The OPM Query Language and Translator *

(OPM Version 4.1)

I-Min A. Chen[†]

Anthony Kosky[‡]

Victor M. Markowitz[§]

Ernest Szeto[¶]

Data Management Research and Development Group
Information and Computing Sciences Division
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720

June 1996

*Issued as Technical Report LBL–38180. This work is supported by the Office of Health and Environmental Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

[†]Author's e-mail address: *IAChen@lbl.gov*, phone: (510) 486-7264, fax: (510) 486-4004
[‡]Author's e-mail address: *Anthony_Kosky@lbl.gov*, phone: (510) 486-5471, fax: (510) 486-4004
[§]Author's e-mail address: *VMMarkowitz@lbl.gov*, phone: (510) 486-6835, fax: (510) 486-4004
[¶]Author's e-mail address: *E_Szeto@lbl.gov*, phone: (510) 486-7565, fax: (510) 486-4004

Abstract

This document describes the Object-Protocol Model (OPM) Query Language (OPM-QL) and Query Translator (OPM-QLT) for version 4.1 of OPM.

OPM is an object data model that supports specifying scientific (e.g., molecular biology) databases and queries in terms of objects, protocols (laboratory experiments), and attributes. Attributes qualify objects and protocols, and take values that are associated with system provided data types or are objects.

OPM is currently implemented on top of relational database management systems (DBMSs). An OPM Schema Translator maps OPM schemas into relational database schemas and SQL procedures, and generates a metadata file with information regarding the correspondence of OPM and DBMS elements.

OPM-QL 4.1 extends OPM-QL 4.0 with support for variables, selection from multiple classes, and two types of projection results, "get tuple" and "get object".

OPM-QLT is based on the metadata file mentioned above, maps OPM-QL queries into SQL queries and/or procedures, executes these SQL queries and procedures, and returns the query results structured in terms of OPM constructs. OPM-QLT 4.1 has been entirely rewritten, rather than being an extension of OPM-QLT 4.0.

Contents

1 Introduction

The *Object-Protocol Model* (OPM) is described in [2]. OPM supports the specification of object classes and protocol classes. An object class is identified by a class name, has an optional class description, and is associated with attributes that qualify the objects (instances) of the class. OPM classes can be organized in a class hierarchy representing subclass-superclass relationships. A subclass is a specialization of its superclasses, and inherits all the attributes associated with its superclasses; multiple inheritance in class hierarchies is supported.

Attributes can have values that are atomic or consist of a tuple of atomic values, and can be single-valued, set-valued, or list-valued. Attributes take values from other classes or are associated with system provided data types. OPM supports the specification of derived attributes using derivation rules involving arithmetic expressions, aggregate functions, and attribute composition.

Attributes can be generic or versioned. Generic attributes can be used for describing stable properties of an object or protocol (such as the social security number of a person), while versioned attributes can be used for describing the evolving properties of an object or protocol (such as the address of a person).

Protocol classes in OPM are used to model laboratory experiments. Given an input, a protocol instance (experiment) results in an output, where both input and output consist of objects. OPM supports the recursive specification (expansion) of protocols. Protocol expansion in OPM allows specifying a protocol in terms of alternative subprotocols, sequences of subprotocols, and optional protocols. A protocol class can be associated with regular as well as input and output attributes. Input and output attributes are used for specifying input and output connections between protocols.

OPM also supports two types of derived object classes: derived subclasses and derived superclasses. A derived subclass is defined as a subclass of another derived or non-derived object class with an optional derivation condition. A derived superclass is defined as a union of two or more derived or non-derived object classes.

The OPM query language (OPM-QL) follows the ODMG-93 standard for object-oriented query languages [5]. OPM is currently implemented on top of relational database management systems (DBMSs). In order to implement OPM on top of a relational DBMS, an OPM Schema Translator maps OPM schemas into DBMS-specific relational schema definitions and SQL procedures [4]. The OPM Schema Translator also generates a metadata file with information regarding the correspondence of OPM and DBMS elements.

In this document we describe the OPM Query Translator for OPM 4.1. The OPM Query Translator is based on the metadata file mentioned above, maps OPM queries into SQL queries and/or procedures, executes these SQL queries and procedures, and returns the query results structured in terms of OPM constructs.

The rest of this document is organized as follows. The OPM query language is described in section 2. Query translation strategies are discussed in section 3. The OPM query trans-

lator is presented in section 4. Section 5 contains a user guide for the OPM Query Language Translator. An OPM schema example is given in Appendix A. Appendix B contains the syntactic definition for the OPM query language.

2 The OPM Query Language

In this section we present the main constructs of the OPM query language.

The OPM query language follows the ODMG-93 standard for object-oriented query languages [5].

An OPM query consists of a data manipulation statement that can be a `SELECT`, `INSERT`, `DELETE`, or `UPDATE` statement. An insert, delete or update query involves a single *target class*, while a select query may involve one or more target classes. These statements can involve conditions consisting of and-or compositions of atomic comparisons (see section 2.1.4).

2.1 Select Queries

An OPM `SELECT` query consists of three parts: a `SELECT` statement specifying the what attribute values should be retrieved for the objects and/or protocols satisfying the query condition; a `FROM` statement specifying the classes containing the objects and/or protocols that are considered by the query; and an optional `WHERE` statement specifying conditions on objects and/or protocols. Intuitively the meaning of a `SELECT` query is "for each binding of variables specified in the `FROM` statement, such that the conditions specified in the `WHERE` statement are satisfied, return the values of the attributes specified by the `SELECT` statement".

```
<SELECT query> ::= <select statement>
                   <from statement>
                   <optional where statement> ';'
                 | <select statement>
                   <from statement>
                   <optional where statement>
                   ORDER BY <order expressions> ';'
                 ;

<select statement> ::= SELECT <select declarations>
                     | SELECT DISTINCT <select declarations>
                     ;

<select declarations> ::= <select declaration>
                        | <select declarations> ',' <select declaration>
                        ;
<select declaration> ::= <attribute> = <select expression>
                       | <select expression>
                       ;

<select expression> ::= <expression> '(' <attribute list> ')'
                      | <expression> '(' '*' ')'
```

```
                    | <expression>
                    ;

<attribute list> ::= <attribute>
                   | <attribute list> ',' <attribute>
                   ;
```

The select statement binds expressions to the various attributes in the result of a query. A query will therefore return a set of tuples, with the specified attributes. If an attribute name is omitted then the system will generate an appropriate name based on the expression.

If e is an expression evaluating to an object, then a select declaration a `=` e in the select statement will bind the object identity of e to the attribute a. A select declaration a `=` e`(`$b_1, \ldots, b_n$`)` binds a tuple to the attribute a consisting of the object identity e and the attributes b_1 to b_n of e. Note that the attributes of e may be set valued. A select declaration a `=` e`(*)` binds a tuple consisting of the object identity of e and all the attributes of e to the attribute a. A more detailed description will be deffered until section 2.1.3.

```
<from statement> ::= FROM <variable declarations>
                   ;
<variable declarations> ::= <variable declaration>
                          | <variable declarations> ',' <variable declaration>
                          ;
<variable declaration> ::= <variable> IN <class expression>
                         ;

<variable> ::= <identifier>
             ;

<class expression> ::= <class name>
                     | <class name> ALL VERSIONS
                     | <variable> '.' <projection element>
                     ;

<projection element> ::= <attribute name> '[' <class name> ']'
                       ;

<optional where statement> ::= <null>
                             | WHERE <conditions>
                             ;

<order expressions> ::= <expression>
                      | <order expressions> ',' <expression>
                      ;
```

A variable declaration in a `FROM` statement names a variable and declares the set of values that it may range over. For example the from statement

```
FROM P IN PERSON, F IN P.owns[FRAGMENT]
```

would declare that P is a variable ranging over the class `PERSON`, and that, for any instantiation of P, F is a variable ranging over the values of abstract attribute `owns` of P which are objects in class `FRAGMENT`.

If a class, C, used in a variable binding is versioned (i.e., has versioned attributes), then the keywords `ALL VERSIONS` can be specified after the binding in order to get *all* the versions of objects of that class; otherwise, the variable is restricted to *default* versions only.

We say a variable X *has class* C if the `FROM` statement of a clause either contains a binding X `IN` C or a binding X `IN` $Y.a$`[`C`]` for some variable Y and attribute a. We require that each variable only has one class in a particular query, though a variable may have multiple bindings.

We also require that every variable occuring in a query be declared in its `FROM` statement, and that a `FROM` statement does not contain any cyclic declarations of variables. For example,

```
FROM X IN Y.spouse[PERSON], Y IN X.spouse[PERSON]
```

would not be allowed.

More precisely, given a `FROM` statement, we define its *dependency graph* to be a directed graph, G, with the variables declared in the `FROM` statement as nodes, such that G contains an edge from X to Y iff the `FROM` statement contains a declaration Y `IN` $X.attr$`[`$Class$`]`. We require that the dependency graph associated with any `FROM` statement be acyclic.

