SQL Extensions and Algebras for Object-Oriented Query Languages

Carolina Culik, Randall Mack and Dave Shewchun

Contents

1 Introduction **3**

2 Query Languages **3**
2.1 What is a Query Language? . . . 3
2.1.1 Purpose . . . 3
2.1.2 Characteristics . . . 3
2.1.3 Design . . . 4
2.2 RELOOP . . . 5
2.2.1 Syntax . . . 6
2.2.2 Accessing Nested Structures . . . 7
2.2.3 Constructing New Entities . . . 8
2.2.4 Nested Structures . . . 8
2.2.5 Some Additional RELOOP Facilities . . . 9
2.2.6 Aggregate Operators . . . 10
2.3 OSQL . . . 10
2.3.1 Syntax . . . 11
2.3.2 Referencing Objects Through Keys . . . 11
2.3.3 Using Functions in Queries . . . 12
2.3.4 Accessing Nested Data . . . 12
2.3.5 Aggregate Functions . . . 13

3 Algebras for Object-oriented Database Systems **13**
3.1 An Algebra for EXTRA/EXCESS . . . 13
3.1.1 The EXTRA Data Model . . . 13
3.1.2 The EXCESS Query Language . . . 14
3.1.3 The Algebra . . . 14
3.1.4 Operators . . . 14
3.1.5 Examples . . . 16
3.2 An Algebra for ENCORE . . . 16
3.2.1 The ENCORE Data Model . . . 16
3.2.2 Operators . . . 17

3.3 An Algebra for RELOOP . . . 19
3.3.1 RELOOP . . . 19
3.3.2 The O_2 Data Model . . . 19
3.3.3 Algebra Operators . . . 19
3.3.4 Example . . . 20
3.4 An Algebra for OODAPLEX . . . 21
3.4.1 OODAPLEX Data Model . . . 21
3.4.2 OODAPLEX Queries . . . 21
3.4.3 OOAlgebra . . . 22
3.4.4 OOAlgebra Operators . . . 22
3.4.5 Examples . . . 23
3.5 Conclusions . . . 24

4 Overall Conclusions **25**

1 Introduction

The query language used to access a database and the underlying algebra form one of the most important components of a database management system. We will discuss the issues involved in the design of a query language, examine several SQL-like query languages and review several possible algebras for the object-oriented model.

The emphasis in this paper is on practical, user-friendly models. Query languages which are attempts at extending SQL to the object-oriented model were chosen because of the large population of users already familiar with SQL, as well as its simplicity and formal basis.

2 Query Languages

In this section we introduce the concept of a query language, explaining some of the issues which affect its design and use. We analyze in depth the query languages RELOOP for the O_2 [CDLR89] object-oriented database management system (OODBMS) and OSQL [FAC$^+$89] for Iris OODBMS.

2.1 What is a Query Language?

2.1.1 Purpose

Query languages are used for four main purposes [BCD89].

1. for interactive queries
2. to access the database from a programming language
3. for simple programming tasks
4. to simplify complicated programming tasks

However, these four modes of use do not apply equally to all query languages. In general, object-oriented query languages are intended to merge smoothly with the programming language. This means that a programmer should be able to access the database directly from the host language, removing the need to use the query language for this purpose. On the other hand, traditional SQL embeddings only permit modification of the database through SQL queries and the lack of close ties between the two languages will preclude using SQL for programming shortcuts.

Despite the variety of uses to which a query language can be put, we will concentrate on its aptness for phrasing interactive queries. From the perspective of object-oriented query languages, that is certainly the most fundamental application. Any other purpose can be seen as more of a useful side-effect.

2.1.2 Characteristics

An *ad hoc* query language should have several characteristics: it should be possible to simply and concisely express reasonable queries; the run-time system should be able to find a good strategy for obtaining the result; and it should be independent of any specific application. As well, the interactive query language need not be as powerful as a complete programming/query language combination. Simplicity is more important.

2.1.3 Design

There are many issues involved in the design of a query language. A sample of some of the more critical ones might be [BCD89]:

- Should encapsulation be supported?
- What form will the answer take?
- Will the query language be integrated into the host language?
- Should queries be explicitly typed?

From the point of view of an *ad hoc* query language for an OODBMS, we can provide tentative answers to these questions.

Encapsulation

Encapsulation is the concept of packaging a set of methods with any given item of data, where only the methods will be visible to the user. Essentially this notion is just abstract data types as applied to an object-oriented language, and current wisdom says that the use of abstract data types is a good thing.

From this we can see that for queries buried in a programming language, encapsulation is a must. However, for interactive queries it is not so obvious. The main reason for encapsulation (to enforce modularity in programs) does not apply here. It is also easy to come up with examples where encapsulation would significantly hamper the user. For example [BCD89], a user does not want to query a stack through push and pop operations, he wants to be able to see the entire stack.

It appears that the best approach would be to enforce encapsulation for embedded queries, but relax this restriction at the interactive level.

Form of the Answer

There are essentially three options for how to represent/store the value returned from a query.

1. Queries can only return information in the form in which it currently exists in the database.
2. Query results belong to a meta-class which has pre-defined methods permitting displaying and printing.
3. A new class and methods permitting access to all of its fields are dynamically defined for every query result.

Option one is too restrictive for many purposes. Option three requires more overhead than most OODBMS's can spare. This leaves option two as the most reasonable compromise, but re-using the results can cause problems if encapsulation is enforced. We see a tradeoff between encapsulation and the power/efficiency of the language.

