Abstract

This paper presents the design of ORL, an Object Retrieval Language interface to the EQUAL object algebra. ORL has an extended SQL syntax and is implemented on top of the ONTOS object-oriented database system. The complete system provides an interactive query facility as an alternative to the programatic OSQL interface provided by ONTOS. In general, ORL provides a more concise, user-friendly approach to the expression of queries over objects than the object algebra on which it is based. This paper demonstrates the translation of ORL queries to their equivalent object algebra expressions.

1 INTRODUCTION

Since object-oriented databases (OODB's) represent a merger of programming language and database concepts, current OODB's have both programming language and database characteristics. Most OODB's, however, are weak in support of traditional ad-hoc query languages. For realistic applications, OODB's still require query languages to allow quick, set-oriented retrieval of information about objects, without writing programs or using object-at-a-time browsers. As in the relational model, an object algebra, is needed to support set-oriented retrieval of objects in an OODB.

Several recent research projects have addressed the development of operators for accessing objects [ALAS89, BANE88, MANO86, SHAW90, BEER90, OSBO89]. Among those researchers, Shaw and Zdonik [SHAW90] present a comprehensive set of operations for the associative retrieval of objects. The algebra is known as EQUAL and is associated with the ENCORE OODB. Like the relational algebra, however, the EQUAL algebra is not the ideal language for use by end-users. A more user-friendly query language is needed as an interface to EQUAL.

The focus of this research is on the design of a query language known as ORL (Object Retrieval Language) that serves as a user interface for EQUAL. ORL is based on existing query languages that have been defined for extended relational systems [MACG85, ZANI83], semantic data models [SHIP81], and OODB's [ALAS89, CARE88, BANC89, FISH87]. Furthermore, since SQL has been proposed as an ANSI standard for database query languages [LUSA88], an objective of the language design was to develop a syntax as close to SQL as possible.

A compiler for ORL has been implemented as part of the research described in [LAI91], compiling ORL queries into an internal representation expressed in terms of the EQUAL operators. The EQUAL operators were then implemented on top of the ONTOS OODB [ONT090, SAXE91]. The complete system provides an interactive query facility that is a dramatic improvement over the current programatic OSQL interface provided by ONTOS, providing an ad-hoc, set-oriented approach to object retrieval without requiring the development of code in C++. Since ORL is also based on a formal object algebra, the language is more expressive than the programatic OSQL interface of ONTOS, supporting the dynamic creation of new objects. ORL is also flexible in the use of set-valued properties within object-to-object path expressions, allowing for a concise expression of object-oriented queries that involve set-valued properties. Most other object-oriented languages require the explicit definition of variables that range over set-valued properties.

To make the paper self-contained, an overview of EQUAL as presented in [SHAW90] is included in Section 2. Section 3 then presents the design of ORL, including examples of translating ORL to EQUAL. A summary and discussion of future use of the language for object-oriented view updates and heterogeneous database systems is then provided in Section 4.
2 OVERVIEW OF EQUAL

EQUAL was originally designed for the ENCORE object-oriented data model [SHAW90]. The ENCORE data model supports abstract data types, type inheritance, typed collections of typed objects, and object identity. A class is treated as a collection of objects of the same type. If type T is a subtype of type S, T can be used in any context where S is expected. Besides user-defined abstract data types and atomic types, ENCORE supports a global supertype Object and the tuple and set parameterized types.

ENCORE supports the comparison of objects by object identity and object equality [Khos86], allowing users to specify how "deeply" nested objects should be compared with the =i comparator. For example, =0 indicates comparison by object identity, =1 indicates shallow equality, and =3 indicates comparison of attribute values to a level of three objects deep. If two objects satisfy the =i comparator, the objects are said to be i-equal.

Figure 1 presents a University Schema using the ENCORE Data model. This schema will be used for example queries throughout the remainder of this paper. Single-headed arrows denote single-valued properties, while double-headed arrows denote multi-valued properties (i.e., properties of type Set[T]).

The EQUAL algebra includes the following operations: Select, Image, Flatten, Project, Ojoin, Nest, UnNest, DupEliminate, Coalesce, Intersection, Union, and Difference.

The Select Operator
The Select operator provides a way to select objects that satisfy a given predicate. The result of the operation is a subset of the original collection:

\[
\text{select (student, lambda s s.major.dname = "CS")}
\]

In the above example, lambda s is a variable that ranges over the student class.

The Image and Flatten Operators
Image is an operator used to return a type other than the type over which the query is initiated. Image applies a single function to each member of a set and returns a new set of objects as a result, with duplicate objects removed. For instance, the query:

\[
\text{image (graduate, lambda g g.pname)}
\]

returns a set of strings representing all of the graduate students' names. If pname was a multi-valued property, the result would be Set[Set[T]]. The Flatten operation can be used to restructure a set of sets as a single set of objects.