A *valid instantiation* for a `FROM` statement is an assignment of object identities or primitive values to each variable declared in the the `FROM` statement such that, if the `FROM` statement contains a declaration of the form Y `IN` $Class$ or Y `IN` $X.attr$`[`$Class$`]` then the value associated with Y is either an object identity or primitive value belonging to the class $Class$, and, if the `FROM` statement contains a declaration Y `IN` $X.attr$`[`$Class$`]` then the value associated with X is an object identity, and the value associated with Y is in the $attr$ attribute of the object associated with X.

2.1.1 Semantics in Relational Model Terms

We will attempt to make the semantics of the query language more precise by describing how it would be interpreted in an underlying representation of an OPM database in the relational model.

Suppose that `C` is an object class with single valued attributes $a_1, \ldots, a_n$ and set or list valued attributes $b_1, \ldots, b_m$. For simplicity we will assume that `C` is represented by $m + 1$ relations: a relation C with attributes `_oid` and $a_1, \ldots, a_n$, and for each set-valued attribute, b_j, a relation C_{b_j} with attributes `_oid` and b_j.

Note that, in practice, the representation of classes will be more complicated than this because of inheritance, and because of the presense of tuple attributes. However the relationships mentioned above could be formed as joins and projections on the actual relationships used to represent a class.

We assume a distinguished value *Null* which will be used to represent *unbound variables*.

A *valid instantiation* for a `FROM` statement is then an assignment of a value to each variable declared in the `FROM` statement such that:

1. if the `FROM` statement contains a declaration Y `IN` C or Y `IN` $X.a$`[`C`]` then either $Y \in \pi_{\texttt{\_oid}}(C)$ or $Y = Null$;
2. if the `FROM` statement contains a declaration of the form Y `IN` $X.a_i$`[`C`]`, where a_i is a single-valued attribute, then $(X, Y) \in \pi_{(\texttt{\_oid},a_i)}(C)$;
3. if the `FROM` statement contains a declaration of the form Y `IN` $X.b_j$`[`C`]`, where b_j is a set-valued attribute, then either $(X, Y) \in \pi_{(\texttt{\_oid},b_j)}(C_{b_j})$ or $Y = Null$; and
4. if $Y = Null$ then either the `FROM` statement contains a declaration of the form Y `IN` $X.a_i$`[`C`]` where $(X, Null) \in \pi_{(\texttt{\_oid},a_i)}(C)$ or $X = Null$, or the `FROM` statement contains a declaration Y `IN` $X.b_j$`[`C`]` where $\sigma_{(\texttt{\_oid}=X)}(C) = \emptyset$ or $X = Null$.

Example. Suppose we had an object class

```
OBJECT CLASS Person
  ATTRIBUTE name: [0,1] String
  ATTRIBUTE children: set-of [0,] String
```

and in an instance the class was represented by the table `Person`:

_oid	name
1	"Fred"
2	"Joe"
3	*Null*

and the table $\texttt{Person}_{\texttt{children}}$:

_oid	children
1	"Arthur"
1	"Sally"
3	"Jim"

and that a query has the `FROM` statement

```
FROM X IN Person, Y IN X.name, Z IN X.children
```

Then the valid instantiations of X, Y and Z are given by:

X	Y	Z
1	"Fred"	"Arthur"
1	"Fred"	"Sally"
2	"Joe"	*Null*
3	*Null*	"Jim"

2.1.2 Expressions

An OPM Query Language expression may be either a variable, a primitive value or a finite set of primitive values.

```
<expression> ::= <variable>
               | <primitive value>
               | <set of values>
               ;

<primitive value> ::= <number>
                    | <string>
                    ;

<set of values> ::= '{' <set of numbers> '}'
                  | '{' <set of strings> '}'
                  ;
<set of numbers> ::= <number>
                   | <set of numbers> ',' <number>
                   ;
<set of strings> ::= <string>
                   | <set of strings> ',' <string>
                   ;
```

Given an instantiation of variables, we may extend it to assign values to expressions: a variable expression is assigned the same value as associated with the variable by the instantiation, while values are assigned to primitive expressions (numbers and strings) in the standard way.

2.1.3 Select Expressions and Select Statements

Given an instantiation of variables, we also assign a values to select statements and select expressions for that instantiation.

The value associated with a select expression ϵ is given by:

1. If e is an expression then its value as a select expression is the same as its value as an ordinary expression;

2. If e has the form e'`(`$a_1, \ldots, a_n, b_1, \ldots, b_m$`)` where e' is an expression which evaluates to an object identity, o of class C in the variable instantiation, $a_1, \ldots, a_n$ are single-valued attributes of C and $b_1, \ldots, b_m$ are set valued attributes of C, then the value of e is a tuple
$$(\texttt{\_oid} = o,\ a_1 = \pi_{a_1}(\sigma_{\texttt{\_oid}=o}(C)),\ \ldots, a_n = \pi_{a_n}(\sigma_{\texttt{\_oid}=o}(C)),\\ b_1 = \pi_{b_1}(\sigma_{\texttt{\_oid}=o}(C_{b_1})),\ \ldots, b_m = \pi_{b_m}(\sigma_{\texttt{\_oid}=o}(C_{b_m})))$$

3. If e has the form e'`(*)`, where e' is an expression that evaluates to an object identity o of class C, and $a_1, \ldots, a_n$ are all the single valued attributes of C and $b_1, \ldots, b_n$ are all the set-valued attributes of C, then the value of e is
$$(\texttt{\_oid} = o,\ a_1 = \pi_{a_1}(\sigma_{\texttt{\_oid}=o}(C)),\ \ldots, a_n = \pi_{a_n}(\sigma_{\texttt{\_oid}=o}(C)),\\ b_1 = \pi_{b_1}(\sigma_{\texttt{\_oid}=o}(C_{b_1})),\ \ldots, b_m = \pi_{b_m}(\sigma_{\texttt{\_oid}=o}(C_{b_m})))$$

Given a select statement, `SELECT` a_1`=`e_1`, ...,` a_k`=`e_k then the value of the select statement for a particular variable instantiation is the tuple
$$(a_1 = v_1,\ \ldots, a_k = v_k)$$
where v_i is the value of the select expression e_i for $i = 1, \ldots, k$.

A special case is where a `SELECT` statement has only one select declaration with the attribute omitted. That is the `SELECT` statement has the form

```
SELECT <select expression>
```

In this case the value of the select statement is the value of the expression, as opposed to a tuple with one system-generated attribute pointing to the value of the expression.

2.1.4 Conditions

Conditions in the OPM Query Language are used to restrict the possible instantiations of variables in a `SELECT`, `DELETE` or `UPDATE` query. Conditions are formed using conjunctions and disjunctions of various atomic comparisons of expressions.

```
<conditions> ::= <atomic condition>
               | ( <conditions> )
               | <conditions> AND <conditions>
               | <conditions> OR <conditions>
               ;
<atomic condition> ::= <expression> <is null op>
```

```
                    | <expression> <bin op> <expression>
                    | <expression> <comp op> ALL <variable>
                    | <expression> <match op> <string>
                    ;
<is null op> ::= IS NULL
               | IS NOT NULL
               ;
<bin op> ::= <comp op>
           | <in op>
           | <contains op>
           ;
<comp op> ::= '='  | '!='  | '>' | '>='  | '<'  | '<='
            ;
<contains op> ::= CONTAINS | NOT CONTAINS
                ;
<in op> ::= IN | NOT IN
          ;
<match op> ::= MATCH | NOT MATCH
             ;
```

An atomic condition is said to be satisfied by a particular instantiation of variables if, for that instantiation of variables, the values of the expressions involved in the condition make the condition true. More precisely:

1. A condition e `IS NULL` is satisfied iff the value of the expression e in the variable instantiation is *Null*, while e `IS NOT NULL` is satisfied if the value of e is not equal to *Null*;

2. A condition e_1 θ e_2, where θ is one of `=`, `!=`, `<`, `>`, `<=` or `>=`, is satisfied if the values associated with e_1 and e_2 are both not equal to *Null*, and the binary relationship associated with θ holds between e_1 and e_2. (e.g., e_1 `=` e_2 is satisfied if the values of e_1 and e_2 are equal and not equal to *Null*, while e_1 `!=` e_2 is satisfied if the values associated with e_1 and e_2 are not equal, and neither value is equal to *Null*.)