Embedding the Query Language

The most important thing to remember about this topic is that one of the major objectives in the design of OODBMS's is to remove the impedance mismatch found with relational systems. Assuming that problem is avoided, we have one remaining choice. Whether to make the query language a subset of the programming language, or merely ensure that it mixes well enough that calls from within the programming language are natural. While either is acceptable it would appear that making the query language a subset of the programming language would be preferable. Using this approach, there is no need for the programmer to learn two distinct languages. Learning the programming language is sufficient.

Typing

Similar to the discussion on encapsulation, we find that explicit typing is good for embedded queries, but bad for *ad hoc* queries. Explicitly typing embedded queries improves the correctness of a program. For an interactive user however, correctness is not quite as big an issue. Since the type can be inferred from the structure/class, we do not need to declare an explicit type with the query. This will increase the clarity and ease the creation of new, *ad hoc* queries.

2.2 RELOOP

The Altair research group in Rocquencourt, France, is currently implementing O_2 [CDLR89], an object-oriented database management system (OODBMS). O_2 supports two languages, each one adding data description and data manipulation to an existing programming language. These programming languages, when coupled with the database management facilities, solve the traditional impedance mismatch problem. However, they do not provide a very nice way in which to query a database.

Therefore, to compete with relational systems, the Altair group is implementing RELOOP, an interactive query language designed to allow users to specify reasonable queries easily and concisely.

RELOOP is being implemented on top of the O_2 OODBMS. O_2 supports all of the standard object-oriented features. Its support of complex objects is in the form of providing three minimal set constructors: sets, lists and tuples. All of these constructors can be applied to each other. For example, the user can create sets of sets, tuples with set valued fields, etc. O_2 also has one additional distinctive feature – its values. Values differ from objects in the following ways [BCD89]:

- Values do not obey object-oriented principles.
- Their structures are very similar to objects, but they are visible to the user whereas objects are encapsulated.
- Values have no identity.
- Values are manipulated by operators, not by methods.

For instance, the '.' operator allows for the extraction of a field from its tuple, like in SQL. Iteration is possible over a set value, and two list values can be concatenated.

A Sample O_2 Database Schema

Person = [name: [f-name: String, l-name: String],
age: Integer,
address: [city: String, zipcode: Integer]]

Woman = [name: [f-name:String, l-name:String],
age: Integer,
sex: String,
address: [city: String, zipcode: Integer]]
Woman is a subclass of person.

Man = [name: [f-name: String, l-name: String],
age: Integer,
sex: String,
address: [city: String, zipcode: Integer]]
Man is a subclass of person.

Family = [name: String,
mother: Woman,
father: Man,
children: {Person}]

All the fields of the above classes can be accessed through methods such as the following.

name: Person $\Rightarrow$ [first-name: String, last-name: String]
address: Person $\Rightarrow$ [city: String, zipcode: String]
age: Person $\Rightarrow$ Integer
is-child-of: Person $\times$ Person $\Rightarrow$ Boolean
spouse: Woman $\Rightarrow$ Man
spouse: Man $\Rightarrow$ Woman
name: Family $\Rightarrow$ String
mother: Family $\Rightarrow$ Woman
father: Family $\Rightarrow$ Man
children: Family $\Rightarrow$ {Person}

Note that these methods allow complete access to all data; thus, there is no reason to violate encapsulation.

The following is a presentation of the features of the RELOOP language [CDLR89]. All example queries access a sample database implemented on the above defined database schema.

2.2.1 Syntax

Since most potential users of the query language know SQL, the designers of RELOOP decided to make the language SQL-like. Consequently, RELOOP queries will feature the **select from where** block.

We begin with a simple query illustrating the syntactical differences between RELOOP and SQL.

Q1: What are the names of the minors?
select name(p)
from p **in** Person
where age(p) $<$ 18

Basically, the query can be read and understood much like an SQL query. The **select** clause defines the result (the names of some persons). The **from** clause says where to get the information. Note that in this case, “Person” is a special set value, so it is legal to define the variable p as ranging over the set and representing one of its elements. The **where** clause specifies the conditions that must be satisfied.

The first obvious difference from SQL is the specification of “name(p)” instead of “p.name” in the **select** clause. The designers believe that the functional notation is easier to read. Another advantage of this notation is that it is used both for operators and methods, so inexperienced users do not need to understand the difference between objects and values.
For instance, to get the first name of a person, we first have to apply the “name” method on a “Person” object, and then use the field extract operator to obtain the result. Clearly, the expression first-name(name(p)) is easier to read and use than the expression name(p).first-name.

Second, RELOOP introduces a new keyword, “**in**” in the **from** clause. Any expression which results in a set value can be used in the **from** clause. Therefore, in the case of complex expressions, a separator between the variable name and its domain makes the query easier to read.

Finally, a **select from where** block result will always be a set value. This results from the fact that users are usually trying to retrieve a collection of database entities. However, if the user knows that they are extracting a single item from the database, they may use the RELOOP operator “extract” to get the unique item in a set.

2.2.2 Accessing Nested Structures

The following query shows how to access data in a nested structure.

Q2: Find the Families where the husband lives in Waterloo.
select f
from f **in** Family
where city(address(husband(f))) = ”Waterloo”

The expression in the **where** clause can be seen as a path within the object composition hierarchy. For each family, we apply to it the husband method which returns a Man object. We then apply the address method to the Man object, and then finally the city operator to extract the city field from the address. We can access any data in the database using some such expression.
The result of the query is a set of database objects. The next section will illustrate how the user can create new entities in the **select** clause.