The Project, Nest, and UnNest Operators
The Project operator extends Image by allowing for the application of more than one function to each object of a set. The result is a collection of tuples, one tuple for each object in the collection being queried. The following query, which retrieves the names and titles of all faculty members, is an example of the former type:

\[
\text{project (faculty, lambda f \langle(F.NAME, f.pname) (F.TITLE, f.title)\rangle)}
\]

The result of the above operation is a set of tuples with two columns, F.NAME and F.TITLE. The type of F.NAME is the same as that of pname, and the type of F.TITLE is the same as that of title. Each tuple has a unique identifier, but some tuples may have the same values (i.e., tuples may be 1-equal). The Nest and UnNest operators can be used to manipulate set values within tuples.

The Ojoin Operator
The Ojoin operator is used to create relationships between objects from two collections in the database. Like Project, new tuples are created to store the generated relationships. As an example, the following query returns tuple object for any faculty member who is younger than his/her advisee:

\[
\text{ojoin (faculty, student, FACULTY, STUDENT, lambda f lambda s f.birth-date > s.birth-date and s in f.advisees)}
\]

The result is a set of new tuple objects with one column for faculty objects and one column for student objects.

The Dupeliminate and Coalesce Operators
Many of the above operators create new objects, each with its own unique object identity. These objects can be equal, though not identical. DupEliminate and Coalesce are operators provided for removing duplicates created by other operators (see [SHAW90] for details).

3 ORL DESIGN

ORL was developed to provide a more user-friendly approach to expressing queries in EQUAL. This section presents the design of ORL. The syntax of the language is described first in Section 3.1 followed examples of ORL-to-EQUAL translations in Section 3.2. The translation examples illustrate the conciseness of ORL queries compared to their EQUAL equivalents.

3.1 Query Language Interface
A query in ORL has the following SQL-like structure:
## The University Schema

<table>
<thead>
<tr>
<th>Class</th>
<th>Properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>person</td>
<td></td>
</tr>
<tr>
<td>course</td>
<td></td>
</tr>
<tr>
<td>addr</td>
<td></td>
</tr>
<tr>
<td>graduate</td>
<td>subclass of student;</td>
</tr>
<tr>
<td>enrollment</td>
<td>subclass of student;</td>
</tr>
<tr>
<td>schedule</td>
<td>subclass of person;</td>
</tr>
<tr>
<td>under-grad</td>
<td>subclass of person;</td>
</tr>
<tr>
<td>student</td>
<td>subclass of person;</td>
</tr>
</tbody>
</table>

```sql
select Epname, f.advisees.pname
from fin faculty
where f.dept.dname = "CS";
```

(Example 1)

The result of Example 1 is a set of tuples with two columns, each column being single-valued (sv). In other words, the result is a flat table of tuple objects.

### 3.1.1 The Select and From Clauses

A unique feature of ORL is found in the use of multi-valued properties in the select clause list. In ORL, the dot notation allows multi-valued (mv) properties in the middle of a path expression. In Example 1, for instance, "advisees" is an mv property of faculty, followed by "pname" which is an sv property. Other query languages (e.g., O2 [BANC89]) do not support such a feature, causing the use of unnecessary range variables.

When an mv property appears in a path expression, the result of the path expression is inherently multi-valued. Unless indicated otherwise, the mv result is flattened to provide a relational-like view. If a flattened view is not desired, the "*" feature of ORL can be used to retain the nested structure in the query result. In Example 1, if the names of each faculty object's advisees are to be collected into a set, the "*" indicator can be used as follows:

```sql
select f.pname, f.advisees.pname*
from f in faculty
where f.dept.dname = "CS";
```

(Example 2)

Items in the select clause list are not restricted to printable values. Object identifiers are not considered to be displayable units of information. However, in ONTOS, objects are assigned logical object names which are used as external identifiers. Our implementation of ORL in ONTOS uses logical object names as display values for object identifiers.

As in SQL, ORL supports the use of the keyword DISTINCT, which indicates the elimination of duplicate values in the result of the query. ORL also provides the "select-o" clause to indicate outerjoin semantics. Thus, the following query will return an empty set in the second column for those freshmen or Computer Science undergraduates who do not live in the same city as any professor:

```sql
select-o s.pname, f.pname*
from s in under-grad, f in faculty
where (s.level = "freshman" or s.major.dname = "CS") and s.address.city = f.address.city and f.title = "Professor";
```

(Example 3)

As in other OO query languages, ORL supports both
explicit and implicit range variables. Using class names as implicit variables enhances the conciseness of most queries. Thus, Example 1 is equivalent to the following query:

```sql
select faculty.pname, faculty.advisees.pname
where faculty.dept.dname = "CS";
```

(Example 4)

### 3.1.2 The Where Clause

The where clause is used to specify a Boolean combination of conditions that are to be met by objects that appear in the result of the query. The following sections present representative examples of where clause features in ORL. A more detailed discussion of each predicate can be found in [LAI91].

