decides the retrieving/ update method. The routine *SeleRetrMetd()* is used to select the access method. It takes the qualification part of query as an argument, and returns a *method identifier* (mid) which specify the access method to execute the query. SQP retrieves/updates tuples using the method and evaluates them against the qualification of the query. Also proper target list is built.

DAS provides different kinds of access methods for retrieving tuples. It provides hashing, ISAM, B-tree, and K-D-B-tree. In addition, it provides three general routines for accessing the methods. The routines are *StartScan()*, *Scan()*, and *EndScan()*. StartScan() sets up the subsequent scan of tuples. It takes *mid* as an argument and sets up the storage structures and information needed for the access method corresponding to that *mid*. Scan() does the actual search for tuples using the access method set up by StartScan(). EndScan() releases the storages allocated by StartScan().

BS is for managing the working buffer space. It is a collection of subroutines to perform different functions related to internal working space. It includes functions to allocate and release working space for relations, provide services to other components of AD such that contents of working relations always appear in the working buffer for them. A new storage structure of relations/objects is designed, and a new buffer system is written which allow objects to cross over physical pages. Flags are set in a page to indicate an object continuing in another page. Page size is currently set to 1024 bytes. Each working relation has a *inode* structure which has an array of pointers pointing to the working buffer pages. If the page is not in the main memory, the pointer points to *null*. The arrays are searched for releasing storage, if pages are required for other relations.

Two of the major problems of extending Raid to O-Raid are the structures of its QP and storage structure. In Raid, QP translates a query statement to several relational algebra statements (e.g. select, project, and cross-product). Join statement is translated to a cross- product statement followed by a select/project statement. In O-Raid, cross-product is avoided, and tuple substitution is used instead. Tuple substitution saves execution time and spaces. In Raid, storage structure of relations is organized as a fixed-size two dimensional table. It allocates fixed-size storage for each attribute and tuple. Also, there is no facility of allowing indexing in Raid. In O-Raid, each attribute could be a nested object; size of the attribute could be arbitrary large. A paging storage structure is designed. Also, indexing facility is provided in O-Raid (e.g. K-D-B-Tree). An access methods interface is built, so that it provides a general interface for different access methods. Since query processor and storage structures are the most important things in a DBMS, a lot of code has to be rewritten

Appendix B: Syntax of O-Raid Queries

This appendix summarizes the general syntax of basic queries. For detail discussion of the usage of each query the reader is referred to section 2 of the paper. The basic queries could be classified into four categories - data definition, data manipulation, EDIT and DISPLAY queries, and structure-editing queries.

(1) Data Definition Command - CREATE.

CREATE [index_method ON (attribute(s))] relation(s) (attribute_definition(s))

(2) Data Manipulation Commands - SELECT, INSERT, UPDATE, and DELETE.

SELECT (attribute(s)) [INTO relation(s)] FROM relation(s)
[WHERE predicate]

INSERT INTO relation(s) VALUES (expression(s));

INSERT INTO relation(s) SELECT ...

UPDATE INTO relation(s) SET (expression(s))
[WHERE predicate]

DELETE FROM relation(s)
[WHERE predicate]

(3) EDIT and DISPLAY queries

EDIT (attribute (s)) INTO window_relation FROM relation
[WHERE predicate]

DISPLAY (attribute (s)) INTO window_relation FROM relation
[WHERE predicate]

(4) Structure Editing Commands - ELIDE, EXPAND, etc.

command (attribute(s)) FROM relation [WHERE predicate]