2.2.3 Constructing New Entities

The following query shows how to construct tuple values.

Q3: Find the first name and age of the minors.
select [name: f-name(name(p)), age: age(p)]
from p **in** Person
where age(p) $<$ 18

The tuple operator [ ] allows the construction of tuple values. It takes as arguments field names and their descriptions. A description can be made through any valid RELOOP expression. Therefore, as we will see later, this feature allows us to build nested structures.

As mentioned earlier, the result of a query is always a set value. Therefore, since the above select clause constructs tuples, the final answer to the query will be of the following type: {[name: String, age: Integer]}

2.2.4 Nested Structures

In this section we will illustrate both how to create nested structures and how to flatten nested ones.

The first query shows how to construct a flat structure from a nested one.

Q4: Find the mother of every child.
select [child: child, mother: mother(f)]
from f **in** Family
child **in** children(f)

An interesting observation is that it is valid to define a variable domain in the **from** clause through the application of a method on a previously defined variable. In fact, any RELOOP expression which returns a set value can be used for this purpose. This feature allows the user access to all levels of a nested structure, and thus, allows easy unnesting.

The last two queries show, conversely, how to build nested structures from flat ones.

Q5: Associate with each woman her minor children's first names, assuming that a woman may appear in different families.
make-method minors (woman: mom)
select f-name(name(child))
from f **in** Family
child **in** children(f)
where age(child) $<$ 18 **and**
mother(f) = mom

select [mother: mom, children: minors(mom)]
from mom **in** Woman

The sets of minor children that we wish to construct are not database entities, therefore, they must be constructed. One way in which we can accomplish this is to create a method on the Woman class which will collect the appropriate set of children. The above method examines every child in every family and picks those that are under 18 years of age and whose mother is the method's argument. Once the method is complete, we only need to build a simple query that creates a tuple result from which we call the method. Therefore, the type of the final result is {[mother: Woman, children: {String}]}

In this example, the nesting was done on an object. However, often we work with values instead of objects. Since we cannot use methods on values, we must find another way of formulating the above query.

The following is an alternate RELOOP expression for the above query which can deal with both objects and values.

```
select [mother: mom, children:
    select   f-name( name(child))
    from     f in Family
             child in children(f)
    where    age(child) < 18 and
             mother(f) = mom]
from mom in Woman
```

The only difference between the two formulations is that in the latter we find the body of the method where the method was previously called. It is unclear from the literature written on RELOOP whether recursive calls to methods are permitted.

2.2.5 Some Additional RELOOP Facilities

We conclude our discussion of RELOOP by introducing two of the language's facilities, the **collapse** operator, and the set constructor. The **collapse** operator flattens a set of sets.

For example, **collapse** ({{1,2}, {2,3}}) = {1,2,3}

The set constructor {} builds a set value from its arguments.

The following two queries illustrate the uses of these facilities.

Q6: Find the people that have at least one child.

```
        select   mother(f)
        from     f in Family
        where    children(f) ≠ {}
union
        select   father(f)
        from     f in Family
        where    children(f) ≠ {}
```

This query first finds all the women and then all the men that have at least one child, and then it merges the results. The union of the results is possible because women and men are both persons.

Using the **collapse** operator, we can also use a different query formulation to obtain the same result.

```
collapse    (select    {mother(f), father(f) }
             from      f in Family
             where     children(f) ≠ {})
```

In this query, for each family with more than one child, a set containing a mother and a father is constructed. Therefore, the result of this query is a set of sets. The **collapse** operator flattens the result into just one set.

An interesting observation is that this query does not use an instruction such as "distinct" to avoid duplicate values. This is because of the fact that we are manipulating sets in the **select** clause, and thus duplicates (and order) have no meaning.

2.2.6 Aggregate Operators

Aggregate operators, like those we are familiar with in SQL
(sum, avg, ...), are not yet implemented, but will be available in the final version of RELOOP.

2.3 OSQL

Hewlett-Packard Laboratories are currently developing the Iris [FAC+89] object-oriented database management system (OODBMS). Iris is a research prototype of a next generation DBMS, intended to meet the needs of new and emerging applications.

Currently, one of the three interactive interfaces that is provided by Iris is Object SQL (OSQL) [FAC+89]. OSQL is an object-oriented extension of SQL.

A Sample OSQL Database Schema

```
Person = (first-name Charstring required,
          last-name Charstring required,
          age Integer,
          city Charstring),

Employee = (first-name Charstring required,
          last-name Charstring required,
          age Integer,
          city Charstring,
          company Charstring,
          salary Integer),
```

Employee is a subclass of person.

All the fields in the above classes can be accessed through property functions such as the following.

first-name: Person ⇒ Charstring
last-name: Person ⇒ Charstring
age: Person ⇒ Integer
city: Person ⇒ Charstring
company: Employee ⇒ Charstring
salary: Employee ⇒ Integer

Note that the above property functions allow complete access to all data; thus, there is no reason to violate encapsulation.

The following is a presentation of the features provided by OSQL [FAC+89]. Example queries access a sample database implemented on the above database schema.

2.3.1 Syntax

OSQL basically adheres to the SQL query format, but its syntax differs more from SQL than that of RELOOP. For example, consider the following simple query:

Q1: What are the first names of the minors ?
Select **distinct** first-name(p)
for each Person p
where age(p) < 18;

An obvious difference is that the **from** keyword has been replaced by **for each**.