3. A condition e θ `ALL` v, where e is an expression and v a variable, is satisfied by a particular instantiation iff for *any* valid variable instantiation which is equal to the current instantiation on all variables other than v (and possibly on v as well), the condition e θ v is satisfied;

4. A condition e_1 `CONTAINS` e_2 is satisfied iff the value of e_1 is a set and the value of e_2 occurs in e_1. e_1 `NOT CONTAINS` e_2 is satisfied if the value of e_1 is a set, the value of e_2 is not equal to *Null*, and the value of e_2 does not occur in the value of e_1;

5. A condition e_1 `IN` e_2 is satisfied iff the value of e_2 is a set and the value of e_1 occurs in e_2. e_1 `NOT IN` e_2 is satisfied if the value of e_2 is a set, the value of e_1 is not equal to *Null*, and the value of e_1 does not occur in the value of e_2;

6. A condition e_1 `MATCH` e_2 is satisfied if the values of e_1 and e_2 are strings, and the value of e_2, treated as a *regular expression* matches the value of e_1. e_1 `NOT MATCH` e_2 is satisfied if the values of e_1 and e_2 are strings, and the value of e_2 does not match the value of e_1.

and so on.

For example suppose we have a variable instantiation which is valid for the `FROM` statement

```
FROM X IN PERSON, Y IN X.owns[FRAGMENT]
```

Then the condition `Y IS NULL` would be satisfied iff the `owns` attribute of `X` is the empty set.

Note: this semantics of comparison operators with respect to Nulls appears to be reasonable and necessary because of the use of *Null* to represent empty set-valued attributes. For example, if we wished to ask a query "what are the names of people who own a Fragment owned by John", we could write

```
SELECT X.name
FROM   X IN PERSON, J IN PERSON,
       XF IN X.owns[FRAGMENT], JF IN J.owns[FRAGMENT]
WHERE  J.name = "John"
AND    XF = JF
;
```

If we made the condition `XF = JF` true when both `XF` and `JF` were equal to *Null*, and if the `owns` attribute of John was the empty set, then this query would return the names of any People that do not own any Fragments.

2.2 Syntactic Extensions and Shorthand Notations

The syntax introduced so far is relatively straightforward to translate into SQL queries on an underlying relational database, but is somewhat unwieldy, and makes navigating paths of attributes from an object difficult. In this section we will introduce a series of extensions to this syntax intended either to make the query language easier to use, or to allow for queries expressed in previous versions of the query language. None of these extensions add to the expressive power of the language, and queries expressed using these extensions can be translated back into queries using only the constructs introduced above.

2.2.1 Path Expressions

We first extend our *projection elements* to include reverse attributes and attributes with implicit classes. The new BNF for projection elements is

```
<projection element> ::= <attribute name> '[' <class name> ']'
                       | '!' <attribute name> '[' <class name> ']'
                       | <attribute name>
                       ;
```

Here the construct `!`*attr*`[`*Class*`]` represents a *reverse attribute*. For example the variable declarations `Y IN FRAGMENT`, `X IN Y.!owns[PERSON]`, would be equivalent to the declarations `X IN PERSON`, `Y IN X.owns[FRAGMENT]`.

If the `[`*Class*`]` is omitted from a projection element, then it is equivalent to using the class of the attribute. For example if the `owns` attribute of class `PERSON` has class `FRAGMENT`, then the declaration `X IN Y.owns` is equivalent to `X IN Y.owns[FRAGMENT]`.

Next we allow variable declarations to involve series of attributes or reverse attributes rather than just single attributes. The extended syntax for variable declarations is

```
<variable declaration> ::= <variable> IN <class expression>
                         ;
<class expression> ::= <class name>
                     | <variable> '.' <projection composition>
                     ;
<projection composition> ::= <composition element list> <projection element>
                           ;
<composition element list> ::= <null>
                             | <composition element> <composition element list>
                             ;
<composition element> ::= <attribute name> '[' <class name> ']'
                        | '!' <attribute name> '[' <class name> ']'
                        | <attribute name> '.'
                        ;
```

A variable declaration involving a projection composition is equivalent to a series of variable declarations involving *implicit* or *system generated* variables. For example the variable declaration `X IN Y.owns[FRAGMENT]sequence` would be equivalent to `_1 IN Y.owns[FRAGMENT]`, `X IN _1.sequence[VARCHAR(200)]`, where `_1` is a new variable.

2.2.2 Implicit Variable Declarations

As the syntax is defined so far, all variables must be declared in the `FROM` statement of a query, and path expressions may only occur in the `FROM` statement. However it is often more

convenient to use path expressions directly in the SELECT and WHERE clauses of query. For example, rather than writing

```
SELECT Z
FROM   X IN FRAGMENT, Y IN X.!owns[PERSON]name, Z IN X.fragment_id
WHERE  Y = "John"
;
```

we would like to be able write

```
SELECT X.fragment_id
FROM   X IN FRAGMENT
WHERE X.!owns[PERSON]name= "John"
;
```

In order to allow such queries we extend the definition of *expressions* to include class expressions:

```
<expression> ::= <class expression>
               | <variable>
               | <primitive value>
               | <set of values>
               ;
```

When a class expression, say *CE*, occurs in a WHERE or a SELECT statement, it is equivalent having some *new* system generated variable, say *_i* in place of the class expression, and having the declaration *_i* IN *CE* in the FROM statement.

It is important to note that each occurrence of a class expression in a SELECT or WHERE statement is equivalent to a *distinct* new variable. For example the query

```
SELECT X.name
FROM   X IN PERSON
WHERE  X.owns[FRAGMENT]length > 1000
AND    X.owns[FRAGMENT]length < 10000
;
```

is equivalent to

```
SELECT _1
FROM   X IN PERSON, _1 IN X.name,
       _2 IN X.owns[FRAGMENT]length, _3 IN X.owns[FRAGMENT]length
WHERE  _2 > 1000
AND    _3 < 10000
;
```

That is, the query returns the names of those people who own a fragment with length greater than 1,000 and a fragment with length less than 10,000, not those people who own a fragment with length between 1,000 and 10,000. To express the later query we would need to use an explicit variable:

```
SELECT X.name
FROM   X IN PERSON, Y IN X.owns[FRAGMENT]length
WHERE  Y > 1000
AND    Y < 10000
;
```

2.2.3 More Implicit Variables

In the special case where we have exactly one starting class for a query, and no repeated variables, we can avoid explicit variables altogether. In this case we can allow from statements of the form

```
FROM <class name>
```

expressions of the form

```
<projection composition>
```

and select statements of the forms

```
  SELECT '(' <attribute list> ')'
| SELECT *
```

Here a `FROM` statement, `FROM` C would be shorthand for `FROM` $\_i$ `IN` C where $\_i$ is some implicit variable; an expression *projs* where *projs* is some projection composition would be shorthand for $\_i$.*projs*, where $\_i$ is the same implicit variable as bound in the in `FROM` statement; and the `SELECT` statements `SELECT (`a_1`, ...,`a_n`)` and `SELECT *` are equivalent to `SELECT` $\_i$`(`a_1`, ...,`a_n`)` and `SELECT` $\_i$`(*)` respectively.

For example the query

```
SELECT (name, age)
FROM   PERSON
WHERE  age > 50
;
```

would be equivalent to

```
SELECT _0 (name, age)
FROM   _0 IN PERSON
WHERE  _0.age > 50
;
```

Note that these extensions to expressions and `SELECT` statements are only meaningful in the case where the `FROM` statement consists of exactly one class name, and are not allowed in any other situation.

2.2.4 Tuple Attributes

If a class involves tuple attributes then we need a method of *grouping* together the attributes of a single tuple. For example suppose the schema for the class `PERSON` had an attribute

```
OBJECT CLASS PERSON
  ....
  ATTRIBUTE degree_info (degree_date, degree, field) :
    list-of [1,] ([0,1] DATETIME, [0,1] CHAR(20), [0,1] CHAR(50))
```

We would like to be able to ask queries such as finding the Persons that had obtained a degree in the field of "Genetics" in the year 1980.

We need to extend the definition of variable declarations to allow grouping of variables:

```
<variable declaration> ::= <variable> IN <class expression>
                         | '(' <variable list> ')' IN
                              <variable> '.' '(' <projection element list> ')'
                         ;

<variable list> ::= <variable>
                  | <variable list> ',' <variable>
                  ;
<projection element list> ::= <projection element>
                            | <projection element list> ',' <projection element>
                            ;
```

For example a query to find those people who obtained a degree in Genetics in 1980 would be:

```
SELECT X.name
FROM   X IN PERSON, (Y, Z) IN X.(degree_date, field)
WHERE  Y MATCH "%1980"
AND    Z = "Genetics"
;
```

In terms of our relational representation of OPM classes, if a class `C` has a tuple attribute d with fields $d_1, \ldots, d_m$ then this can be represented by a relation C_d with $m+1$ attributes, $d_1, \ldots, d_m$ and `_oid`.

A variable instantiation is then valid for a declaration of the form $(Y_1, \ldots, Y_k)$ `IN` `X`$.(d_1, \ldots, d_k)$ iff $(X, Y_1, \ldots, Y_k) \in \pi_{(\texttt{\_oid}, d_1, \ldots, d_k)}(C_d)$.