Before introducing each predicate type, it is first necessary to define what ORL refers to as join terms and restriction terms in the following examples. If a predicate involves different objects (e.g., s.birth-date < f.birth-date), the predicate is called a join term. A predicate that involves only one variable or one object (e.g., "s.birth-date < 620101" or "s.birth-date < s.advisor.birth-date") is called a restriction term.

#### The Comparison Predicate

Comparison predicates use standard relational operators to form restriction terms and/or join terms. Example 1 is a basic example of a comparison predicate in the where clause. An additional example is provided below.

```sql
select under-grad
where under-grad.cum_gpa >
    all (select graduate.cum_gpa);
```

(Example 5)

#### The Equality Predicate

Equality predicates are used for the comparison of objects using object identity or object equality as specified by the IS(n) operator. The "(n)" specifies the degree of equality, where "IS(0)" indicates identity.

```sql
select g.ssn
from g in graduate, f in faculty
where g.is(0) f;
```

(Example 6)

#### The Isin Predicate

The isin predicate is used in the where clause to test for set membership.

```sql
select g.ssn
from g in graduate
where g.courses_taken.acourse.subject isin
    (select f.courses_taught.subject*
     from f in faculty
     where f.title = "Professor");
```

(Example 7)

The result of the subquery in Example 7 is a set of sets of course objects, with the courses taught by each professor within a nested set. The isin predicate tests if the set of course objects taken by a graduate student is contained by any of the sets produced by the subquery. Three other variations of the isin predicate are described in [LAI91].

#### The Quantification Predicate

The quantification predicate is useful when it is necessary to test that all/any elements of a set satisfy some condition. There are two patterns of the quantification predicate: quantification over a class of objects and quantification over multi properties (as in the following example).

```sql
select d
from d in department
where for any s in d.dept.faculty.advisees
    (s.status = "probation");
```

(Example 8)

#### The Contains Predicate

The contains predicate tests whether a set contains all/any elements of another set.

```sql
select faculty
where faculty.advisees contains all
    (select student
     where student.status = "honor");
```

(Example 9)

The result of the subquery in Example 9 is a set of student objects with the status of "honor". The contains predicate tests whether a faculty member's set of advisees contains any of these student objects.

### 3.2 Object Algebra Translation Examples

This section presents examples of representing ORL queries in EQUAL. The complete set of rules for translating ORL to EQUAL can be found in [LAI91]. As illustrated in many of the examples, an ORL query is more concise and user-friendly than its corresponding set of EQUAL operators, especially those queries involving multi-valued properties.

Example 3 in the previous section is an example of a complex query with two range variables involved. The translation of the where clause starts with a series of Select operations, one for each class having a restriction term.

```sql
s0 := select (under.grad, lambda s s.level = "freshman"
    or s.major.dname = "CS")
s1 := select (faculty, lambda f.f.title = "Professor")
```

Further translation of the query depends on the join semantics of the query. Assume for the purpose of this example that Example 3 indicates join semantics rather than outer join (without the "*" on f.pname). An Ojoin operation...
with the join term as a predicate is then performed on s0
and s1 to produce the relationship between the Under-grad
objects and the Faculty objects that depend on the types of predicates involved in each query.

To retrieve the names of each pair of Student/Faculty
objects generated by s2, the following, operation is performed:

$$s2 := \text{ojoin}(s0, s1, c0, c1, \lambda s, \lambda f \text{ s.address.city = f.address.city})$$

The examples above illustrate the basic rules of transla-
tion for joins between classes. Additional translation rules
depend on the types of predicates involved in each query.

The Comparison Predicate
Example 5 illustrates the translation of a comparison
predicate involving a subquery:

$$s1 := \text{image}(\text{graduate, lambda f graduat Graduate.cum.gpa})$$
$$s2 := \text{select}(\text{under-grad, lambda under-grad for all s (in s1 and under-grad.cum.gpa > s)})$$

S1 corresponds to the select clause of the subquery, pro-
ducing a set of strings representing the cumulative GPA's
of all graduate students. S2 is the translation of the where
clause of the main query in Example 5. The result of s2 is
a subset of the Student class.