We will see further differences in OSQL syntax in subsequent sections, especially shortcuts we can make in a query formulation when calling functions from the query.

OSQL does provide SQL's "distinct" operator which eliminates duplicates, while RELOOP does not. (RELOOP **select** statements automatically eliminate duplicates).

2.3.2 Referencing Objects Through Keys

Objects may be referenced directly rather than indirectly, through their keys. Interface variables may be bound to objects on creation or retrieval and may then be used to refer to the objects in subsequent statements.
The following OSQL expression creates an instance of the person object.

Create Person (first-name, last-name, age, city) **instance**
p1 (John, Doe, 25, Waterloo);

Q2: Find the age of the person with identifier p1.
Select age(p1);

First we create a new person and then give this person the unique identifier p1. We can then use the identifier when we want to refer to its object as in the above **select** clause.

Notice that no such feature is provided in RELOOP.

2.3.3 Using Functions in Queries

User-defined functions and Iris system functions may appear in **where** and **select** clauses to give concise and powerful retrieval.

For example, let us define the function Marriage as,

Create function Marriage(Person p) ⇒
<Person spouse, Charstring date> **forward**;

(**forward** specifies that the function implementation will be specified later.)

This function returns a compound result – the spouse and the date of the marriage.

We have already seen the use of a function in the select clause in the previous section. The following example shows how a function is invoked in the **where** clause.

Q3: Find Bob's wife and the date on which they where married.
Select s, d
for each Person s, Charstring d
where <s, d> = Marriage(Bob);

Here we are comparing every combination of a Person object and a character string to the result of the Marriage function when applied to the Person Object "Bob". Note that the above example demonstrates that OSQL permits queries to range over infinite domains.

The above query can be abbreviated to:

Select each Person s, Charstring d
where <s, d> = Marriage(Bob);

Or even more simply to:

Select Marriage(Bob);

The last two query formulations stray from the SQL **select from where** block. The first abbreviation combines the two keyword phrases **Select** and **for each** into one "**Select each**" phrase. The second abbreviation does not explicitly specify the type of its result, but implies that it will be of the form of the result of the Marriage function.

Notice that these types of syntactical shortcuts are not possible in RELOOP.

2.3.4 Accessing Nested Data

As in RELOOP, the expression in the **where** clause can be seen as a path within the object composition hierarchy. All data in the database can be accessed using an expression such as the following:

Q4: Find all families where the husband lives in Waterloo.
Select f
for each Family f
where city(husband (f)) = "Waterloo"

2.3.5 Aggregate Functions

Several aggregate functions have already been implemented in OSQL. In previous sections we saw functions defined on individual objects. There is also a need for functions over multi-sets (sets containing duplicates).

The following query illustrates the use of the function "average".

Q5: Find the average salary of all employees.
Select y
for each Real y
where y = Average (**select** Salary(x) **for each** Employee x);

The nested **select** function returns a mult-set of values which is used as input to the Average function.

Iris semantics are such that nested **select** statements and function calls are equivalent in retrievals. Thus, given a function, EmpSal(), that returns the salaries of all employees, the above query could be rewritten by replacing the nested **select** statement with a call to Empsal.

As mentioned earlier, aggregate operators have not yet been implemented in RELOOP but will be available in the final version. It will be interesting to see how a query such as the above will be implemented since the RELOOP **select** statement eliminates duplicates.

3 Algebras for Object-oriented Database Systems

One of the strong points for the relational database model is the existence of a formal algebra for query processing and optimization. Attempts are now being made to develop a similar algebra for object-oriented database systems. This section will examine four such algebras: an algebra for the EXTRA DDL and EXCESS DML [VD91], an algebra for the ENCORE data model [SZ], an algebra for RELOOP [CDLR89], and OOAlgebra, an algebra for OODAPLEX [Day].

3.1 An Algebra for EXTRA/EXCESS

3.1.1 The EXTRA Data Model

In the EXTRA data model [VD91] complex objects are defined as complex structures. These structures are built from tuple, multiset, and array type constructors. Multisets are sets with duplicates. Arrays are one-dimensional only, and of variable length. Methods can also be defined for a structure as a sequence of EXCESS statements. Object identity is supported with the **ref** keyword which specifies a reference to an extant object (ie: an OID). In this way, **ref** can actually be viewed as another type constructor. Inheritance is handled by allowing one structure to inherit another, which causes all attributes and methods of the structure being inherited to become attributes and methods in the new structure. Any inherited attribute or method can be overridden in the new structure definition. Multiple inheritance is also allowed.

3.1.2 The EXCESS Query Language

The EXCESS query language allows for both querying and updating of the database. Any structure defined in EXTRA can be fully manipulated by EXCESS. EXCESS can also be extended with user-defined functions and operators written in the E language.