2.3 Insert Queries

An `INSERT` statement, unlike a `SELECT` statement, must specify a single *target* class. OPM-QL supports two distinct kinds of insert: a regular insert which, if successful, will always create a new object with a new oid, and `INSERT AS` statements, which extend an existing object to a new subclass.

A regular `INSERT` statement expresses the insertion of an instance into a class, and can specify values for all non-derived attributes of the class (including values for inherited attributes). An `INSERT` is required to specify values for any non-null attributes of a class.

If an `INSERT` statement inserts an object with oid x into a class O_i, then it will also automatically insert objects with oid x into any superclasses O_j of O_i. If such an insert causes any integrity constraints to be violated then the insert will be rolled back and fails. For example, if class O_i inherits a *key* attribute a from class O_j, and we attempt to insert an object x into O_i, where O_j already contains an object y with the same value as x on the attribute a, then the insert would fail.

The value of an abstract attribute is represented by a class name together with values for an identifying set of attributes (a *key*). Multiple values assigned to a set or list valued attribute are enclosed within { }. Tuple attribute values are enclosed within ( ).

For example consider the classes `SEQUENCE`, `PROJECT`, `ENTRY`, and `GEL_LANE` of the OPM schema in Appendix A. The following `INSERT` statements inserts a new sequence into `SEQUENCE` and two entries into `ENTRY`:

```
INSERT SEQUENCE (
    sequence_id = 101,
    generated_by = PROJECT [project_id = 12],
    length = 150,
    entries = { ENTRY [entry_id = 1011], ENTRY [entry_id = 1012] },
    gel_analysis = GEL_LANE [anal_tp = 'xxx']
 );

 INSERT ENTRY (
   entry_id = 1011,
   from_sequence = SEQUENCE [sequence_id = 101],
   (begin_pos, end_pos) = (1, 100)
 );
```

```
INSERT ENTRY (
  entry_id = 1012,
  from_sequence = SEQUENCE [sequence_id = 101],
  (begin_pos, end_pos) = (101, 150)
);
```

Unlike a regular INSERT statement, an INSERT AS statement does not create a new object identity. Instead it inserts an object into its target class with the same oid as an object already present in a superclass. An INSERT AS statement must specify values for any required local attributes, and values for key attributes which are sufficient to identify an object of a superclass from which the object inserted can inherit. A INSERT AS statement may specify values for inherited attributes, but, in order for the insert to be valid, each value specified for an inherited attribute must agree with the existing values of the attributes for the object, unless the existing values of the attributes are *Null*.

For example suppose an object with oid x were in a class Q_j and an INSERT AS statement was adding x to a class Q_i, a subclass of Q_j. If Q_j has an attribute a and the INSERT AS statement specifies a value v for the (inherited) attribute a, then the value of the a attribute of x in Q_j must either be *Null* or be v already in order for this insert to be valid.

Further an INSERT AS statement is required to specify values for any required, inherited attributes that are not already determined by existing instances of the object being inserted. For example suppose Q_i is a subclass of classes Q_j and Q_k, that x is an object of class Q_j which is not in Q_k (or Q_i), and that Q_k has a required local attribute a. Then an INSERT AS statement adding x to class O_i is required to specify a value for the inherited attribute a as well as for any local required attributes of O_i.

```
<INSERT query> ::= INSERT <class name> '(' <assign values> ')'  ';'
                 | INSERT <class name> '(' <assign values> ')'
                   AS <an id value> ';'
                 ;

<assign values> ::= <assign attribute values>
                  | <assign values> ',' <assign attribute values>
                  ;

<assign attribute values> ::= <simple single assignment>
                            | <simple multi assignment>
                            | <tuple single assignment>
                            | <tuple multi assignment>
```

```
                            ;

<simple single assignment> ::= <attribute name> '=' <attr value>
                             | <attribute name> '=' NULL
                             ;

<id assignment> ::= <attribute name> '=' <a primitive value>
                  ;

<attr value> ::= <a primitive value> | <an id value>
               ;

<a primitive value> ::= <integer> | <string> | <real>
                      ;

<an id value> ::= <class name> '[' <id values> ']'
                ;

<id values> ::= <id assignment> | <id values> ',' <id assignment>
              ;

<simple multi assignment> ::= <attribute name> '='
                              '{' <multiple assignments> '}'
                            ;

<multiple assignments> ::= <attr value>
                         | <multiple assignments> ',' <attr value>
                         ;

<tuple single assignment> ::=  '(' <component names> ')' '='
                               '(' <multiple assignments> ')'
                            |  '(' <component names> ')' '=' NULL
                            ;

<tuple multi assignment> ::= '(' <component names> ')' '='
                             '{' <multi comp assignments> '}'
                           ;

<multi comp assignment> ::= '(' <multiple assignments> ')'
                          | <multi comp assignments> ','
                            '(' <multiple assignments> ')'
                          ;
```

```
<component names> ::= <attribute name>
                    | <component names> ',' <attribute name>
                    ;
```

2.4 Delete Queries

A DELETE statement is associated with a target class, and can be associated with a WHERE statement involving this class. A DELETE statement expresses the removal of instances from the target class, where the deletion is automatically propagated to all the subclasses of the target class. The target class is represented by one variable, but the condition expressed using variables may involve more than one class.

```
<DELETE query> ::= DELETE <simple variable> <from statement>
                     <optional where statement> ';'
                   ;
```

Examples. Consider classes SEQUENCE and ENTRY, of the OPM schema in Appendix A. The following DELETE statements deletes sequence *123* and all of its entries:

```
DELETE S FROM S in SEQUENCE WHERE S.sequence_id = 123;

DELETE E FROM E in ENTRY WHERE E.from_sequence.sequence_id = 123;
```

2.5 Update Queries

The attribute values of an instance can be modified using UPDATE statements. An UPDATE statement involves non-derived local and inherited attributes associated with a target class, and can be associated with a WHERE statement involving this class. Again, the target class is represented by one variable, but the condition expressed using variables may involve more than one class.

```
<UPDATE query> ::= UPDATE <simple variable> '(' <attribute updates> ')'
                     <from statement> <optional where statement> ';'
                   ;

<attribute updates> ::= <attribute update>
```

```
                      | <attribute updates> ',' <attribute update>
                      ;

<attribute update> ::= <attribute insert statement>
                     | <attribute modify statement>
                     ;

<attribute insert statement> ::= ADD <assign attribute values>
                               ;

<attribute modify statement> ::= SET <assign attribute values>
                               ;
```

Examples. Consider class `SEQUENCE` of the OPM schema in Appendix A. The following `update` statement modifies a sequence:

```
UPDATE S (
  SET generated_by = PROJECT [project_id = 20],
  SET length = 50,
  ADD entries = ENTRY [entry_id = 1013]
)
FROM S in SEQUENCE
WHERE S.sequence_id = 101;
```

Consider class `CLONE` of the OPM schema in Appendix A. The following `UPDATE` statement modifies all the clones generated by project 30:

```
UPDATE C (
  SET clone_type = 'Unknown',
  SET owner = NULL
)
FROM C in CLONE
WHERE C.project.project_id = 30;
```

Note that carrying out an insert, delete, or update query entails enforcing the appropriate insert, delete, or update rules.

3 Query Processing Strategies

We discuss in this section alternative ways of processing OPM queries expressed over a relational database developed with a commercial relational DBMS.

3.1 Processing Select Queries

Processing an OPM-QL `select` query Q can be divided into four steps:

1. translating OPM query Q into one or more SQL queries;
2. executing the SQL queries using the underlying relational DBMS;
3. converting relational query results into an OPM representation;
4. further processing the query results if necessary.

The query translation process is based on the mapping (metadata) information regarding the representation of classes in the relational database. We briefly discuss below three strategies for processing OPM queries.

3.1.1 Method Based Query Processing

Representing OPM classes using relational DBMS constructs entails representing not only class definitions using relations and constraints, but also retrieval and update methods using SQL procedures (see [3]).

A retrieval method for a class O_i involves SQL procedures representing retrieval methods for each attribute A of O_i: procedures for non-derived attributes consist of simple selections from the primary or auxiliary relation containing the relational attribute representing A; procedures for derived attributes consist of computing their values using arithmetic expressions, aggregate functions, or a sequence of joins (for composition derived attributes). The retrieval method for a class O_i consists of combining the retrieval methods for the attributes of O_i.

The retrieval methods mentioned above can be used for retrieving the values of all the attributes mentioned in the `select` and `where` statements of the OPM query. Subsequently, a query processor external to the DBMS can be used for evaluating the query conditions and for selecting the instances that satisfy these conditions.

Consider the following query for finding the names of persons owning a fragment that is also owned by John:

```
SELECT X.name
FROM   X IN PERSON, J IN PERSON,
       XF IN X.owns[FRAGMENT], JF IN J.owns[FRAGMENT]
WHERE  J.name = "John"
AND    XF = JF
;
```

Using retrieval methods, the values of all attributes for all instances of class `FRAGMENT` can be retrieved; using the retrieval method for attribute `name` of class `PERSON`, the values of derived (composition) attribute `owner.name` can be retrieved; finally, for each `FRAGMENT` instance x, the query processor verifies whether the `owner.name` value for x matches "John".