The Equality Predicate
The equality operator can be directly translated into the
equality operator, $=\$, of EQAL. When the equality pred-
icate involves join terms, the predicate is translated into an
Join; otherwise, the predicate is translated into a Select.
The translation of Example 6 is of the former type:

$$s2 := \text{ojoin}(\text{graduate, faculty, c0, c1, lambda g lambda f g = (0) f})$$
$$s3 := \text{project}(s2, \lambda s <(\text{G-SSN, s.c0.ssn}>)$$

The Isin Predicate
The most difficult translation of the isin predicate is when the predicate is of the pattern "$\langle\text{mv.path}> \text{ISIN}\langle\text{mv.subquery}\rangle$", where \text{mv.path} indicates a path expression with an \text{mv} property in the middle of the path and
\text{mv.subquery} indicates an ORL subquery that returns a set of
sets as a result. The problem is that ORL supports path expressions with \text{mv} properties in the middle of path
expressions, but EQAL does not. As an example, the translation of Example 7 is:

$$s1 := \text{select}(\text{faculty, lambda f f.title = "Professor"})$$
$$s2 := \text{image}(s1, \lambda s, \text{f.course.taught})$$
$$s3 := \text{image}(s2, \lambda s \text{ image(s, s.courses-taught)})$$
$$s4 := \text{select}(\text{graduate, lambda g for any j (in s3 and
for all k (k in g.courses.taken and k.acourse.subject in j)})$$
$$s5 := \text{project}(s4, \lambda g <(\text{G.SSN, g.ssn}>)$$

S1 - s3 correspond to the subquery in Example 7. In
s2 an Image operation with the function "$\text{f.courses.taught}$"
first performed on the subset of Faculty class defined
in s1, generating a set of sets of Schedule objects. In s3,
another Image with an embedded Image is performed on s2
to transform the set of sets of courses.taught in s2 to a set
of sets of subjects. S4 is the translation of the isin predicate
of Example 7, testing if each course taken by a student is
in any set of courses contained in the set specified by s3.
In comparing the ORL query with its EQAL equivalent,
the ORL version is more concise.

The Quantification Predicate
The quantification predicate of a query can always be
translated into a select operation. Since ORL allows for
more than one \text{mv} property in a path expression, multiple
quantification predicates have to be generated to traverse
through nested sets as illustrated in Example 8:

$$s1 := \text{select}(\text{department, lambda d for any s (in d.dept.faculty and for any s (in s1.advises and s.status = "probation")))}$$

The Contains Predicate
When the contains predicate has the meaning of a set of
objects containing all/any objects in another set, the transla-
tion is straightforward. For the "all" quantifier, the predi-
cate can be translated into a select operation with the subset
test operator in the predicate. For instance, a query of this
type is illustrated by the translation of Example 9:

$$s1 := \text{select}(\text{student, lambda student student.status = "honor"})$$
$$s2 := \text{select}(\text{faculty, lambda faculty s1 subset of
faculty.advises})$$

Different rules must be applied when the "any" quantifier
is used or when both sides of the contains predicate are a
set of values rather than a set of objects [LAI91].

The Select Clause List
The final step in the translation process for any type
of ORL query depends on the format of the Select clause
list. The presence of the DISTINCT phrase, use of the
"+" feature, the number of items in the Select clause list,
and the presence of mv path expressions have an effect on how the final result of the query is presented. Consider the following query that returns the ssn and name of all students older than each faculty member:

```sql
select f.ssn, f.pname, s.ssn*, s.pname*
from f in faculty, s in student
where f.dept is s.major and f.birth_date < s.birth_date;
```

(Example 10)

The translation of Example 10 is:

```sql
s2 := ojoin (faculty, student, c0, c1, lambda f lambda s f.dept =0 s.major and f.birth_date < s.birth_date)

s3 := project (s2, lambda s <(c0, s.c0), (c1, s.c0),
              (c2, s.c1.ssn), (c3, s.c1.pname))

s4 := nest (s3, c2)

s5 := nest (s4, c3)

s6 := project (s5, lambda s <(f.SSN, s.c0.ssn), (F.PNAME, s.c1.pname),
              (S.SSN,s.c2), (S.PNAME, s.c3))
```

In comparison, the ORL version of the query is easier to express, avoiding the details involved in proper use of the project and nest operators for formatting the output.

4 SUMMARY AND FUTURE RESEARCH

This paper provides a description of ORL, an extended SQL syntax for an object algebra, together with examples of transforming queries in ORL into their object algebra equivalents. We are currently using ORL to support two different research directions. First of all, we are investigating updatable object-oriented views using ORL as a view definition mechanism. Preliminary results are reported in [URBA92]. We are also using ORL as an interface to a heterogeneous database, using ONTOS as an intermediate store for data retrieved from different databases [ABDE92]. We plan to apply our work on view updates to support updatable user views in a heterogeneous database.

References


[MACG85] MACGREGOR, R. ARIEL--A semantic front-end to relational DBMSs. 11th VLDB (Stockholm, 1985), 305-315.


[SHAW90] SHAW, G.M., AND ZDONIK, S.B. A query algebra for object-oriented databases. 6th Int. Conf. on Data Eng. (Los Angeles, Feb. 1990), 154-162.