As an example, consider the following EXCESS query which 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
```

Here, **Employees** has been created in EXTRA to be a set of **Employee** structures (objects). The "." notation is used to access the properties of a complex structure.

3.1.3 The Algebra

The EXCESS/EXTRA algebra is formally defined as a set of n-ary operators that operate on a set of objects, and are closed with respect to this set of objects. Elements of this set of objects are called "structures". A *structure* contains two components, a *schema* and an *instance*. A schema is a digraph with the nodes representing one of the four type constructors (multisets, tuples, arrays, or refs) or a simple scalar value, and the edges representing a "component of" relationship. So, a schema is just a digraph representation of a structure defined in the EXESS DDL. Since the algebra operates on different "sorts" (different type constructors), it is referred to as a "multi-sorted" algebra.

3.1.4 Operators

The EXTRA/EXCESS algebra contains a large number of operators to handle the various features of the EXCESS query language. Instead of having all operators work on sets, as is done in many algebras, different operators are defined for each of the four "sorts" within the algebra.

Multiset Operators

There are eight operators for multisets:

1. **Additive Union**: Combines two multisets and does not eliminate duplicates.
2. **Set Creation**: Creates a new multiset with a single member.
3. **Set Apply**: Apply an algebraic expression to all members of the multiset, replacing them with the result of the expression.
4. **Grouping**: Partition a multiset based an algebraic expression.
5. **Duplicate Elimination**: Convert a multiset to a regular set.
6. **Difference**: Subtract the cardinality of set members to determine the result cardinality.
7. **Cartesian Product**: The regular set-theory Cartesian product.

8. **Set Collapse**: Takes a multiset of multisets and returns the union of the member multisets.

Tuple Operators

There are four operators for tuples:

1. **Projection**: The same as the relational projection except it works only on a single tuple.
2. **Concatenation**: Concatenates two tuples.
3. **Field Extraction**: Extracts a single field from a tuple. Unlike projection, the result is the field, not a 1-tuple.
4. **Creation**: Create a new 1-tuple.

Array Operators

There are nine operators for tuples:

1. **Creation**: Create a single element array.
2. **Array Extract**: Extract a single element from an array.
3. **Subarray**: Extract a range of elements from an array and place them in a new array.
4. **Array Concatenation**
5. **Array Apply**
6. **Array Collapse**
7. **Array Difference**
8. **Duplicate Elimination**
9. **Cartesian Product**

The last six operators are the same as the corresponding set operators except that they preserve order.

OID Operators

Recall that EXTRA/EXCESS OIDs are just references to objects which exist in the database independent of the objects which reference them. There are two operators for OIDs:

1. **Dereference**: Dereference an OID node within a schema. The OID node is replaced by what it referenced.
2. **Reference**: This operator creates an OID for its operand and then inserts this new OID into a schema.

Predicates

There is one additional operator within the algebra to handle predicates called COMP. A predicate is specified with the COMP operator and it returns its input exactly if the predicate is true, or nothing if the predicate is false. Predicates may only appear in this COMP operator.

retrieve (TopTen[5].name, TopTen[5].salary)

$$\pi_{name,salary}(DEREF(ARR_EXTRACT_5(TopTen)))$$

Figure 1: EXCESS Query and Algebraic Representation

3.1.5 Examples

The EXTRA/EXCESS algebra will now be illustrated with two examples. For these examples a database is being used that contains a set of **Student** objects named **Students**, a set of **Department** objects named **Departments**, a set of **Employee** objects named **Employees**, and a fixed length array of size ten of **Employee** objects called **TopTen**. All of these sets and the array contain references to objects, rather than the objects themselves. The **Student**, **Employee**, and **Department** objects are all tuples.

Figure 1 shows the first example. It contains an EXCESS query that returns the name and salary of the 5th element of the **TopTen** array. Bellow the query is the corresponding algebraic representation.

In this example **ARR_EXTRACT** is the *array extract* array operator, **DEREF** is the *dereference* OID operator, and π is the *projection* tuple operator. This expression extracts the 5th element of the **TopTen** array with the array extract operator, dereferences it to obtain the actual **Employee** tuple, and then uses projection to obtain the name and salary.

Figure 2 gives a second example. Here the query retrieves the names of the departments of all employees who work in Madison. To simplify the presentation of the algebraic representation, a graphical notation is used. In this notation an arc from A to B is used in place of B(A).

In this example **SET_APPLY** has been abbreviated as **S_A**, and
TUP_EXTRACT is the *tuple extract* tuple operator. The first expression in this example (bottom most) converts the multiset of **Employee** references to a multiset of actual bf Employee tuples. The second expression selects all tuples such that the employee works in Madison. The third expression dereferences the "dept" attribute of the employee tuples and replaces it with the actual "dept" value. The final expression preforms a projection on the "name" attribute to obtain the desired result (department names).

3.2 An Algebra for ENCORE

3.2.1 The ENCORE Data Model

A type in ENCORE [SZ] is an abstract data type. Each type has a name (which is used for type equivalence), a set of properties, and a set of operations. Operations are functions that can be applied to instances of the type. A property is a typed object that can be implemented as an actual value, or as a procedure or function that returns an object of the correct type.

Multiple inheritance is supported by allowing the specification of a set of supertypes in the definition of a type. A type inherits all of the properties and operations of its supertypes. Properties

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

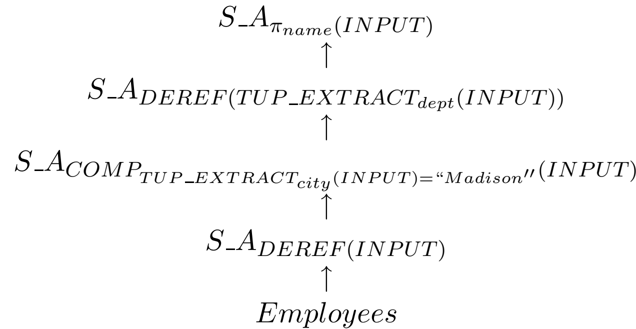

Figure 2: EXCESS Query and Algebraic Representation

cannot be overridden in inheritance. However, operators can be overridden as long as the operator for the new type has the same argument and output types as the operator from the superclass. All types in encore are subtypes of the type **Type**.

ECORE also supports parameterized types, with the tuple and set parameterized types built in.