Using retrieval methods has the advantage of being simple and complying with the object-oriented principle of encapsulation. However, this approach can be slow if only a small number of instances in the target class satisfy the query condition. For example, if there are only a few fragments that satisfy the condition of the query then using retrieval methods for processing the query will result in first retrieving all fragments and then examining them one by one.

3.1.2 Query Rewriting

OPM queries can be translated directly into one or several SQL queries by rewriting the OPM query conditions into SQL query conditions and by incorporating the necessary join conditions into the generated SQL queries, as proposed in [1]. Provided that all OPM query conditions can be translated into SQL conditions, query processing outside the DBMS is not needed.

Query rewriting outperforms the retrieval method based approach when simple conditions are involved in the OPM queries and when only a small amount of data satisfy the query conditions. For expressing more complex conditions, such as conditions involving set comparisons, translation into the SQL dialect of commercial DBMSs becomes more difficult and less efficient since it requires the generation of complex SQL (sub)queries. For example, conditions involving aggregate function derived attributes that entail using `group by` and `having` SQL constructs, must be translated into several SQL subqueries, and therefore involve temporary relations for intermediate query results. Query processing involving temporary relations is usually inefficient when large amounts of data are involved. Furthermore, if the resulting SQL queries involve several levels of nested subqueries or multiple joins, then the execution of such queries is very slow, and therefore is not desirable for a production system. Generating SQL queries involving numerous joins is particularly problematic when some of the joins are outerjoins. Left outerjoins and regular joins are not associative and join ordering cannot be expressed in a single SQL query in DBMSs such as Oracle and Sybase.

Finally, an OPM query can be translated into different SQL queries that are semantically equivalent but perform differently. For example, an OPM query involving disjunctive conditions can be translated into different SQL queries, such as: (i) an SQL query involving a `union` of subqueries, (ii) an SQL query involving `select distinct` (for removing duplicates) and `or` conditions, or (iii) an SQL query involving nested subqueries. The performance of these three queries is significantly different and depends on the underlying DBMS (e.g., for Sybase the first query performs best). Such DBMS-dependent performance issues are not addressed in research papers on query optimization.

An additional cost of the query rewriting approach regards the conversion of relational query results into an object-oriented representation (see step 3 above). For example, when an SQL query is used for retrieving instances together with several set-valued attributes, the returned result contains the cross product of all set-valued attribute values. Further processing is needed for converting such a result into class instances with proper attribute values.

3.1.3 Iterative Query Processing

Iterative query processing involves using query rewriting to generate an SQL query for retrieving the values of single-valued attributes appearing in a `select` clause and for evaluating conditions involving single-valued comparisons, and retrieving separately (i.e., using different SQL queries) values of set-valued and derived attributes appearing in the `select` clause. Some conditions involving set-valued attributes (e.g., testing whether the values of a set-valued attribute contains a constant or the value of a single-valued attribute) can be also translated directly into SQL conditions. Other conditions involving set-valued attributes (e.g., set-comparison of set-valued attributes) do not have a straightforward translation, and can result in deeply nested (with multiple levels of `not in` or `not exists` SQL subqueries [1]) and inefficient SQL queries. Such conditions are processed iteratively by first retrieving the values of the set-valued attribute(s) involved in set-comparisons and then using a query processor external to the DBMS for evaluating the conditions.

For example, only query rewriting needs to be used for the query above because its associated condition does not involve set-valued comparisons and therefore can be translated directly into SQL conditions, and its `select` clause involves only a single-valued attribute.

The main purpose of the iterative query processing is to avoid the generation of SQL queries that are too complex, too dependent on the SQL dialect of a particular DBMS, or are inherently inefficient (e.g., avoid employing outerjoins which are both inefficient and supported differently by different DBMSs). By evaluating part of the OPM query conditions using SQL, this query processing strategy takes advantage of the underlying DBMS. At the same time, its query translation is relatively simple and avoids generating queries with numerous (outer)joins. Furthermore, retrieving set-valued attributes separately from single-valued attributes simplifies formatting (converting) the SQL query results into OPM class instances.

3.2 Processing Update Queries

OPM insert, delete, or update queries can be processed using insert, delete, or update methods, respectively. Object insertions, deletions, and updates refer to individual OPM class instances and therefore involve their `Oid`'s. Since `Oid`'s cannot be accessed directly by users, OPM class instances are identified in queries using conditions involving regular

attributes. The mapping between the values of these attributes and internal `Oid` values can be achieved using retrieval queries (e.g., via one of the approaches mentioned above).

Object insert, delete, or update queries can be also translated directly into SQL transactions. For this translation, `Oid` values do not have to be known ahead of time.

4 The OPM Query Translator

This section describes some of the features of the query execution plans implemented by the OPM Query Translator.

4.1 Query Plan Features

The OPM Query Translator supports the following query execution options for Sybase and Oracle DBMSs.

4.1.1 Execution Options

1. Raw SQL: implemented for query retrieval for both Sybase and Oracle.
2. Stored Procedures: implemented for query retrieval and update for Sybase. Oracle stored procedure support is forthcoming.

Raw SQL is used for ad hoc query specifications, specifications that cannot be "canned" ahead of time using stored procedures. Raw SQL is required for condition evaluation since most condition specifications are more complex than retrieval by `Oid`, the basic retrieval parameter for stored procedures. Raw SQL is also required for ad hoc path or branch evaluation, paths or branches not specified in a schema and implemented by a derived composition attribute stored procedure. Lastly, raw SQL is required for retrofitted OPM databases operating without stored procedures.

Stored procedures provide an intermediate layer of abstraction between the OPM Query Translator and the underlying DBMS. They have the advantage of not needing compilation by the DBMS for execution. Use of stored procedures also reduces network traffic: instead of shipping a long complex SQL string across the net to the server, a stored procedure call is usually pretty terse. Stored procedure queries are useful for certain fixed queries, such as retrieval by `Oid`.

For updates, stored procedures also provide a valuable service in handling integrity constraints and version management.

Stored procedures currently follow certain name conventions, documented in the `README.opmsp` file. Insert procedures names, begin with `OPMI_`. For example, the procedure for inserting into class `PERSON` is called `OPMI_PERSON`. Delete procedures begin with `OPMD_`. Differential updates begin with `OPMdU_` Procedures for retrieval begin with `OPMvG_`. Set-valued attributes procedure names are formed by concatenating the set valued attribute name to the class name and stored procedure prefix. For example, the insert procedure for the set valued attribute `project` in class `PERSON` is `OPMI_PERSON_project`.

Stored procedure arguments usually begin with the input arguments `@_oid`, as well as `@_version` if versioning is involved. The other arguments usually involve OPM level attribute

names. In the procedure `OPMvG_` ("verbose get"), arguments can also involve `REP` attributes concatenated to the original attribute name. (`REP`'s serve a similar function as class `ID`'s in the sense that they represent a class instance. They differ from `ID`'s in that they may be versioned, they allow nulls, and they may have duplicate instances. They provide more human readable information for real world applications over a raw `Oid` number of an abstract attribute value.)

4.1.2 Projections

The OPM Query Translator supports two types of projections which may be used together.

1. Denormalized tuple output.

2. `Oid` based object output within the denormalized output.

COnsider the following query example:

```
SELECT name = @pname, @p (age, degree_date, degree, field)
FROM   @p in PERSON, @pname in @p.name, @pfield in @p.field
WHERE  @pfield = "Genetics";
```

(Note: the `@variable` convention is useful to adopt for easy identification of variables as well as for avoiding potential name conflicts with schema identifiers.)

In this example, the output variables `@pname` and `@p` (class instance `Oid` for `PERSON`) provide two columns in the denormalized tuple output. The attributes `age`, `degree_date`, `degree`, and `field` specify the `Oid` based projection output attributes for a class instance of `PERSON` specified by variable `@p`.

Condition evaluation generates the first level denormalized output. This level of evaluation takes into account restrictions on qualifying set-valued attributes instances and join correlations specified in the condition. The denormalized tuple projection is efficiently implemented in one main SQL query, with possibly one or more subqueries within the main query, and is useful for retrieving raw values for higher level software. For example, a query processor may use "get tuple" to retrieve values for condition evaluation.

For each abstract or class output variable in the denormalized output, the user may additionally specify output attributes. Unlike the denormalized tuple output, these output attribute instances are not constrained by the condition in the manner described above. They are constrained only by the class instance `Oid` of a variable instance. `Oid` based projection results in all attribute instances for a given class instance and may be assembled into an "object" data output consisting of variously structured single and set-valued attributes, as well as simple and tuple attributes. In addition, implied retrievals, such as `REP`'s are also handled at this level.

While `Oid` based projection provides a structurally richer output, retrieval is an order of magnitude slower than the denormalized output. For each class instance in the denormalized output variable (`@p` in this example), a relational query, or queries, are resubmitted to retrieve all specified attributes for one `Oid`. The retrieval of these attributes by `Oid` may be accomplished using raw SQL or stored procedures.

Both these types of projections have their appropriate place in application strategy. The first provides efficient retrieval of raw values in a tuple, and as mentioned earlier, may be used to retrieve raw values for a higher level query processor. The second provides a "full object" retrieval for a single class instance. The second is inefficient for large volume query retrievals, but may be useful for queries like "get me the first N object instances" based on a condition specification, where N is a small number. `Oid` based projection may also be used for connection oriented applications, where the user submits the query, but views one class instance result at a time, and retrieves the next class instance from the query result by hitting some "next" button. For such an application interaction, the user has the option to quit any time without retrieving all instances. `Oid` based projections may also be used in connection-less applications, such as Web based CGI retrievals. The CGI application may use the "get tuple" projection to efficiently retrieve all the `Oid` and `REP`'s qualified in a condition. The user can then select one class instance from this result list and use the "get object" projection to look at one full class instance at a time.

4.1.3 The Translator for a specific DBMS

A sample unix session of the OPM Query Translator for Sybase is shown below. `metadata.so` is the dynamically linked metadata shared object file. The ascii file `sel01.oqt` contains the OPM Query Translator query specification.

```
% oqt4syb -s metadata.so -q sel01.oqt
name "John Smith"
@p 100456
  age 35
  Tuple
    degree_date "6/20/1995"
    degree "Ph.D"
    field "Genetics"
  Tuple
    degree_date "6/15/1984"
    degree "B.S."
    field "Biology"

...
%
```

Analogously, `oqt4ora` may be used for Oracle DBMS.

(Note that component attributes are automatically nested inside their tuple attribute specification in the output presentation.)

The OPM Query Translator for Sybase is provided as a utility tool for querying data. It's source code provides an example of how driver code may be written. Some application developers may be content to embed `opm4syb` inside a PERL script. Others, may prefer to compile and link with the source code directly due to efficiency considerations or for handling more customized needs. Future considerations also include making the query translator (for a specific DBMS) a part of the PERL language. In this manner, PERL developers may write script applications without having to look at, or compile and link with the C++ code. Another possibility involves incorporating the translator into a concurrent server with socket based communications. An open OPM java development kit, with API and sample source code of sample applications or applets, will allow users to develop or customize applet front ends. The use of ILU (*Inter-Language Unification*), a freely available subset of CORBA from Xerox PARC [7] for an OPM API with java front ends [8] is another possibility under consideration.

4.2 Metadata

Metadata processing can be slow depending on the size of one's schema. The original `.OPM` file must be parsed. The `.MAP` and `.spmap` files also need to be parsed to establish the correspondence between OPM and relational constructs (including the stored procedures). Derived metadata information, such all (including transitive) superclasses of a class, are also inferred in order to support the handling of inheritance, rules, etc. Note that for command line utilities, or "one shot" Web gateways, reloading metadata can be an expensive process each time a query is executed.

For the OPM query translator and related tools, metadata is pre-processed. To further speed up the loading process, dynamic linking is utilized with shared object libraries. In this case, the shared object library is the metadata itself. The application does not need to regenerate its own internal metadata. It merely sets a pointer into the shared object image. The C++ class `Dlopm` ("dynamic link OPM") provides the API for this metadata.

The shared object (`.so`) file is generated by preparing a `C` source file that contains array initializations of C structures representing OPM metadata. This file is then compiled. `opm2meta` takes an `.OPM` and `.MAP` file and generates `.c` file. `makeso` is a unix script that compiles and links the `.c` file to generate the `.so` file using the operating system compiler and linker. A sample session is shown below:

```
% opm2meta -u szeto -p pw -o shotgun.OPM -m shotgun.MAP \
    -s shotgun.spmap -c shotgun.c
% makeso shotgun
```

4.3 Modules

The OPM Query Translator consists of several modules that are intimately related. The OqJoin class handles the generation of SQL joins and SQL queries in general. The OqVar class handles the evaluation of variables. OqCond handles condition evaluation. All of these modules are used by OqtCom, the common translator module, subclassed by OqtRaw, the raw SQL translator, and OqtSsp, the Sybase stored procedure translator.

For purposes of testing, debugging, studying the output SQL translation, as well as applications development, the SQL translation drivers `oqtraw` and `oqtssp` are provided in the developer's kit. Sample unix sessions are shown below:

```
% oqtraw -v 2 -s metadata.so -q sel01.oqt -T tupout.sql -O oidout.sql
% oqtssp -v 2 -s metadata.so -q sel01.oqt -o out.sql
```

The, `-v 2` or "verbosity level 2" tells the translator to also generate projection descriptors information as comments in the output SQL files. The `tupout.sql` file contains the tuple projection output SQL. The `oidout.sql` file contains the SQL template to be reused for retrieving an object instance. The `Oid` parameter is replaced by the actual value in the query resubmission. The `out.sql` file for stored procedure provides a cursor based output that combines both tuple and object output using stored procedures. This SQL output may be executed directly using Sybase's `isql` utility for testing. For update queries, `oqtssp` is sufficient. For retrieval queries, one gets a relational dumps from `isql`, but no repackaging of the results in an object format, hence, it is not sufficient for serious application use.

4.4 Using the OPM Query Translator

This section provides a brief overview of how the OPM Query Translator can be employed. Briefly, employing the OPM Query Translator involves three stages: (1) creating an OPM-based database, (2) preparing the metadata, and (3) using the translator. The following sections will describe each of these stages in more detail.

4.4.1 Creating an OPM Database

We will describe the process of creating an OPM database using Sybase as an underlying DBMS. OPM databases can also be created using Oracle or some other DBMS, or an existing relational database can be retrofitted with an OPM view using the OPM retrofitting tools.

First an OPM schema is specified using an Ascii text editor or the OPM Schema Editor (`oped`). For example, suppose the Ascii file `shotgun.OPM` contains an OPM schema. Then the OPM Schema Translator `opm2dbms` (see [4]) can be used to generate files containing the

Sybase database definitions and procedures from this `.OPM` file. These files can be loaded into the database with `isql`.

```
% opm2dbms shotgun.OPM
...
% isql -Uusername -Ppassword < shotgun_relations.SYB11
% isql -Uusername -Ppassword < shotgun_datarules.SYB11
...
```

The customizable PERL script `opm2syb` can also be used to automate this process. `opm2syb` is a driver script that runs `opm2dbms` and `isql` and is usually sufficient for most database creation cases.

```
% opm2syb -u username -p password -r shotgun.OPM
```

The user can choose to load the file containing the referential integrity triggers later with `isql`, after bulk loading data from other sources. (The '-r' suppresses loading referential integrity files.)

Next, stored procedures can be created and loaded into the database. The `opmsp` utility takes the `.OPM` file and `.MAP` file (the latter generated by `opm2dbms`) and generates `.opmsp` files for Sybase.

```
% opmsp shotgun.OPM shotgun.MAP
...
% isql -Uusername -Ppassword < ASSEMBLE.opmsp
% isql -Uusername -Ppassword < CONSTRAINT.opmsp
% isql -Uusername -Ppassword < CONNECTION_TABLE.opmsp
% isql -Uusername -Ppassword < CONTIG_MAP.opmsp
...
```

Besides generating the .opmsp stored procedure files, `opmsp` generates a .spmap file. The .spmap file contains stored procedure mapping information for the metadata. A similar procedure may be used with Oracle's `sqlplus` utility. As with `opm2syb`, a driver script, `osp2syb`, can be used for running `opmsp` and `isql` to automate the above process.

```
% osp2syb -u username -p password shotgun.OPM
...
```

4.4.2 Preparing Metadata

Metadata is generated by compiling a C file as a shared object library. `opm2meta` takes an `.OPM` and `.MAP` file and generates a `.c` file. `makeso` is a unix script that compiles and links the `.c` file into a `.so` file. The Sybase username and password is for storage in the binary metadata file, for later login to Sybase. (One can put in dummy values if one wants to handle Sybase login separate from the metadata.) A sample session is shown below:

```
% opm2meta -u szeto -p pw -o shotgun.OPM  -m shotgun.MAP \
    -s shotgun.spmap -c shotgun.c
% makeso shotgun
```

4.4.3 Using the OPM Query Translator

Several sample sessions illustrating the employment of the OPM Query Translator are shown below:

```
# To get usage information
% oqtssp
...
% oqtraw
...
% oqt4syb
...
% oqtssp -s shotgun.so -q sel01.oqt -o out.sql
% more out.sql

% oqtraw -s shotgun.so -q sel01.oqt -T tupout.sql -O oidout.sql
% more tupout.sql
% more oidout.sql

% more ins01.oqt
% oqtssp -s shotgun.so -q ins01.qlt -o out.sql
% isql -Uusername -Ppassword -e < out.sql

% more sel01.qlt
% oqt4syb -s shotgun.so -q sel01.qlt
...
```

References

[1] Bertino, E., Negri, M., Pelagatti, G., and Sbattella, L., Object-Oriented Query Languages: The Notion and the Issues, *IEE Transaction on Knowledge and Data Engineering*, **4**, 3 (June 1992), pp. 223-237.