The objects in ENCORE are simply instances of the type **Object** or subtypes of the type **Object**. Type **Object** defines some basic operations for objects, such as *comparison* and *type_of*. All objects in ENCORE have a unique identity.

It should be noted that sets and tuples are objects within ENCORE since they are instances of the set and tuple parameterized types. The set and tuple types are subtypes of the top level type **Object**.

3.2.2 Operators

The ENCORE algebra has three groups of operators, the basic operators, the set operators, and the object identity operators. Unlike EXTRA/EXCESS all of the operators in ENCORE operate on collections of objects. Also, while there are twelve operators in total, the group of four basic operators are the main ones that would be handled by the query optimizer, compared to the large number that the optimizer for EXTRA/EXCESS must deal with.

The group of identity operators in ENCORE is of special note. Queries in ENCORE may produce new objects that did not previously appear in the database. These objects will have their own unique object identities. It is possible that a new group of objects could be created when only a single object is needed. That is, the objects have different identities but are shallow equal. To handle this, the concept of *i-equality* is introduced, where *i* indicates how "deep" a comparison

should be made. The identity operators make use of this *i-equality*.

The Basic Operators

There are four basic operators defined in the algebra. For the following definitions, S and R are collections of objects, p is a first-order Boolean predicate, the A_i are attribute names having type T_i, and the f_i are functions returning objects of type T_i.

1. $Select(S, p) = \{s \mid (s \ in \ S) \wedge p(s)\}$

   Select takes a set of objects and returns only those that satisfy the predicate.

2. $Image(S, f) = \{f(s) \mid s \ in \ S\}$

   Image is similar to the apply operators of EXTRA/EXCESS. It applies a function to each object in the given collection.

3. $Project(S, < (A_1, f_1), \ldots, (A_n, f_n) >) =$

   $\{< A_1 : f_1(s), \ldots A_n : f_n(s) > \ \mid s \ in \ S\}$

   Project is an extension of *Image* and allows more than one function to be applied to the objects. For each object in S a tuple is returned which contains the result of applying each of the functions (f_i) to the object. Note that this operator actually generates new objects (tuples) that become part of the database.

4. $Ojoin(S, R, A, B, p) = \{< A : s, B : r > \ \mid s \ in \ S \wedge r \ in \ R \wedge p(s, r)\}$

   Ojoin is used to create new relationships in the database. For each pair (s, r) where s is in the collection of objects S and r is in the collection of objects R, it returns a tuple containing s and r if they satisfy the specified predicate. In other words, it generates a set of tuples containing two objects that have a particular relationship. As with *Project* these tuples are new objects that become part of the database.

Set Operators

The algebra contains the following set operators:

1. *Union, Difference, Intersection*

   These are the standard set operators where set membership is determined by object identity.

2. *Flatten*

   This operator takes an object that is of type set of sets and returns an object that is of type set.

3. *Nest, UnNest*

   These operators perform nesting and unnesting of tuples on a single attribute. This is best illustrated with an example. The UnNest operator on B on the set $\{< A : o_1, B : s_1 >\}$, where s_1 is a a set containing o_1 and o_2, would produce the set of tuples $\{< A : o_1, B : o_2 >, < A : o_1, B : o_2 >\}$. The Nest operator is essentially the inverse operation.

Identity Operators

The algebra contains two identity operators:

1. $DupEliminate(S, i)$

   This operator keeps only one copy of i-equal objects from a collection of objects.

2. $Coalesce(S, A_k, i)$

   For a collection of tuple objects, this operator eliminates i-equal duplication in the A_k components (attributes) of the tuples.

3.3 An Algebra for RELOOP

3.3.1 RELOOP

Since the RELOOP query language has already been discussed in this paper, no further mention of it will be made here. Instead, the reader is directed to the appropriate sections earlier in this paper.

3.3.2 The O_2 Data Model

The O_2 data model [CDLR89] consists of a set of classes. Associated with each class is a type expression. A type expression is recursively defined and contains atomic type expressions (integer, real, etc), class names, tuple type expressions, and set type expressions. Note that a class name can appear in a type expression, and this is how inheritance is achieved.

An object in O_2 is a value corresponding to some type expression (class). Associated with each object is a unique object identifier. For each class, there is an assignment function that returns the set of object identifiers for the objects that are members of that class.

O_2 also allows for the definition of methods.

3.3.3 Algebra Operators

There are only a small number of operators in the RELOOP algebra, and all of these operate on sets of tuples (relations). As with OOAlgebra (described in a later section), most of these operators are the same as those found in the relational algebra. The operators are presented formally below. In the following a_i is an attribute name, v_i is a value, and r or r_i is a relation (set of tuples).

1. **Set Operators**

   The algebra supports the standard set operators union, intersection, and difference.

2. **Projection**

   The projection operator in the RELOOP algebra is the same as that in the relational algebra. It removes all but a specified set of attributes from all tuples in a set.

3. **Selection**

   The selection operator is similar to that from relational algebra. Given a boolean predicate and a set of tuples, it returns the set of tuples for which the predicate is true. The predicate expression has as its argument a set of tuple values, and has the form $E_1 \theta E_2$, where E_1 and E_2 are expressions and θ is one of the traditional comparators. Note that a predicate cannot use the boolean operators.

4. **Cartesian Product**

   The Cartesian product operator is similar to that from relational algebra, but behaves slightly differently. A list of attribute names must be given, along with a corresponding list of sets of tuples. The result is a set of tuples containing the cross product of the tuples from the given sets, using the given attribute names. Formally this is specified as:

   $\otimes < a_1, \ldots, a_n > (r_1, \ldots, r_n) =$

   $\{[a_1 : v_1, \ldots, a_n : v_n] \mid \forall i \in [1, n], (v_i \in r_i)\}$

5. **Reduction**

   The reduction operator is used to remove the values from tuples, discarding the attribute names. The result of the reduction operator on a set of tuples is the set of values from the tuples. Formally this is:

   $\triangle(r) = \{v \mid [a : v] \in r\}$

3.3.4 Example