[2] Chen, I.A., and Markowitz, V.M., The Object-Protocol Model (Version 4.0), Lawrence Berkeley Laboratory Technical Report LBL-32738 (revised), 1995.

[3] Chen, I.A., and Markowitz, V.M., Mapping Object-Protocol Schemas into Relational Database Schemas and Procedures, Lawrence Berkeley Laboratory Technical Report LBL-33048 (revised), 1995.

[4] Chen, I.A., and Markowitz, V.M., OPM Schema Translator 4.0, Reference Manual, Lawrence Berkeley Laboratory Technical Report LBL-35582 (revised), 1995.

[5] *The Object Database Standard: ODMG-93*, Release 1.2, Cattell, R. G. G. (ed), Morgan Kaufmann, 1996.

[6] Rundensteiner, E.A., MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases, *Proc. of 18th VLDB Conference*, 1992.

[7] ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[8] http://coho.stanford.edu/~hassan/Java/Jylu/

A An OPM Example Schema

```
CONTROLLED VALUE CLASS OVERLAP_PROG   { "insert", "overlap" }

CONTROLLED VALUE CLASS AGE   { 0-100 }
CONTROLLED VALUE CLASS SENIOR_AGE { 65-100 }

OBJECT CLASS FRAGMENT
   DESCRIPTION: "DNA fragment"
   ID: fragment_id
   ATTRIBUTE fragment_id: [1,1] INTEGER
   ATTRIBUTE sequence: [0,1] VARCHAR(200)
   ATTRIBUTE length: [0,1] INTEGER
   ATTRIBUTE owner
                    DERIVATION: !owns [PERSON]
   ATTRIBUTE scientist_owner
                    DERIVATION: !owns [SCIENTIST]

OBJECT CLASS CONNECTION_TABLE
   DESCRIPTION: "connection table"
   ID: table_id
   ATTRIBUTE table_id: [1,1] INTEGER
   ATTRIBUTE left_entry: [0,1] FRAGMENT
   ATTRIBUTE right_entry: [0,1] FRAGMENT
   ATTRIBUTE distance: [0,1] INTEGER

OBJECT CLASS CONTIG_MAP
   DESCRIPTION: "contig map"
   ID: contig_id
   ATTRIBUTE contig_id: [1,1] INTEGER
   ATTRIBUTE note: [0,1] TEXT
   ATTRIBUTE entries(entry, position): set-of [0,] ([0,1] FRAGMENT, [0,1] INTEGER)
   ATTRIBUTE location: set-of [1,] CHAR(40)
   ATTRIBUTE constructed_by: [0,1] PERSON
   ATTRIBUTE owner
                DERIVATION: !owns [PERSON]
   ATTRIBUTE no_entry
                DERIVATION: count of entry

OBJECT CLASS PERSON
```

```
   DESCRIPTION: "person"
   ID: person_id
   ATTRIBUTE person_id: [1,1] INTEGER
   ATTRIBUTE name: [0,1] CHAR(80)
   ATTRIBUTE age: [0,1] AGE
   ATTRIBUTE owns: set-of [1,] CONTIG_MAP or FRAGMENT
   ATTRIBUTE address : [0,1] VARCHAR(250)
   ATTRIBUTE (degree_date, degree, field) :
     list-of [1,] ([0,1] DATETIME, [0,1] CHAR(20), [0,1] CHAR(50))

OBJECT CLASS SCIENTIST isa PERSON
   DESCRIPTION: "scientist"
   ATTRIBUTE affiliation: [0,1] CHAR(80)
   ATTRIBUTE project: set-of [1,] CHAR(40)

PROTOCOL CLASS CONSTRUCT
   DESCRIPTION: "construct a contig map"
   ID: construct_id
   EXPANSION: (OVERLAP or CONSTRAINT) , ASSEMBLE
   ATTRIBUTE construct_id: [1,1] INTEGER
   ATTRIBUTE fragments: set-of [1,] FRAGMENT      input
   ATTRIBUTE contig: [1,1] CONTIG_MAP             output

PROTOCOL CLASS OVERLAP
   DESCRIPTION: "compare fragments using computer programs"
   ID: overlap_id
   ATTRIBUTE overlap_id: [1,1] INTEGER
   ATTRIBUTE fragments: set-of [1,] FRAGMENT      input isa CONSTRUCT.fragments
   ATTRIBUTE connect_table: set-of [1,] CONNECTION_TABLE    output
   ATTRIBUTE (program_name, program_version): [0,1] (CHAR(10), CHAR(6))

PROTOCOL CLASS CONSTRAINT
   DESCRIPTION: "manually compare fragments using constraints"
   ID: constraint_id
   ATTRIBUTE constraint_id: [1,1] INTEGER
   ATTRIBUTE fragments: set-of [1,] FRAGMENT       input isa CONSTRUCT.fragments
   ATTRIBUTE connect_table: set-of [1,] CONNECTION_TABLE    output
   ATTRIBUTE constraint_type: [0,1] CHAR(20)

PROTOCOL CLASS ASSEMBLE
```

```
DESCRIPTION: "assemble contigs"
ID: assemble_id
ATTRIBUTE assemble_id: [1,1] INTEGER
ATTRIBUTE connect_table: set-of [1,] CONNECTION_TABLE
                  input from OVERLAP via connect_table
                          or CONSTRAINT via connect_table
ATTRIBUTE contig_map: [1,1] CONTIG_MAP
                  output isa CONSTRUCT.contig
```