To illustrate the RELOOP algebra an example will now be given. For this example a database with the following properties is used:

- Person = $\{i_1, i_2, i_3\}$ where i_1, i_2, i_3 are object identifiers.
- age(i_1) = 2, age(i_2) = 18 and age(i_3) = 19
- first-name(name(i_1)) = "John",
  first-name(name(i_2)) = "Joan", and
  first-name(name(i_3)) = "Jack".
- last-name(name(i_1)) = "Doe",
  last-name(name(i_2)) = "Doe", and
  last-name(name(i_3)) = "Smith".

Figure 3 gives a query to find the first name and age of all of the adults in the database described above. Below the query is the corresponding algebraic representation.

The first expression in this example will produce the set of all possible tuples with the type {[p:Person, name:String, age:Integer]}. The second expression is a selection operation on the given condition and will produce the following tuples:

{[p:i_2, name:Joan, age:18], [p:i_3, name:Jack, age:19]}

The final expression is a projection on name and age and will produce:

{[name:Joan, age:18], [name:Jack, age:19]}

select [name: first-name(name(p)), age: age(p))]
from p **in** Person
where age(p) ¿= 18

$R_1 \leftarrow \otimes$ <p, name, age>(dom(elt(type(Person))),
dom(type(first-name(name(p)))), dom(type(age(p))))
$R_2 \leftarrow \sigma$ <age(p) >= 18$\wedge$ name = first-name(name(p)) $\wedge$ age = age(p)
$\wedge$ p $\in$ Person> (R_1)
$R_3 \leftarrow \pi$ <name, age> (R_2)

Figure 3: RELOOP Query and Algebraic Representation

3.4 An Algebra for OODAPLEX

3.4.1 OODAPLEX Data Model

The OODAPLEX data model [Day] is based on *types*, *entities*, and *functions*. An entity is simply an object in the database and each entity has its own unique identity. Each entity is of a particular type. The definition of a type includes the name of the type and the signatures of its functions (described below). A type also contains an *extent* which gives the set of instances (entities) of the type in the database.

OODAPLEX makes a distinction between types for which instances are not actually manipulated (called *immutable types*), such as integers, reals, and booleans, and types for which instances must be explicitly manipulated by users (called *mutable types*). The aggregate types sets, multisets, and tuples are also supported.

Properties of entities are modeled with functions. A function can accept zero or more input arguments, and produce zero or more output arguments, and the type for all arguments must be specified. The *signature* of a function is a specification of the function name, and its input and output arguments. A function is considered to be *attached* to all types which are mentioned in the function's input arguments.

3.4.2 OODAPLEX Queries

A query in OODAPLEX is simply a function that accepts a set of input arguments, and returns a set of output arguments. These functions effectively map one collection of entities to another collection of entities. Note that the set of input and output arguments is not necessarily given explicitly, but may be deduced from the body of the query (function).

The body of an OODAPLEX query is a set of expressions. These expressions contain variables, constants, function names, and predicates. Looping constructs also exist to handle sets, multisets, and tuples.

OODAPLEX also supports database updates by allowing for creation and destruction of entities,

and modification of sets, tuples, and functions.

3.4.3 OOAlgebra

The algebra for OODAPLEX is named OOAlgebra. It is based on, and is very similar to, the PDM algebra. OOAlgebra is based on the traditional relational algebra, and has many similar operators.

OOAlgebra operates on sets, multisets, and tuples. Here, sets are treated as sets of tuples, and can also be referred to as relations.

3.4.4 OOAlgebra Operators

The algebra contains two groups of operators, one for tuples and one for sets. These two groups are described below. In addition to these operators, a large number of aggregate operators such as count, sum, and average are included in the algebra.

Tuple Operators

The following tuple operators exist in the algebra:

1. $rename(T, L)$

   Here, T is a tuple, and L is a list of pairs of labels in the form $[Lin_1 : Lout_1, \ldots Lin_n, Lout_n]$. The rename operator returns a tuple the same as T but with label Lin_i changed to $Lout_i$.

2. $project(T, [L_1, \ldots, L_n])$

   Project returns a tuple containing only the components from T that are labeled with one of the L_i's.

3. $select(T, B)$

   Here, T is a tuple and B is a boolean-valued function. This operator returns T if $B(T)$ is *true*, or *null* otherwise.

4. $product(T_1, T_2)$

   This operator produces a tuple that is the concatenation of T_1 and T_2.

5. *apply_append*

   This operator is similar to the apply operators in other algebras in that it applies a function over each tuple of input arguments. However, unlike the apply in other algebras, apply_append does not replace the tuples with the function result, but instead appends the output of the function to the input of the operator.

Set Operators

In addition to the basic set operators, union, intersection, and difference, the following set operators also exist in the algebra:

1. $rename(S, L)$

   Here, S is a set of tuples. This is the same as the rename for tuples except that the rename is performed on each member of the set.

major(the s in extent(student) where ssno(s) = "123456789")

T1 := rename(ssno(P:person, N:string)[P:S])
T2 := apply_append(extent(student)(S:student), T1(S:person, N:string))
T3 := select_tuple(T2(S:student, N:string), equals(N, "123456789"))
T4 := apply_append(T3(S:student, N:string), major(S:student, D:department))
T5 := project(T4(S:student, N:string, D:department), [D])

Figure 4: OODAPLEX Query and OOAlgebra Representation

2. $project(S, [L_1, \ldots, L_n])$

   This is the same as project for tuples, except that the projection is applied to each member of the set.

3. $remove_duplicates(S)$

   This operator removes duplicates from a multiset.

4. $select(S, B)$

   This is similar to the tuple select operator. The result of the operator is the set of tuples from S for which the the boolean expression evaluates to *true*.

5. $select_tuple(S, B)$

   This operator first performs a select on S. If the result is a singleton set, then the single tuple from the set is returned. Otherwise, an error condition is raised.

6. $product(S_1, S_2)$

   The result of this operator is a set containing each member of S_1 concatenated with each member of T_2.

3.4.5 Examples

The OOAlgebra will now be illustrated through two examples.

Figure 4 gives an OODAPLEX query which returns the department entity for the department the student with social security number "123456789" is majoring in. Below the query is the corresponding OOAlgebra representation.

Figure 5 gives a second example. The query in this example returns the name and age of all people in the database. The result is returned as a multiset of name, age pairs.