B Syntactic Definition for the OPM Query Language

```
<OPM query> ::= <SELECT query> | <INSERT query> | <UPDATE query> |
                <DELETE query>
              ;
<SELECT query> ::= <select statement> <from statement>
                   <optional where statement> ';'
                 | <select statement> <from statement>
                   <optional where statement>
                   ORDER BY <order expressions> ';'
                 ;
<select statement> ::= SELECT <select declarations>
                     | SELECT DISTINCT <select declarations>
                     ;
<select declarations> ::= <select declaration>
                        | <select declarations> ',' <select declaration>
                        ;
<select declaration> ::= <attribute alias> '=' <select expression>
                       | <select expression>
                       | <single class select declaration>
                       ;
<select expression> ::= <expression>
                      | <expression> '(' <attribute list> ')'
                      | <expression> '(' '*' ')'
                      ;
<attribute list> ::= <attribute> | <attribute list> ',' <attribute>
                   ;
<single class select declaration> ::= '(' <attribute list> ')' | '*'
                                    ;
<attribute> ::= <projection composition>
              | <attribute alias> '=' <projection composition>
              ;
<from statement> ::= FROM <variable declarations>
                   | FROM <class name>
                   ;
<variable declarations> ::= <variable declaration>
                          | <variable declarations> ',' <variable declaration>
                          ;
<variable declaration> ::= <simple variable> IN
                           <class expression> <optional all versions>
                         | '(' <variable list> ')' IN <simple variable> '.'
```

```
                          '(' <projection element list> ')'
                           <optional all versions>
                        ;
<optional all versions> ::= <empty> | ALL VERSIONS
                         ;
<simple variable> ::= <identifier>
                    ;
<variable list> ::= <simple variable>
                  | <variable list> ',' <simple variable>
                  ;
<projection element list> ::= <projection element>
                            | <projection element list> ',' <projection element>
                            ;
<class expression> ::= <class name> | <path expression>
                     ;
<path expression> ::= <simple variable> '.' <projection composition>
                    ;
<projection composition> ::= <projection element>
                           | <composition element list> <projection element>
                           ;
<composition element list> ::= <composition element>
                             | <composition element list> <composition element>
                             ;
<composition element> ::= <attribute name> '[' <class name> ']'
                        | '!' <attribute name> '[' <class name> ']'
                        | <attribute name> '.'
                        ;
<projection element> ::= <attribute name> '[' <class name> ']'
                       | '!' <attribute name> '[' <class name> ']'
                       | <attribute name>
                       ;
<optional where statement> ::= <empty> | WHERE <conditions>
                             ;
<order expressions> ::= <expression> <optional direction>
                      | <order expressions> ',' <expression>
                        <optional direction>
                      ;
<optional direction> ::= <empty> | ASC | DESC
                       ;
<expression> ::= <variable> | <primitive value> | <set of values>
               ;
<variable> ::= <simple variable> | <path expression>
```

```
            ;
<primitive value> ::= <number> | <string>
                    ;
<set of values> ::= '{' <set of numbers> '}'
                  | '{' <set of strings> '}'
                  ;
<set of numbers> ::= <number>
                   | <set of numbers> ',' <number>
                   ;
<set of strings> ::= <string> | <set of strings> ',' <string>
                   ;
<conditions> ::= <atomic condition>
               | '(' <conditions> ')'
               | <conditions> AND <conditions>
               | <conditions> OR  <conditions>
               ;
<atomic condition> ::= <expression> <is null op>
                     | <expression> <bin op> <expression>
                     | <expression> <all op> <variable>
                     | <expression> <match op> <string>
                     ;
<is null op> ::= IS NULL | IS NOT NULL
               ;
<bin op> ::= <comp op> | <in op> | <contains op>
           ;
<all op> ::= <comp op> ALL
           ;
<comp op> ::= '=' | NE | '>' | GE | '<' | LE
               ;
<contains op> ::= CONTAINS | NOT CONTAINS
                ;
<in op> ::= IN | NOT IN
          ;
<match op> ::= MATCH | NOT MATCH
             ;
<attribute name> ::= <identifier>
                   ;
<attribute alias> ::= <idnetifier>
                    ;
<class name> ::= <identifier>
               ;
<number> ::= <integer> | <real>
```

```
          ;
<INSERT query> ::= INSERT <class name> '(' <assign values> ')'  ';'
                 | INSERT <class name> '(' <assign values> ')'
                   AS <an id value> ';'
                 ;
<assign values> ::= <assign attribute values>
                  | <assign values> ',' <assign attribute values>
                  ;
<assign attribute values> ::= <simple single assignment>
                            | <simple multi assignment>
                            | <tuple single assignment>
                            | <tuple multi assignment>
                            ;
<simple single assignment> ::= <attribute name> '=' <attr value>
                             | <attribute name> '=' NULL
                             ;
<id assignment> ::= <attribute name> '=' <a primitive value>
                  ;
<attr value> ::= <a primitive value> | <an id value>
               ;
<a primitive value> ::= <integer> | <string> | <real>
                      ;
<an id value> ::= <class name> '[' <id values> ']'
                ;
<id values> ::= <id assignment> | <id values> ',' <id assignment>
              ;
<simple multi assignment> ::= <attribute name> '='
                              '{' <multiple assignments> '}'
                            ;
<multiple assignments> ::= <attr value>
                         | <multiple assignments> ',' <attr value>
                         ;
<tuple single assignment> ::=  '(' <component names> ')' '='
                               '(' <multiple assignments> ')'
                            |  '(' <component names> ')' '=' NULL
                            ;
<tuple multi assignment> ::= '(' <component names> ')' '='
                             '{' <multi comp assignments> '}'
                           ;
<multi comp assignment> ::= '(' <multiple assignments> ')'
                          | <multi comp assignments> ','
                            '(' <multiple assignments> ')'
```

```
                          ;
<component names> ::= <attribute name>
                    | <component names> ',' <attribute name>
                    ;
<UPDATE query> ::= UPDATE <simple variable> '(' <attribute updates> ')'
                   <from statement> <optional where statement> ';'
                 ;
<DELETE query> ::= DELETE <simple variable> <from statement>
                   <optional where statement> ';'
                 ;
<attribute updates> ::= <attribute update>
                      | <attribute updates> ',' <attribute update>
                      ;
<attribute update> ::= <attribute insert statement>
                     | <attribute modify statement>
                     ;
<attribute insert statement> ::= ADD <assign attribute values>
                                 ;
<attribute modify statement> ::= SET <assign attribute values>
                                 ;
```

□

C OPM Query Examples Expressed on GDB 6.0

These queries were suggested by Ken Fasman, informatics director of GDB, Johns Hopkins School of Medicine, Baltimore. The queries are based on the GDB 6.0 available at http://gdbgeneral.gdb.org/.

Query 1. Retrieve attributes of all Maps containing a MapElement associated with GenomicSegments CFTR or DMD.

```
SELECT M(displayName, accessionID, units, objectClass, minCoord, maxCoord)
FROM M IN Map,
     GS IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName
WHERE GS = "cftr" OR GS = "dmd"
ORDER BY M.displayName;
```

Query 2. Retrieve attributes of all Maps containing MapElements associated with GenomicSegments CFTR and DMD.

```
SELECT M(displayName, accessionID, units, objectClass, minCoord, maxCoord)
FROM M IN Map,
     GS1 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName,
     GS2 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName
WHERE GS1 ="cftr" AND GS2 ="dmd"
ORDER BY M.displayName;
```

Note that this query retrieves `Maps` with possibly distinct `MapElements` associated with the `GenomicSegments` CFTR and DMD. The original statement of the query asked for "Maps containing *a* MapElement associated with GenomicSegments CFTR and DMD". Such a query could be expressed as

```
SELECT M(displayName, accessionID, units, objectClass, minCoord, maxCoord)
FROM M IN Map,
     ME IN  M.!map[MapElement],
     GS1 IN  ME.segment[GenomicSegment]!dBObject[ObjectName]searchName,
     GS2 IN  ME.segment[GenomicSegment]!dBObject[ObjectName]searchName
WHERE GS1 ="cftr" AND GS2 ="dmd"
ORDER BY M.displayName;
```

However the `segment` attribute of `MapElement` is single-valued: that is, a `MapElement` can be associated with at most one `GenomicSegment`. In addition, one would not expect the same GenomicSegment to be associated with both of the `ObjectNames` "cftr" and "dmd". Consequently this second OPM query will always have an empty result.

Query 3. Retrieve attributes of all Linkage and Cytogenetic Maps containing MapElements associated with GenomicSegments CFTR and DMD.

```
SELECT M(displayName, accessionID, units, objectClass, minCoord, maxCoord)
FROM M IN Map,
     GS1 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName,
     GS2 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName,
     OC IN M.objectClass
WHERE GS1 ="cftr" AND GS2 ="dmd"
  AND OC IN { "string", "LinkageMap", "CytogeneticMap" }
ORDER BY M.displayName;
```

The comment for query 2 above also applies to query 3.

Query 4. Retrieve attributes of all Maps containing the cytogenetic region q21-q31. This query involves looking for maps of chromosome 1 that contain (different) MapElements pointing to GenomicSegments q21 and q31.

```
SELECT M(displayName, accessionID, units, objectClass, minCoord, maxCoord)
FROM M IN Map,
     C IN M.chromosome[Chromosome]searchName,
     GS1 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName,
     GS2 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName
WHERE C = "1" AND GS1 = "q21" AND GS2 = "q31"
ORDER BY M.displayName;
```

Query 5. Retrieve the coordinates of MapElements linked to q21 and q31 in Maps containing the cytogenetic region q21-q31.

```
SELECT ME(map[Map]accessionID, LFM_coord, RFM_coord)
FROM ME IN MapElement,
     M IN ME.map[Map],
     C IN M.chromosome[Chromosome]searchName,
     GS1 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName,
     GS2 IN  M.!map[MapElement]segment[GenomicSegment]!dBObject[ObjectName]searchName
WHERE C = "1"
  AND GS1 = "q21"
  AND GS2 = "q31"
  AND ME.segment[GenomicSegment]searchName in { "string", "q21", "q31" };
```

Note that this query could not be expressed as a single query in the OPM 4.0 version of OPM-QL. This is because it would not be possible to fix the map `M` for a particular `MapElement` `ME` and then find distinct `GenomicSegments` associated with that `MapElements` of that `Map`. Instead it would have been necessary to do a single query to find the accession numbers for `Maps` containing the region q21-q31 (Query 4), and then use these accession numbers in a second query:

```
SELECT ME(map[Map]accessionID, LFM_coord, RFM_coord)
FROM ME IN MapElement
WHERE ME.map[MAP]accessionID in {"string", "GDB:785407", "GDB:794763"}
  AND ME.segment[GenomicSegment]searchName in { "string", "q21", "q31" };
```

Query 6. Retrieve all relevant attributes for a particular map in a given subclass by accessionID.

```
SELECT CM(accessionID, addDate, chromosome, comment, displayName, likelihoodType,
       maxCoord, minCoord, modDate, objectClass, publiclyEditable, units, version)
FROM CM IN CytogeneticMap
WHERE CM.accessionID = "GDB:785407";
```

Query 7. Retrieve all relevant attributes for all the elements of a particular map between two coordinates.

```
SELECT ME(LFM_coord, RFM_coord, accessionID, coordinate, displayName,
       searchName, draw, point, segment, sortCoord, style, tier)
FROM ME IN MapElement, M IN ME.map
WHERE M.accessionID = "GDB:785407"
      AND ME.LFM_coord >= 10866
      AND (ME.coordinate != 10866 OR ME.point != "End" )
      AND ME.RFM_coord <= 1122
      AND (ME.coordinate != 1122 OR ME.point != "Start" )
ORDER BY ME.sortCoord;
```