```
for each p in extent(person)
    [name(p), age(p)]
end

T1 := apply_append(extent(person)(P:person), name(P:person, N:string))
T2 := apply_append(T1(P:person, N:string), age(P:person, A:integer))
T3 := project(T2(P:person, N:string, A:integer), [N, A])
```

Figure 5: OODAPLEX Query and OOAlgebra Representation

3.5 Conclusions

Each of the four algebras presented here has a unique approach to creating an algebra for an object-oriented database. Two of the algebras, RELOOP and OOALGEBRA are based on the original relational algebra. They take the path of extendeding and modifying the relational algebra for the object-oriented world. Both contain a fairly small number of operators that are closely related to those found in the relational algebra. While the RELOOP algebra operates only on sets of tuples, the OOALGEBRA operators operate on both sets and individual tuples.

The other two algebras, EXTRA/EXCESS and ENCORE have taken the approach of starting fresh, and not simply modifying the relational algebra. Of these two, EXTRA/EXCESS has the most novel approach. It contains a large number of operators for the different "sorts" supported. Object identity is handled by two special operators, **reference** and **dereference**. By comparison, ENCORE also has a fairly large number of operators, except that all of them operate on collections of objects. ENCORE also introduces the concept of i-equality to help solve some of the problems associted with object identity.

One key question about any algebra is how many operators are required. If care is not taken in choosing the operators, the algebra may contain a large number of operators that are never used by the query optimizer. Or, it may lack some operators that would make query optimization much easier. The two relational based algebras presented here have taken the "safe" approach. They make the assumption that the existing relational algebra contains the correct set of operators, and only make changes as required by the new environment. Of the other two algebras, EXTRA/EXCESS has the most variety in operators. However, the authors claim that because of the multi-sorted nature of the algebra, only certain operators are applicable at certain times. This, operator choice is greatly simplified, and all operators will be used. While the ENCORE also contains a large number of operators, there is only a small set of basic operators which would be used by the optimizer, and these are in some ways similar to the relational operators. The additional operators have been added only to address certain problems specific to the algebra.

4 Overall Conclusions

The query language used to access a database and the underlying algebra form one of the most important components of a database management system. Extending SQL into a query language for OODBMS's would both make it easier for the many relational database users to move to the more powerful object-oriented paradigm, and would provide a uniform standard across the database field. Although we have examined several possible candidates for this component, much research still needs to be done before this dream can be realized.

References

[AK89] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM SIGMOD Conference*, 1989.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *Proceedings of the Second International Workshop on Database Programming Languages*, 1989.

[CDLR89] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP, an Algebra Based Query Language for an Object-Oriented Database System. In *Proceedings of the Second International Workshop on Database Programming Languages*, 1989.

[Day] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Database Programming Languages.*

[dB90] Jan Van den Bussche. A Formal Basis for Extending SQL to Object-Oriented Databases. In *Bulletin of the European Association for Theoretical Computer Science*, number 40. February 1990.

[FAC+89] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the Iris DBMS. In *Object Oriented Concepts, Databases, and Applications*, pages 219–250. 1989.

[Fis88] D.H. Fishman. An Overview of the Iris Object-Oriented DBMS. In *Proceedings of the IEEE Spring COMPCON*, 1988.

[KL89] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *Proceedings of the ACM SIGMOD Conference*, 1989.

[SZ] G.M. Shaw and S.B. Zdonik. An Object-Oriented Query Algebra. In *Database Programming Languages.*

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of the ACM SIGMOD Conference*, 1991.